# CMSC 420
# Data Structures[1]

David M. Mount
Department of Computer Science
University of Maryland
Spring 2023

[1]Copyright, David M. Mount, 2023, Dept. of Computer Science, University of Maryland, College Park, MD, 20742. These lecture notes were prepared by David Mount for the course CMSC 420, Data Structures, at the University of Maryland. Permission to use, copy, modify, and distribute these notes for educational purposes and without fee is hereby granted, provided that this copyright notice appear in all copies.

# Lecture 1: Introduction to Data Structures

**Algorithms and Data Structures:** The study of data structures and the algorithms that manipulate them is among the most fundamental topics in computer science. Most of what computer systems spend their time doing is *storing*, *accessing*, and *manipulating* data in one form or another. Some examples from computer science include:

**Information Retrieval:** List the 10 most informative Web pages on the subject of "how to treat high blood pressure?" Identify possible suspects of a crime based on fingerprints or DNA evidence. Find movies that a Netflix subscriber may like based on the movies that this person has already viewed. Find images on the Internet containing both kangaroos and horses.

**Geographic Information Systems:** How many people in the USA live within 25 miles of the Mississippi River? List the 10 movie theaters that are closest to my current location. If sea levels rise 10 meters, what fraction of Florida will be under water?

**Compilers:** You need to store a set of variable names along with their associated types. Given an assignment between two variables we need to look them up in a symbol table, determine their types, and determine whether it is possible to cast from one type to the other).

**Networking:** Suppose you need to multicast a message from one source node to many other machines on the network. Along what paths should the message be sent, and how can this set of paths be determined from the network's structure?

**Computer Graphics:** Given a virtual reality system for an architectural building walkthrough, what portions of the building are visible to a viewer at a particular location? (Visibility culling)

In many areas of computer science, much of the content deals with the questions of how to store, access, and manipulate the data of importance for that area. In this course we will deal with the first two tasks of storage and access at a very general level. (The last issue of manipulation is further subdivided into two areas, manipulation of numeric or floating point data, which is the subject of numerical analysis, and the manipulation of discrete data, which is the subject of discrete algorithm design.) An good understanding of data structures is fundamental to all of these areas.

What is a *data structure*? Whenever we deal with the representation of real world objects in a computer program we must first consider a number of issues:

**Modeling:** The manner in which objects in the real world are modeled as abstract mathematical entities and basic data types

**Operations:** The actions that are used to store, query, and manipulate these entities along with the formal meaning of these operations

**Representation:** The manner in which these entities are represented concretely in a computer's memory

**Algorithms:** The algorithms that are used to perform these operations

Note that the first two items above are essentially mathematical in nature, and deal with the "what" of a data structure, whereas the last two items involve the implementation issues and the "how" of the data structure. The first two essentially encapsulate the essence of

an *abstract data type* (or ADT). In contrast the second two items, the concrete issues of implementation, will be the focus of this course.

For example, you are all familiar with the concept of a *stack* from basic programming classes. This a sequence of *objects* (of unspecified type). Objects can be inserted into the stack by *pushing* and removed from the stack by *popping*. The pop operation removes the last unremoved object that was pushed. Stacks may be implemented in many ways, for example using arrays or using linked lists. Which representation is the fastest? Which is the most space efficient? Which is the most flexible? What are the tradeoffs involved with the use of one representation over another? In the case of a stack, the answers are all rather mundane. However, as data structures grow in complexity and sophistication, the answers are far from obvious.

In this course we will explore a number of different data structures, study their implementations, and analyze their efficiency (both in time and space). One of our goals will be to provide you with the *tools* and *design principles* that will help you to design and implement your own data structures to solve your own data storage and retrieval problems efficiently.

**Course Overview:** In this course we will consider many different abstract data types and various data structures for implementing each of them. There is almost never a single "best" data structure for a given task. For example, there are many common sorting algorithms: Bubble-Sort is easy to code but slow, Quick-Sort is very fast but not stable, Merge-Sort is stable but needs additional memory, and Heap-Sort needs no additional memory but is hard to code (relative to Quick-Sort). It will be important for you, as a designer of the data structure, to understand each structure well enough to know the circumstances where one data structure is to be preferred over another.

How important is the choice of a data structure? There are numerous examples from all areas of computer science where a relatively simple application of good data structure techniques resulted in massive savings in computation time and, hence, money.

Perhaps a more important aspect of this course is a sense of how to design new data structures, how to implement these designs, and how to evaluate how good your design is. The data structures we will cover in this course have grown out of the standard applications of computer science. But new applications will demand the creation of new domains of objects (which we cannot foresee at this time) and this will demand the creation of new data structures. It will fall on the students of today to create these data structures of the future. We will see that there are a few important elements which are shared by all good data structures. We will also discuss how one can apply simple mathematics and common sense to quickly ascertain the weaknesses or strengths of one data structure relative to another.

**Our Approach:** We will consider the design of data structures from two different perspectives: *theoretical* and *practical*. Our theoretical analysis of data structures will be similar in style to the approach taken in algorithms courses (such as CMSC 351). The emphasis will be on deriving asymptotic (so called, "big-O") bounds on the space, query time, and cost of operations for a given data structure. On the practical side, you will be writing programs to implement a number of classical data structures. This will acquaint you with the skills needed to develop clean designs and debug them.

The remainder of the lecture will review some material from your earlier algorithms course, which hopefully you still remember.

**Algorithmics:** *This background from CMSC 351. Review and familiarize yourself with this.*

It is easy to see that the topics of algorithms and data structures cannot be separated since the two are inextricably intertwined. So before we begin talking about data structures, we must begin with a quick review of the basics of algorithms, and in particular, how to measure the relative efficiency of algorithms. The main issue in studying the efficiency of algorithms is the amount of resources they use, usually measured in either the *space* or *time* used. There are usually two ways of measuring these quantities. One is a mathematical analysis of the general algorithm being used, called an *asymptotic analysis*, which can capture gross aspects of efficiency for all possible inputs but not exact execution times. The second is an *empirical analysis* of an actual implementation to determine exact running times for a sample of specific inputs, but it cannot predict the performance of the algorithm on all inputs. In class we will deal mostly with the former, but the latter is important also.[2]

**Running time:** For now let us concentrate on running time. (What we are saying can also be applied to space, but space is somewhat easier to deal with than time.) Given a program, its running time is not a fixed number, but rather a function. For each input (or instance of the data structure), there may be a different running time. Presumably as input size increases so does running time, so we often describe running time as a function of input/data structure size $n$, denoted $T(n)$. We want our notion of time to be largely machine-independent, so rather than measuring CPU seconds, it is more common to measure basic "steps" that the algorithm makes (e.g. the number of statements executed or the number of memory accesses). This will not exactly predict the true running time, since some compilers do a better job of optimization than others, but its will get us within a small constant factor of the true running time most of the time.

When discussing data structures, we do not usually talk about running time from the perspective of the individual operations. For example:

**Query time:** How long does it take to answer a query on a data structure containing $n$ objects?

**Update time:** How long does it take to insert (or delete) and object from a data structure containing $n$ objects?

**Bulk load time:** Given a collection of $n$ objects, how long does it take to construct a data structure containing all of them?

Even measuring running time as a function of input size is not really well defined, because, for example, it may be possible to sort a list that is already sorted, than it is to sort a list that is randomly permuted. For this reason, we usually talk about *worst case* running time. Over all possible inputs of size $n$, what is the maximum running time. It is often more reasonable to consider *expected case* running time where we average over all inputs of size $n$. When dealing with *randomized algorithms* (where the execution depends on random choices), it is common to focus on the worst case over all inputs (of a given size) and expected case over all random choices. Another example that is often used in data structure design is called *amortized analysis*, where the average is taken over a series of operations. Any one operation might be costly, but the overall average in any long sequence cannot be. We will usually do worst-case analysis, except where it is clear that the worst case is significantly different from the expected case.

[2]Of course, there is another aspect of complexity, that we will not discuss at length (but needs to be considered) and that is the software-engineering issues regarding the complexity of programming a correct implementation.

**Review of Asymptotics:** There are particular bag of tricks that most algorithm analyzers use to study the running time of algorithms. For this class we will try to stick to the basics. The first element is the notion of asymptotic notation. Suppose that we have already performed an analysis of an algorithm and we have discovered through our worst-case analysis that

$$T(n) \;=\; 13n^3 + 42n^2 + 2n\log_2 n + 3\sqrt{n}.$$

Asymptotic analysis is based on two assumptions, one reasonable and one questionable.

**We only care about large values of $n$:** This is reasonable. Given the blazing speed of modern computers, almost any data structure (and even no data structure at all) is quite efficient when the data size $n$ is small, say $n \leq 50$. Efficient data structures are most critical when $n$ is very large (think 100,000 up to over a billion).

**We don't care about constant factors:** Slight variations in how an algorithm is implemented, the programming language, and the operating system/hardware platform, will result in relatively small, constant factor, variations in running times. Since we want our solutions to be free of assumptions about programming language and operating system, it is reasonable to ignore these constant factors.

However, this perspective can be abused. For example, you have designed a simple program that achieves a running time of $T_1(n) = \lg n$ (where "$\lg n$" means "$\log_2 n$"). Your friend has designed a complex solution that achieves a running time of $T_2(n) = 10(\lg n)/(\lg\lg n)$. From a purely asymptotic perspective, $T_2$ is faster than $T_1$, but it is better only when $\lg\lg n \geq 10$. That is, $n \geq 2^{2^{10}}$, which (according to Wikipedia) exceeds the number of particles in the observable universe.

To simplify the description of $T(n)$, observe that as $n$ grows larger, the $n^3$ grows the fastest. Also, we can ignore the constant factor 13. Thus, the *asymptotic running time* is $T(n) = O(n^3)$.

**Digression about Notation:** Let us pause for a moment to explain the "=" in the assertion that "$T(n) = O(n^3)$". It is not being used in the usual mathematical sense. By definition, "$a = b$" implies "$b = a$". The use of "=" in asymptotic notation just a lazy way of saying $T(n)$ "is" on the order of $n^3$. It would *not* make sense to express this as $O(n^3) = T(n)$. A more proper way of writing this expression would be "$T(n) \in O(n^3)$", that is $T(n)$ is a member of the set of functions whose asymptotic growth rate is at most $n^3$. But, most of the world simply uses the "=" notation, and so shall we.

**Common Complexity Classes:** To get a feeling what various growth rates mean here is a summary. In data structures, our objective is usually to answer a query in time that is less than the size of the data set $n$, so in this class we will be primarily interested in the following complexity classes, which are collectively called *sublinear*.

$T(n) = O(1)$ : Great. This means your algorithm takes only *constant time*. You can't beat this. (Example: Popping a stack.)

$T(n) = O(\alpha(n))$ : The function $\alpha$ is the inverse of the famous *Ackerman's function*. Ackerman's function grows *insanely* rapidly, and hence its inverse grows *insanely* slowly. How slowly? If $n$ is less than the number of atoms in the visible universe, then $\alpha(n) \leq 5$. Thus, $\alpha(n)$ is a constant "for all practical purposes". However, formally it is *not* a constant. In the limit, as $n$ tends to infinity, $\alpha(n)$ also tends to infinity. It just gets there

extremely slowly! Believe or not, there are actually a number of data structures whose running times are $O(\alpha(n))$, but they are *not* $O(1)$. (Example: Disjoint-set union-find.)

$T(n) = O(\log \log n)$ : Super fast! For most practical purposes, this is as fast as a constant time. (Example: We will probably not show any examples this semester, but one example is a data structure for storing sets of integers, called a Van Emde Boas tree.)

$T(n) = O(\log n)$ : Very good. This is called *logarithmic* time, and is the "gold standard" for data structures based on making binary comparisons. It is the running time of binary search and the height of a balanced binary tree. This is about the best that can be achieved for data structures based on binary trees. Note that $\log 1000 \approx 10$ and $\log 1,000,000 \approx 20$ (log's base 2). (Example: Binary search.)

$T(n) = O((\log n)^k)$ : (where $k$ is a constant). This is called *polylogarithmic* time. Not bad, when simple logarithmic time is not achievable. We will often write this as $O(\log^k n)$. (Example: Orthogonal range searching, that is, counting the number of points in a $d$-dimensional axis-parallel rectangle.)

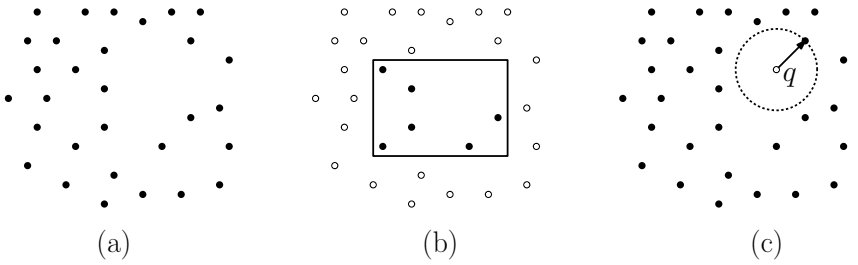



Fig. 1: (a) A point set, (b) orthogonal range search query, and (c) nearest-neighbor query.

$T(n) = O(n^p)$ : (where $0 < p < 1$ is a constant). An example is $O(\sqrt{n})$. This is slower than polylogarithmic (no matter how big $k$ is or how small $p$), but is still faster than linear time, which is acceptable for data structure use. (Example: Nearest neighbor searching in $d$-dimensional space.)

In an algorithms course, it is more common to focus on running times that grow at least linearly. These are described below.

$T(n) = O(n)$ : This is called *linear time*. It is about the best that one can hope for if your algorithm has to look at all the data. (Example: Enumerating the elements of a linked list.)

$T(n) = O(n \log n)$ : This one is famous, because this is the time needed to sort a list of numbers by means of comparisons. It arises in a number of other problems as well. (Example: Sorting, of course.)

$T(n) = O(n^2)$ : Called *quadratic time*. Okay if $n$ is in the thousands, but rough when $n$ gets into the millions. (Example: 3Sum: Given a list of $n$ numbers (positive and negative), do any three sum to zero?)

$T(n) = O(n^k)$ : (where $k$ is a constant). Called *polynomial time*. Practical if $k$ is not too large. (Example: Matrix multiplication.)

$T(n) = O(2^n), O(n^n), O(n!)$ : Called *exponential time*. Algorithms taking this much time are only practical for the smallest values of $n$ (e.g. $n \leq 10$ or maybe $n \leq 20$). (Example: Your favorite NP-complete problem... as far as anyone knows!)

## Lecture 2: Some Basic Data Structures

**Basic Data Structures:** Before we go into our coverage of complex data structures, it is good to remember that in many applications, simple data structures are sufficient. This is true, for example, if the number of data objects is small enough that efficiency is not so much an issue, and hence a complex data structure is not called for. In many instances where you need a data structure for the purposes of prototyping an application, these simple data structures are quick and easy to implement.

**Abstract Data Types:** An important element to good data structure design is to distinguish between the functional definition of a data structure and its implementation. By an *abstract data structure* (ADT) we mean a set of objects and a set of operations defined on these objects. For example, a *stack* ADT is a structure which supports operations such as *push* and *pop* (whose definition you are no doubt familiar with). A stack may be implemented in a number of ways, for example using an array or using a linked list. An important aspect of object-oriented languages, like Java, is the capability to present the user of a data structure with an *abstract view* of its function without revealing the methods with which it operates. Java's *interface/implements* mechanism is an example. To a large extent, this course will be concerned with the various approaches for implementing simple abstract data types and the tradeoffs between these options.

**Linear Lists:** A *linear list* or simply *list* is perhaps the most basic of abstract data types. A list is simply an ordered sequence of elements $\langle a_1, a_2, \ldots, a_n \rangle$. We will not specify the actual *type* of these elements here, since it is not relevant to our presentation. (In Java this would be handled through *generics*.)

The *size* or *length* of such a list is $n$. Here is a very simple, minimalist specification of a list:

`init()`: Initialize an empty list

`get(i)`: Returns element $a_i$

`set(i,x)`: Sets the $i$th element to $x$

`length()`: Returns the number of elements currently in the list

`insert(i,x)`: Insert element $x$ just prior to element $a_i$ (causing the index of all subsequent items to be increased by one).

`delete(i)`: Delete the $i$th element (causing the indices of all subsequent elements to be decreased by 1).

I am sure that you can imagine many other useful operations, for example searching the list for an item, splitting or concatenating lists, generating an iterator object for enumerating the elements of the list.

There are a number of possible implementations of lists. The most basic question is whether to use *sequential allocation* (meaning storing the elements sequentially in an array) or *linked allocation* (meaning storing the elements in a linked list). (See Fig. 2.) With linked allocation there are many other options to be considered. Is the list singly linked (each node pointing to its successor in the list), doubly linked (each node pointing to both its successor and predecessor), circularly linked (with the last node pointing back to the first)?

**Stacks, Queues, and Deques:** There are a few very special types of lists. The most well known are of course *stacks* and *queues*. We'll also discuss an interesting generalization, called the *deque* (see Fig. 3):

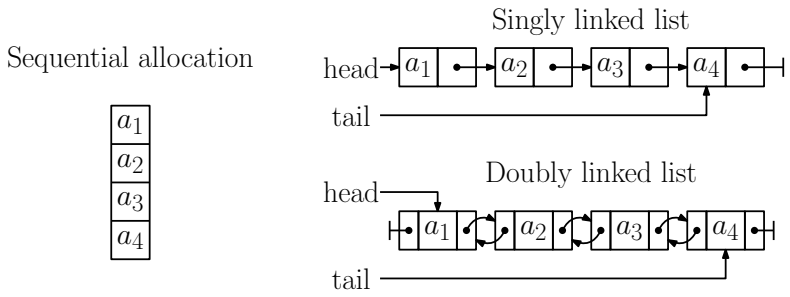



Fig. 2: Common types of list allocation.

**Stack:** Supports insertion (*push*) and removal (*pop*) from only one end of the list, called the stack's *top*. Stacks are among the most widely used of all data structures, and we will see many applications of them throughout the semester.

**Queue:** Supports insertion (called *enqueue*) and removal (called *dequeue*), each from opposite ends of the list. The end where insertion takes place is called the *tail*, and the end where removals occur is called the *head*.

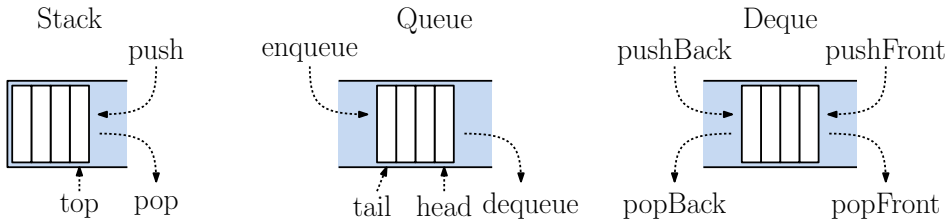



Fig. 3: Stack, Queue, and Deque.

**Deque:** This data structure is a generalization of stacks and queues, called a *double-ended queue* or "*deque*" for short. It supports insertions and removals from either end of the list.

The name is actually a play on words. It is written like "d-e-que" for a "double-ended queue", but it is pronounced like *deck*, because it behaves like a deck of cards, since you can deal off the top or the bottom.

Both stacks and queues can be implemented efficiently as arrays or as linked lists. Note that when a queue is implemented using sequential allocation (as an array) the head and tail pointers chase each other around the array. When each reaches the end of the array it wraps back around to the beginning of the array (see Fig. 4).

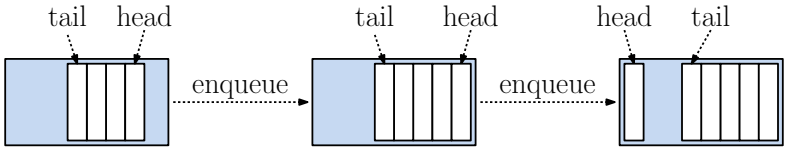



Fig. 4: Head/tail wrap-around in a queue.

**Dynamic Storage Reallocation:** When sequential allocation is used for stacks and queues, an important issue is what to do when an attempt is made to insert an element into an array

that is full. When this occurs, the usual practice is the allocate a new array of twice the size as the existing array, and then copy the elements of the old array into the new one. For example, if the initial stack or queue has eight elements, then when an attempt is made to insert a ninth element, we allocate an array of size 16, copy the existing eight elements to this new array, and then add the new element (see Fig. 6 below). When we fill this up, we then allocate an array of size 32, and when it is filled an array of size 64, and so on.

To understand whether such an approach is efficient, we will view the problem from the perspective of an amortized analysis.

**Amortized Analysis:** In an *amortized analysis* of a data structure, rather than the (worst-case) cost of each individual operation, we consider the average cost of maintaining the data structure over a long sequence of operations. The idea is that many operations are very cheap to perform, but occasionally we need to perform a major reorganization, which can be quite costly. More formally, suppose that the running time of any sequence of $m$ data structure operations (starting from an empty structure) is given by a function $T(m)$. Then we *amortized cost* of each operation is defined to be $T(m)/m$. Amortization is a way of establishing that, when taken as a whole, the costly reorganizations do not dominate the cheap ones. (The term comes from economics. It refers to spreading out expensive payments over small installments.)

To make this more formal in the case of a dynamic stack, let's define *cost model*, which described the "actual cost" of each operation:

**Cheap:** Each time we are asked to perform a push or pop, and the array is not reallocated, it costs one (+1) work unit for the operation.

**Expensive:** If a push would cause the array to overflow, we reallocated the current array of size $n$ into a new array of size $2n$. We charge $2n$ work units to perform the reallocation (which involves allocating the new array, initializing it, copying the $n$ elements over).[3] Finally, it costs 1 additional work unit to perform the actual push operation.

For example, suppose we start with an array of size $n = 1$, and we perform a sequence of $m$ pushes and pops. What is the total cost of all these operations? If we never did a reallocation (just alternated one push and one pop), the overall cost would be exactly $m$, and this is the lowest possible. But how high might the cost be? As an example, suppose, that we push 17 elements from an initially empty stack (see Fig. 5). When we push the second element, we overflow the array of size 1 and double the array size to $n = 2$. When we push the third element, we overflow the array of size 2 and double the array size to $n = 4$. Following this pattern, we see that we will overflow with push numbers 5, 9, and 17, resulting in the creation of new arrays of sizes 8, 16, and 32, respectively. The overall cost is 17 for the individual pushes and $2 + 4 + 8 + 16 + 32 = 62$ for the reallocations, for a total of $17 + 62 = 79$.

**Tokens, Charging, and Amortized Cost:** How can we analyze the overall cost for an arbitrary sequence of pushes and pops? We will show that in any sequence of $m$ pushes and pops, the

[3] You might wonder why we charged $2n$ for the reallocation cost and not, say $n$, since this would represent the cost of copying the elements from the old stack to the new stack. The convenient answer is that it doesn't matter, since the there is only a constant factor difference between them, and we will be happy with an asymptotic analysis of the running time. But this is always the case. If you were to increase the array size by some other function of $n$, the amortized cost may be different depending on whether the actual cost is based on the array size before reallocation versus the array size after reallocation.

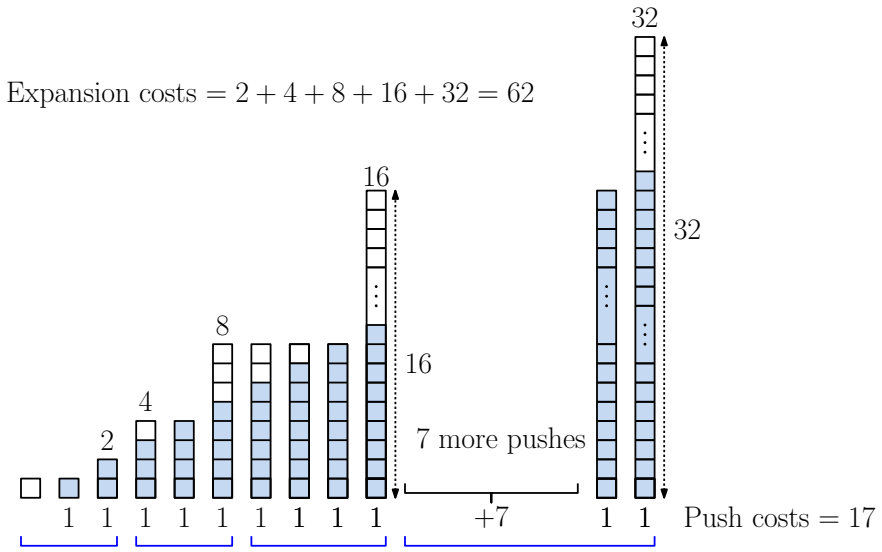



Fig. 5: Example of doubling reallocation for 17 push operations.

overall cost cannot exceed $5m$. (Intuitively, we pay a factor of at most 5 for the luxury of never running out of space in our stack.) Therefore, the amortized cost is $T(m)/m \leq 5$.

Why 5? Normally, when doing such an analysis, you would not know in advance what this value is. So, let's begin with the hypothesis that the amortized cost is some value $\tau$, and our analysis will reveal that $\tau = 5$ does the job. (Note that $\tau$ need not be an integer.)

Our approach is based on a *charging argument*. Here is a high-level overview. We will play the role of an accountant. Whenever the user asks us to perform a push or pop operation, we will assess the user a fixed charge of $\tau$ work tokens. We will store our work tokens in a bank account, from which we will make withdrawals to pay for the actual costs (as defined above). Most of the time, the operation will not involve any reallocation. Since each individual operation costs $+1$ work unit, we will use 1 of the $\tau$ tokens we collected to pay for the operation, and put the remaining $\tau - 1$ units is a bank account, from which we can draw to pay for future expansions. We will show that we always have enough accumulated in our bank account to pay for the expansion costs.

Why does this imply that the amortized cost is $\tau$? Suppose that we perform a total of $m$ operations. We have collected a total of $\tau m$ tokens during this time. For $1 \leq i \leq m$, let $T(i)$ denote the actual cost of performing the first $i$ operations (including both the individual operations and the expansions). Since we always have enough in our bank account to pay for any operation, it follows that $T(i) \leq \tau i$. This holds when $i = m$, and therefore, the amortized cost is $T(m)/m \leq (\tau m)/m = \tau$.

An example is shown in Fig. 6. Let suppose that the current array has size 8, and it contains 5 elements. (This is the natural state just after we have expanded an array of size 4.) We are asked to perform 4 consecutive pushes. The first 3 are all cheap and cost $1 + 1 + 1 = 3$. The last one causes the array to overflow to 9 elements, which necessitates that we reallocate to an array of size 16 for a total cost of $16 + 1 = 17$. The overall actual cost is $3 + 17 = 20$. We charged the user $\tau$ tokens for each operation. Thus, the total number of tokens in our bank account is $4 \cdot \tau$. In order to pay the actual costs, we need to select $\tau$ such that $4\tau \geq 20$. Solving for $\tau$ implies that $\tau = 5$.

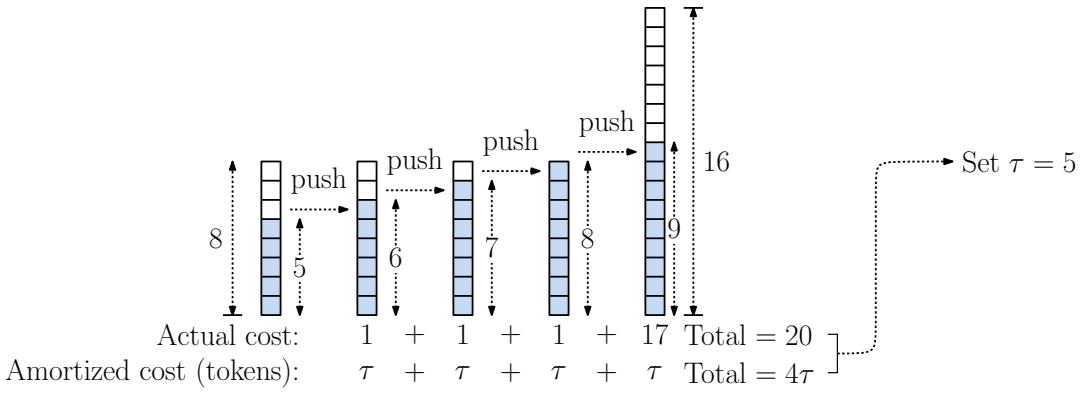



Fig. 6: Amortized analysis of doubling reallocation.

**Lemma:** Starting with an empty stack, any sequence of $m$ push/pop operations to our dynamically allocated stack has a total actual cost of at most $5m$.

**Proof:** Let's take the sequence of length $m$ and break it into subsequences that are easy to analyze. We split it just after each array reallocation. Let's call each such subsequence a *run*. (The runs are indicated by blue bars at the bottom of Fig. 5.) Each run, except possibly the last, consists of a sequence of cheap operations, and its last operation causes a reallocation. We will show that the number of tokens collected during each run suffices to pay the actual costs of the run. To simplify the analysis, we will ignore the first and last runs. (We will leave the full proof as an exercise.)

Let's imagine that we don't know that the final answer is 5. Let's call it $\tau$, and we will derive the fact that $\tau = 5$ works.

Consider the state of affairs at the start of a run. We have just expanded the array from the previous run. Let $n$ denote the size of the array just after the expansion. (Note that $n$ is completely generic, so this analysis applies to every run.) This means that we must have ended the previous run by overflowing an array of size $n/2$. Thus, there are $(n/2)+1$ elements in the current stack (the original $n/2$ plus the newly pushed element). There are $n - ((n/2) + 1) = (n/2) - 1$ available entries in this array.

In order to get to the end of the current run, we need to overflow this array. The fastest way to do this is to do nothing but pushing. After $(n/2) - 1$ pushes, we will fill up the array, and the next push will cause it to overflow. Thus, there are at least $((n/2) - 1) + 1 = n/2$ operations in the current run. (Note that there may also be pops mixed in, but this will only make the run longer, which means we have more tokens.) Therefore, we have collected at least $\tau(n/2)$ tokens throughout the run. We use one of these tokens to pay for each of the operations, which implies that we have banked at least $(\tau - 1)n/2$ tokens, which we can use to pay for the reallocation.

When run ends, we have just overflowed this array of size $n$ resulting in the creation of a new array of size $2n$. By our cost model, there is an actual cost of $2n$ to perform the expansion. We need to collect enough tokens to cover this cost, that is, we need to select $\tau$ so that

$$(\tau - 1)\frac{n}{2} \;\geq\; 2n \qquad \text{or equivalently} \qquad \tau \;\geq\; (2n)\frac{2}{n} + 1 \;=\; 5.$$

Therefore, by setting $\tau = 5$, we will always have sufficiently many tokens to pay for the

operations in the run. Since this holds for every run, it holds for the entire sequence of $m$ operations.

As an immediate consequence, we obtain the amortized analysis of our stack.

**Theorem:** The amortized cost of our doubling stack (under the given cost model) is $\leq 5$.

**Proof:** Letting $T(m)$ denote the total cost for the entire sequence of $m$ operations, the above lemma implies that $T(m) \leq 5m$. Therefore, the amortized time is $T(m)/m \leq 5$.

Okay, we have just shown that doubling works. What about other possible strategies?

**Linear growth:** What if, rather than doubling, we reallocate by adding a fixed increment, say going from $n$ elements to $n + c$ elements for some large integer constant $c \geq 1$? So, the allocated stack sizes would be multiples of $c$, $\langle c, 2c, 3c, \ldots, kc, \ldots \rangle$.

**Exponential growth:** What if, rather than doubling, we reallocated by adding a fixed constant factor, say going from $n$ elements to $\lceil cn \rceil$ for some constant $c > 1$? So, the allocated stack sizes would be powers of $c$, $\langle 1, c, c^2, c^3, \ldots, c^k, \ldots \rangle$.

**Doubly-exponential growth:** What if, rather than doubling, we squared the size of the array, going from $n$ to $n^2$? So, the allocated stack sizes would be $\langle c, c^2, c^4, c^8, \ldots, c^{2^k}, \ldots \rangle$.

We will leave the analyses of these cases as an exercise, but (spoiler alert!) the fixed-increment method will *not* yield a constant amortized cost, the constant-factor expansion will yield a constant amortized cost (but the factor 5 with change). The repeated squaring depends on which cost model we use (whether the cost depends on the size before reallocation or after reallocation).

**Multilists and Sparse Matrices:** Although lists are very basic structures, they can be combined in numerous nontrivial ways. A *multilist* is a structure in which a number of lists are combined to form a single aggregate structure. Java's `ArrayList` is a simple example, in which a sequence of lists are combined into an array structure. A more interesting example of this concept is its use to represent a *sparse matrix*.

Recall from linear algebra that a matrix is a structure consisting of $n$ rows and $m$ columns, whose entries are drawn from some numeric field, say the real numbers. In practice, $n$ and $m$ can be very large, say on the order of tens to hundred of thousands. For example, a physicist who wants to study the dynamics of a galaxy might model the $n$ stars of the galaxy using an $n \times n$ matrix, where entry $A[i, j]$ stores the gravitational force that star $i$ exerts on star $j$. The number of entries of such a matrix is $n^2$ (and generally $nm$ for an $n \times m$ matrix). This may be impractical if $n$ is very large.

The physicist knows that most stars are so far apart from each other that (due to the inverse square law of gravity), only a small number of matrices are significant, and all the others could be set to zero. For example, $n = 10,000$ but a star typically exerts a significant gravitational pull on only its 20 nearest stellar neighbors, then only $20/10,000 = 0.02\%$ of the matrix entries are nonzero. Such a matrix in which only a small fraction of the entries are nonzero is called *sparse*.

We can use a multilist representation to store sparse matrices. The idea is to create $2n$ linked lists, one for each row and one for each column. Each entry of each list stores five things, its row and column index, its numeric value, and links to the next items in the current row and

current column (see Fig. 7). We will not discuss the technical details, but all the standard matrix operations (such as matrix multiplication, vector-matrix multiplication, transposition) can be performed efficiently using this representation.

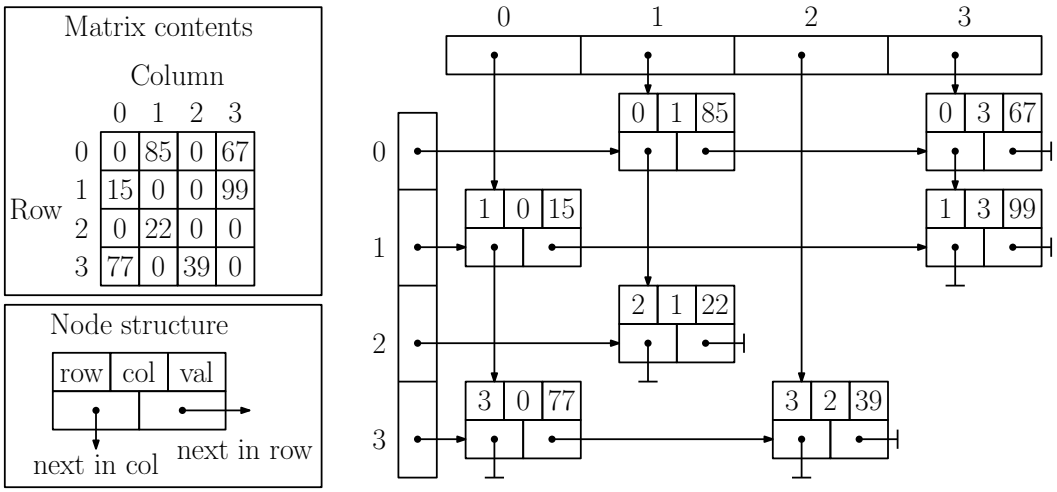



Fig. 7: Sparse matrix representation using a multilist structure.

As a challenge, you might think of how you would write a program to perform matrix multiplication using such a representation. The running time should be "sensitive" to the number of entries in the two matrices.

## Lecture 3: Trees

**Tree Definition and Notation:** Trees and their variants are among the most fundamental data structures. A tree is a special class of graph. Recall from your previous courses that a *graph* $G = (V, E)$ consists of a finite set of *vertices* (or *nodes*) $V$ and a finite set of edges $E$. Each *edge* is a pair of nodes. In an *undirected graph* (or simply "graph"), the edge pairs are unordered, and in a *directed graph* (or "digraph"), the edge pairs are ordered. An undirected graph is *connected* if there is path between any pair of nodes.

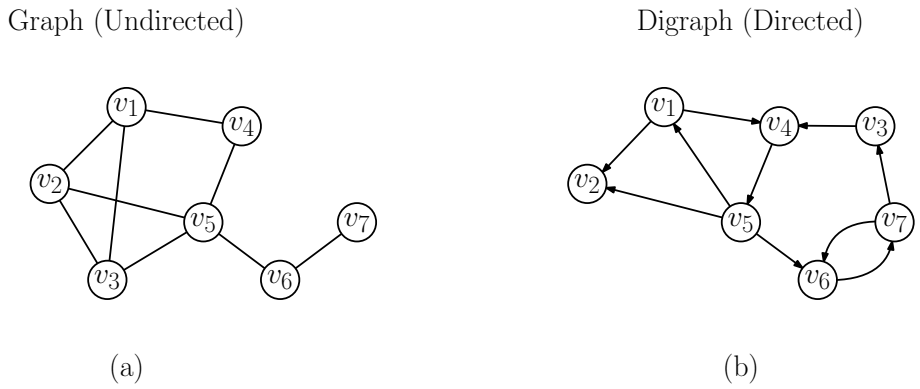



Fig. 8: Graphs: (a) undirected (b) directed.

The most general form of a tree, called a *free tree*, is simply a connected, undirected graph

that has no cycles (see Fig. 9(a)). An example of a free tree is the minimum cost spanning tree (MST) of a graph.

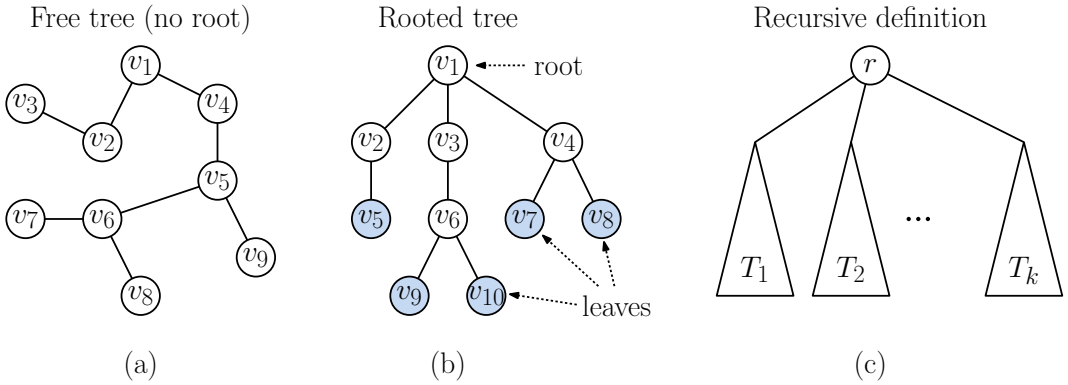



Fig. 9: Trees: (a) free tree, (b) rooted tree, (c) recursive definition.

Since we will want to use trees for applications in searching, it will be useful to assign some sense of order and direction to our trees. This will be done by designating a special node, called the *root*. In the same manner as a family tree, we think of the root as the *ancestor* of all the other nodes in the tree, or equivalently, the other nodes are *descendants* of the root. Nodes that have no descendants are called *leaves* (see Fig. 9(b)). All the others are called *internal nodes*. Each node of the tree can be viewed as the root of a *subtree*, consisting of this node an all of its descendants. Here is a formal (recursive) definition of a rooted tree:

**Rooted tree:** Is either:

(i) A single node, or

(ii) A collection of one more more rooted trees $\{T_1, \ldots, T_k\}$ joined under a common root node (see Fig. 9(c)).

In case (ii), the roots of the subtrees $T_1, \ldots, T_k$ are the *children* of the root node. The *degree* of a node is defined to be its number of children. (Thus, leaves have degree zero.) Note that there is no order among the children (but we will introduce this below with the concept of ordered tress). Since we will be dealing with rooted trees almost exclusively for the rest of the semester, when we say "tree" we will mean "rooted tree." We will use the term "free tree" otherwise.

**Terminology:** There is a lot of notation involving trees. Most terms are easily understood from the family-tree analogy. Every non-root node is descended from its *parent*, and the directed descendants of any node are called its *children* (see Fig. 10(a)). Two nodes that share a common parent are called *siblings*, and other relations (e.g., grandparent, grandchild, first-cousin, second-cousin, etc.) follow analogously.

The *depth* of a node in the tree is the length (number of edges) of the (unique) path from the root to that node. Thus, the root is at depth 0. The *height* of a tree is the maximum depth of any of its nodes (see Fig. 8(b)). For example, the tree of Fig. 9(b) is of depth 3, as evidenced by nodes $v_9$ and $v_{10}$, which are at this depth. As we defined it, there is no special ordering among the children of a node. A rooted tree is *ordered* if there is a total order defined on the children of each node.

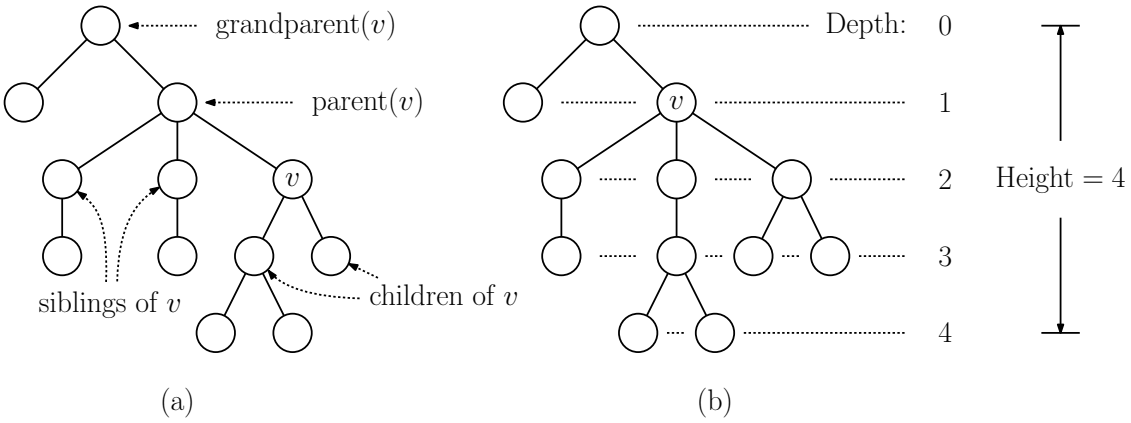



Fig. 10: (a) "Family" relations, (b) depth and height.

**Representing Rooted Trees:** Rooted trees arise in many applications in which hierarchies exist. Examples include computer file systems, hierarchically-based organizations (e.g., military and corporate), documents and reports (volume → chapter → section . . . paragraph). There are a number of ways of representing rooted trees. Later we will discuss specialized representations that are tailored for special classes of trees (e.g., binary search trees), but for now let's consider how to represent a "generic" rooted tree.

The idea is to think of each node of the tree as containing a pointer to the head of a linked list of its children, called its `firstChild`. Since a node may generally have siblings, in addition it will have a pointer to its next sibling, called nextSibling. Finally, each node will contain a *data* element (which we will also refer to as its *entry*), which will depend on the application involved. This is illustrated in Fig. 11(a). Fig. 11(b) illustrates how the tree in Fig. 8(b) would be represented using this technique.

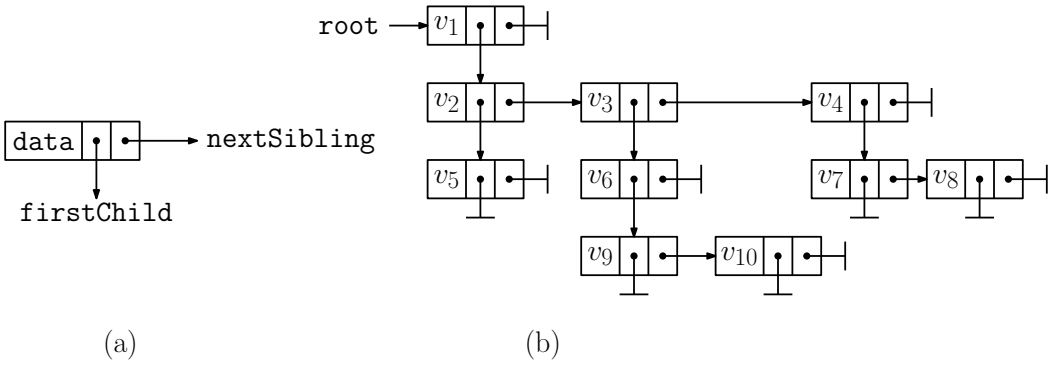
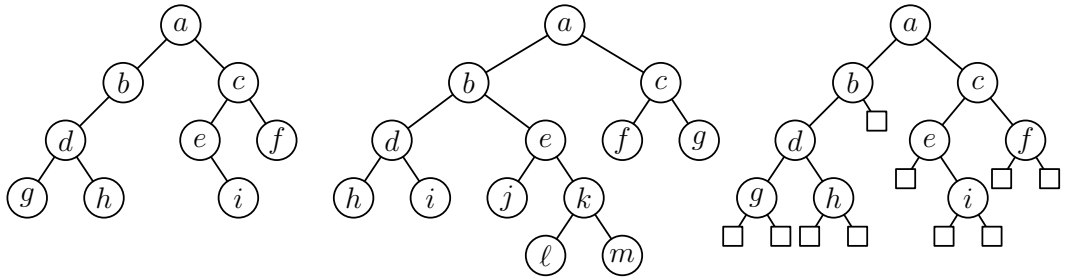



Fig. 11: The "binary" representation of a rooted tree from Fig. 9(b).

Because each node stores two pointers (references), we will often refer to this as the *binary representation* of a rooted tree. Note that this is "minimal" representation. In practice, we may wish to add additional information. (For example, we may also wish to include a reference to each node's parent, the height of a node's subtree, and/or the number of nodes within each node's subtree.)

**Binary Trees:** Among rooted trees, by far the most popular in the context of data structures is the *binary tree*. A *binary tree* is a rooted, ordered tree in which every non-leaf node has two

children, called *left* and *right* (see Fig. 12(a)). We allow for a binary tree to empty. (We will see that, like the empty string, it is convenient to allow for empty trees.)




(a) (b) (c)

Fig. 12: Binary trees: (a) standard definition, (b) full binary tree, (c) extended binary tree.

Binary trees can be defined more formally as follows. First, an empty tree is a binary tree. Second, if $T_L$ and $T_R$ are two binary trees (possibly empty) then the structure formed by making $T_L$ and $T_R$ the left and right children of a node is also a binary tree. $T_L$ and $T_R$ are called the *subtrees* of the root. If both children are empty, then the resulting node is a *leaf*.

Note that, unlike standard rooted trees, there is a difference between a node that has just one child on its left side as opposed to a node that has just one child on its right side. All the definitions from rooted trees (parent, sibling, depth, height) apply as well to binary trees.

Allowing for empty subtrees can make coding tricky. In some cases, we would like to forbid such binary trees. We say that a binary tree is *full* if every node has either zero children (a leaf) or exactly two (an internal node). An example is shown in Fig. 12(b).

Another approach to dealing with empty subtrees is through a process called *extension*. This is most easily understood in the context of the tree shown in Fig. 12(a). We *extend* the tree by adding a special *external node* to replace all the empty subtrees at the bottom of the tree. The result is called a *extended tree*. (In Fig. 12(c) the external nodes are shown has squares.) This has the effect of converting an arbitrary binary tree to a full binary tree.

**Java Representation:** The typical Java representation of a tree as a data structure is given below. The `data` field contains the data for the node and is of some generic entry type `E`. The `left` field is a pointer to the left child (or `null` if this tree is empty) and the `right` field is analogous for the right child.

Binary Tree Node

```
class BTNode<E> {
    E          entry;          // this node's data
    BTNode<E>  left;           // left child
    BTNode<E>  right;          // right child
    // ... remaining details omitted
}
```

As with our rooted-tree representation, this is a minimal representation. Perhaps the most useful augmentation would be a parent link.

Binary trees come up in many applications. One that we will see a lot of this semester is for representing ordered sets of objects, a *binary search tree*. Another is an *expression tree*,

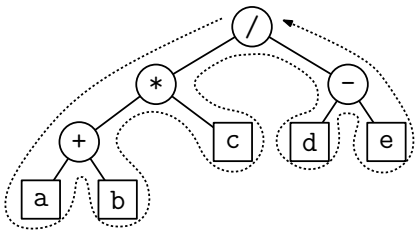
which is used in compiler design in representing a parsed arithmetic exception (see Fig. 13).

**Traversals:** There are a number of natural ways of visiting or *enumerating* every node of a tree. For rooted trees, the three best known are *preorder*, *postorder*, and (for binary trees) *inorder*. Let $T$ be a tree whose root is $r$ and whose subtrees are $T_1, \ldots, T_k$ for $k \geq 0$. They are all most naturally defined recursively. (Fig. 13 illustrates these in the context of an *expression tree*.)

**Preorder:** Visit the root $r$, then recursively do a preorder traversal of $T_1, \ldots, T_k$.

**Postorder:** Recursively do a postorder traversal of $T_1, \ldots, T_k$ and then visit $r$. (Note that this is *not* the same as reversing the preorder traversal.)

**Inorder:** (for binary trees) Do an inorder traversal of $T_L$, visit $r$, do an inorder traversal of $T_R$.




Preorder: `/ * + a b c - d e`

Postorder: `a b + c * d e - /`

Inorder: `a + b * c / d - e`

Fig. 13: Expression tree for $((a + b) * c)/(d - e))$ and common traversals.

These traversals are most easily coded using recursion. The code block below shows a possible way of implementing the preorder traversal in Java. The procedure `visit` would depend on the specific application. The algorithm is quite efficient in that its running time is proportional to the size of the tree. That is, if the tree has $n$ nodes then the running time of these traversal algorithms are all $O(n)$.

Preorder Traversal

```
void preorder(BTNode v)
{
    if (v == null) return;      // empty subtree - do nothing
    visit(v);                   // visit (depends on the application)
    preorder(v.left);           // recursively visit left subtree
    preorder(v.right);          // recursively visit right subtree
}
```

These are not the only ways of traversing a tree. For example, another option would be *breadth-first*, which visits the nodes level by level: "`/ * - + c d e a b`." An interesting question is whether a traversal uniquely determines the tree's shape. The short answer is no, but if you have an extended tree and you know which nodes are internal and which are leaves (as is the case in the expression tree example from Fig. 13), then such a reconstruction is possible. Think about this.

**Extended Binary Trees:** To motivate our next topic, let's consider the utilization of space in our trees. Recall the binary tree shown in Fig. 12(a). It has nine nodes, and each node has two child links, for a total of 18 links. We have eight actual links in the tree, which means

that the remaining 10 links are `null`. Thus, nearly half of the child links are `null`! This is not unusual, as the following theorem states. (Take note the proof structure, since it is common for proofs involving binary trees.)

**Claim:** A binary tree with $n$ nodes has $n+1$ `null` child links.

**Proof:** (by induction on the size of the tree) Let $x(n)$ denote the number of `null` child links in a binary tree of $n$ nodes. We want to show that for all $n \geq 0$, $x(n) = n+1$. We'll make the convention of defnining $x(0) = 1$. (If you like, you can think of the `null` pointer to the root node as this link.)

For the basis case, $n = 0$, by our convention $x(0) = 1$, which satisfies the desired relation.

For the induction step, let's assume that $n \geq 1$. The induction hypothesis states that, for all $n' < n$, $x(n') = n' + 1$. A binary tree with at least one node consists of a root node and two (possibly empty) subtrees, $T_L$ and $T_R$. Let $n_L$ and $n_R$ denote the numbers of nodes in the left and right subtrees, respectively. Together with the root, these must sum to $n$, so we have $n = 1+n_L+n_R$. By the induction hypothesis, the numbers of `null` links in the left and right subtrees are $x(n_L) = n_L + 1$ and $x(n_R) = n_R + 1$. Together, these constitute all the `null` links. Thus, the total number of `null` links is

$$x(n) \;=\; x(n_L) + x(n_R) \;=\; (n_L + 1) + (n_R + 1) \;=\; (1 + n_L + n_R) + 1 \;=\; n + 1,$$

as desired.

In large data structures, these `null` pointers represent quite a significant wastage of space. What can we do about this? One idea is to distinguish two different node types, one for internal nodes, which have (non-`null`) child links and another for leaves, which need no child links. One way to construct such a tree is called "extension." An *extended binary tree* is constructed by replacing each `null` link with a special leaf node, called an *external node*. (The other nodes are called *internal nodes*.) An example of an extended binary tree is shown in Fig. 12(c). Because we replace each `null` link with a leaf node, we have the following direct corollary from our previous theorem.

**Corollary:** An extended binary tree with $n$ internal nodes has $n + 1$ external nodes, and hence $2n + 1$ nodes altogether.

The key "take-away" from this proof is that over half of the nodes in an extended binary tree are leaf nodes. In fact, it is generally true that if the degree of a tree is two or greater, leaves constitute the majority of the nodes.

**Threaded Binary Trees:** We have seen that extended binary trees provide one way to deal with the wasted space caused by `null` pointers in the nodes of a binary tree. In this section we will consider another rather cute use of these pointers.

Recall that binary tree traversals are naturally defined recursively. Therefore a straightforward implementation would require extra space to store the stack for the recursion. Is some way to traverse the tree without this additional storage? The answer is yes, and the trick is to employ each `null` pointer encode some additional information to aid in the traversal. Each left-child `null` pointer stores a reference to the node's inorder predecessor, and each right-child `null` pointer stores a reference to the node's inorder successor. The resulting representation is called a *threaded binary tree*. (For example, in Fig. 14(a), we show a threaded version of the tree in Fig. 12(a)).

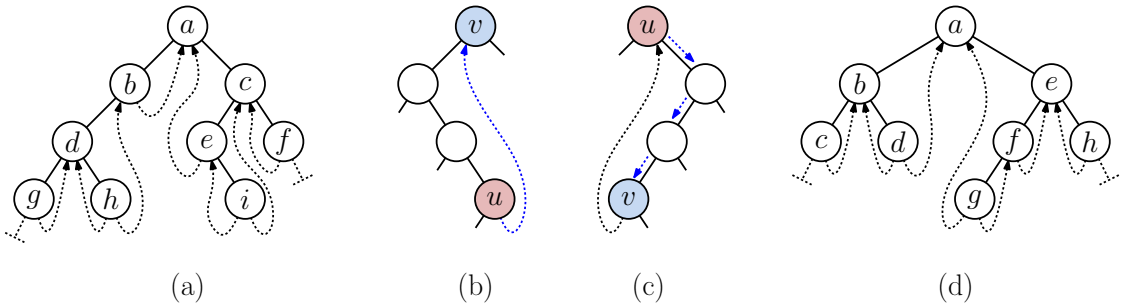



Fig. 14: A Threaded Tree.

We also need to add a special "mark bit" to each child link, which indicates whether the link is a thread or a standard parent-child link. The node structure is shown below.

Threaded Binary Tree Node

```
class ThreadBTNode<E> {
    E                entry;          // this node's data
    Boolean          leftIsThread;   // left child link is a thread
    Boolean          rightIsThread;  // right child link is a thread
    ThreadBTNode<E>  left;           // left child
    ThreadBTNode<E>  right;          // right child
    // ... remaining details omitted
}
```

Let us consider how to do an inorder traversal in a threaded-tree representation. Suppose that we are currently visiting a node $u$. How do we find the inorder successor of $u$? First, if $u$'s right-child link is a thread, then we just follow it (see Fig. 14(b)). Otherwise, we go $u$'s right child, and then traverse left-child links until reaching the bottom of the tree, that is a threaded link (see Fig. 14(c)).

For example, in Fig. 14(d), if we start at $d$, the thread takes us directly to $a$, which is $d$'s inorder successor. If we continue from $a$, we follow the right-child link to $e$, and then follow left-links until arriving at $g$, which is $a$'s inorder successor. Of course, to turn this into a complete traversal function, we would need to start by finding the first inorder node in the tree. We'll leave this as an exercise.

Inorder Successor in a Threaded Tree

```
ThreadBTNode inorderSuccessor(ThreadBTNode v) {
    ThreadBTNode u = v.right;           // go to right child
    if (v.rightIsThread) return u;      // if thread, then done
    while (!u.leftIsThread) {           // else u is right child
        u = u.left;                     // go to left child
    }                                   // ...until hitting thread
    return u;
}
```

Threading is more of a "cute trick" than a common implementation technique with binary trees. Nonetheless, it is representative of the number of clever ideas that have been developed over the years for representing and processing binary trees.

**Complete Binary Trees:** We have discussed linked allocation strategies for rooted and binary trees. Is it possible to allocate trees using sequential (that is, array) allocation? In general it is not possible because of the somewhat unpredictable structure of trees (unless you are willing to waste a lot of space). However, there is a very important case where sequential allocation is possible.

**Complete Binary Tree:** Every level of the tree is completely filled, except possibly the bottom level, which is filled from left to right.

It is easy to verify that a complete binary tree of height $h$ has between $2^h$ and $2^{h+1} - 1$ nodes, implying that a tree with $n$ nodes has height $O(\log n)$ (see Fig. 15). (We leave these as exercises involving geometric series.)

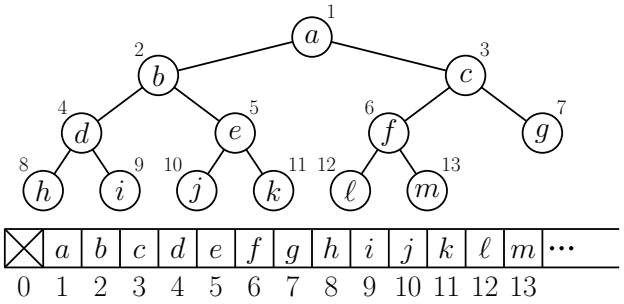



Fig. 15: A complete binary tree.

The extreme regularity of complete binary trees allows them to be stored in arrays, so no additional space is wasted on pointers. Consider an indexing of nodes of a complete tree from 1 to $n$ in increasing level order (so that the root is numbered 1 and the last leaf is numbered $n$). Observe that there is a simple mathematical relationship between the index of a node and the indices of its children and parents. In particular:

**leftChild**$(i)$**:** if $(2i \leq n)$ then $2i$, else `null`.

**rightChild**$(i)$**:** if $(2i + 1 \leq n)$ then $2i + 1$, else `null`.

**parent**$(i)$**:** if $(i \geq 2)$ then $\lfloor i/2 \rfloor$, else `null`.

As an exercise, see if you can give a formula to compute the sibling of node $i$ and the depth of node $i$.

Observe that the last leaf in the tree is at position $n$, so adding a new leaf simply means inserting a value at position $n+1$ in the list and updating $n$. Since arrays in Java are indexed from 0, omitting the 0th entry of the matrix is a bit of wastage. Of course, the above rules can be adapted to work even if we start indexing at zero, but they are not quite so simple.

## Lecture 4: Disjoint Set Union-Find

**Equivalence relations:** An *equivalence relation* over some set $S$ is a relation that satisfies the following properties for all elements of $a, b, c \in S$.

**Reflexive:** $a \equiv a$.

**Symmetric:** $a \equiv b$ then $b \equiv a$

**Transitive:** $a \equiv b$ and $b \equiv c$ then $a \equiv c$.

Equivalence relations arise in numerous applications. An example includes any sort of "grouping" operation, where every object belongs to some group (perhaps in a group by itself) and no object belongs to more than one group. More formally these groups are called *equivalent classes* and the subdivision of the set into such classes is called a *partition*. For example, suppose we are maintaining a bidirectional communication network. The ability to communicate is an equivalence relation, since if machine $a$ can communicate with machine $b$, and machine $b$ can communicate with machine $c$, then machine $a$ can communicate with machine $c$ (e.g. by sending messages through $b$). Now suppose that a new link is created between two groups, which previously were unable to communicate. This has the effect of *merging* two equivalence classes into one class.

We discuss a data structure that can be used for maintaining equivalence partitions with two operations: (1) *union*, merging to groups together, and (2) *find*, determining which group an element belongs to. This data structure should *not* be thought of as a general purpose data structure for storing sets. In particular, it cannot perform many important set operations, such as splitting two sets, or computing set operations such as intersection and complementation. And its structure is tailored to handle just these two operations. However, there are many applications for which this structure is useful. As we shall see, the data structure is simple and amazingly efficient.

**Union-Find ADT:** We assume that we have an underlying finite set of elements $S$. We want to maintain a *partition* of the set. In addition to the constructor, the (abstract) data structure supports the following operations.

`Set s = find(Element x)`: Return an *set identifier* of the set $s$ that contains the element $x$. A set identifier is simply a special value (of unspecified type) with the property that `find(x) == find(y)` if and only if $x$ and $y$ are in the same set.

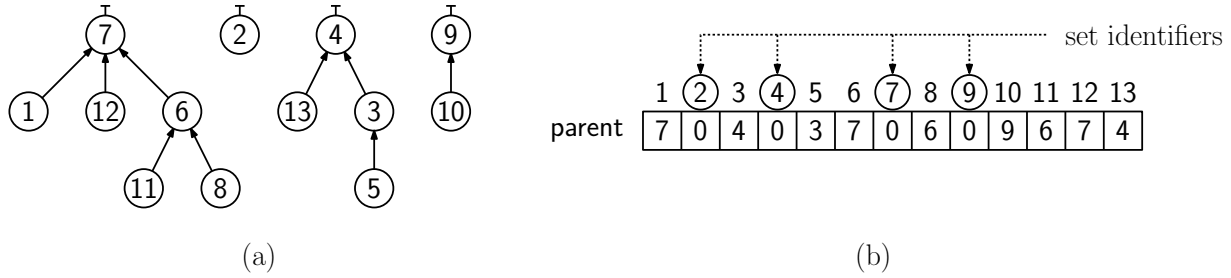
`Set r = union(Set s, Set t)`: Merge two sets named $s$ and $t$ into a single set $r$ containing their union. We assume that $s$, $t$ and $r$ are given as set identifiers. This operations destroys the sets $s$ and $t$.

Note that there are *no key values* used here. The arguments to the find and union operations are pointers to objects stored in the data structure. The constructor for the data structure is given the elements in the set $S$ and produces a structure in which every element $x \in S$ is in a singleton set $\{x\}$ containing just $x$.

**Inverted-Tree Implementation:** We will derive our implementation of a data structure for the union-find ADT by starting with a simple structure based on a forest of *inverted trees*. You think of an inverted tree as a multiway tree in which we only store parent links (no child or sibling links). A root's parent pointer is `null`. There is no limit on how many children a node can have. The sets are represented by storing the elements of each set in separate tree. For example, suppose that $S = \{1, 2, 3, \ldots, 13\}$ and the current partition is:

$$\{1, 6, 7, 8, 11, 12\}, \{2\}, \{3, 4, 5, 13\}, \{9, 10\}.$$

This might be stored in an inverted tree as shown in Fig. 16(a). Note that there is no particular order to how the individual trees are structured, as long as they contain the proper elements.




(a) (b)

Fig. 16: (a) Partition stored as a forest of inverted trees and (b) array-based representation.

**Array representation:** You might wonder, "How do I find the node containing a given element?" Let's assume that the set elements are integers $\{1, 2, \ldots, n\}$ as shown above. Rather than using a standard node and pointer structure, we will think of the elements as being entries in an array `parent[1..n]`. We define `parent[i]` to hold the index the parent of element `i` in our tree or zero if `i` is a root of a tree. (We will sometimes call this zero and sometimes call this `null`, but they mean the same thing.) Even though we will use the tree representations in our illustrations, the actual implementation of the data structure is the array.

Initially, every element is in its own set, which we can set up by setting every element of the `parent` array to zero.

**Set identifiers:** Using this representation, each `Element` of the set is just an integer `x`, where $1 \leq \texttt{x} \leq n$. Each set is represented by a root in an inverted tree. A *set identifier*, called `Set`, is just an integer index `x`, where $1 \leq \texttt{x} \leq n$ and `parent[i] == 0`. For example, in Fig. 16(a) the set $\{3, 4, 5, 13\}$ is represented by the set identifier (root) `4`.

If we want to know whether two elements are in the same set, we simply find the roots of their associated trees (by following parent links) and then check whether these two roots are the same. For example, when we trace the parent links from `12` and `8` both stop at `7`, implying that `12` and `8` are in the same set. On the other hand, when we trace the parent links from `6` and `10`, they stop at the roots `7` and `9`, respectively. Since these are different roots, these elements are in different sets.

**Find Operation:** As mentioned above, given any element `x`, we perform the operation `find(x)` by walking along parent links until reaching the root of its tree. We return the root of the tree as the desired set identifier. This root element is set identifier for the set containing $x$. Notice that this satisfies the above requirement, since two elements are in the same set if and only if they have the same root. We call this `simple-find()`. (Later we will propose an improvement.)

Find operation (without path compression)

```
Set simple-find(Element x) {
    while (parent[x] != null) x = parent[x]  // follow chain to root
    return x;                                // return the root
}
```

For example, in Fig. 16(b)), operation `simple-find(11)` would start at node `11` and trace the path up through `6` up to the root `7`. It returns the index `7` as the answer.

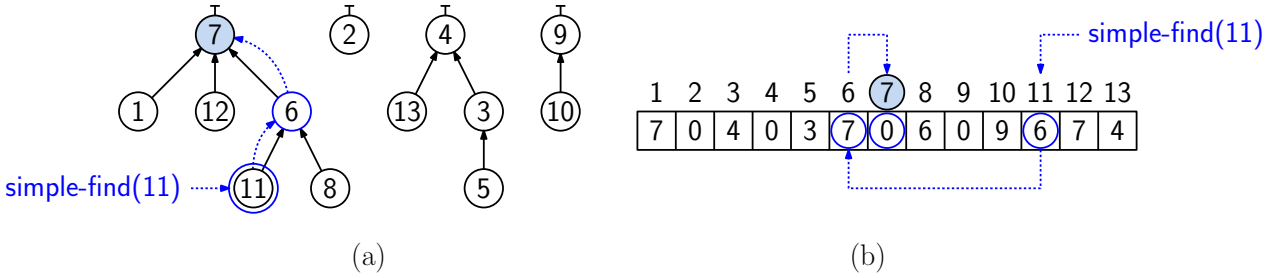



Fig. 17: The operation `simple-find(11)`.

**Union Operation:** A union is similarly straightforward to implement. To perform the union of two sets we just link the root of one tree into the root of the other tree. But here is where it pays to be smart. Recall that *height* of a tree is the maximum number of edges from any leaf to the root. In Fig. 18, we label each tree with its height. If we link the root of `i` as a child of `b`, the height of the resulting tree will be 2, whereas if we do it the other way the height of the tree will only be 1. Clearly, it is better to link the lower height tree under the other to keep the final tree's height as small as possible. This will make future `find` operations run faster. In contrast, if we take the union of two trees of equal height, say `g` and `d`, we can make either the root. (You might wonder why we use rank rather than some other property, and this is a good question for further thought.)

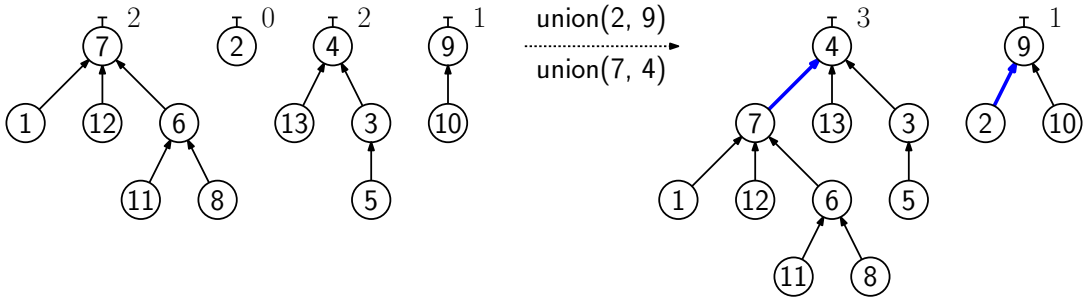



Fig. 18: Union-find with ranks.

In our later enhancements, we will compress paths, and this will change the tree height. So, we will invent an abstract concept, called *rank*, which behaves similar to tree height. We assume that the rank is stored as a field in each node, but we only use it for root nodes. The operation `union(s, t)` is given the roots of two trees. It swaps them if necessary so that `t`'s rank is at least as large as `s`'s. It then links `s` as a child of `t` and update's `t`'s rank to be the maximum of `t`'s rank and 1 plus `s`'s rank. The implementation is shown in the following code block.

**Storing Ranks:** There is a cool trick for storing ranks. As suggested in the code block, we could just implement a separate array. But since ranks are only needed for tree roots, we can do all of this with just a single array. Rather than setting `parent[x] = 0` to indicate a root, we can set `parent[x]` to be the negation of the rank of `x`. When searching for the root of a tree, rather than testing whether `parent[x] == null` (or equivalently zero), we can check whether `parent[x] < 0`. If so, then the rank can be extracted by negating its value.

**Analysis of Running Time:** Consider a sequence of $m$ union-find operations, performed on a

Union operation

```
Set union(Set s, Set t) {
    if (rank[s] > rank[t]) {            // s has higher rank?
        swap s and t                    // swap so that t's rank is larger
    }
    parent[s] = t                       // link s as child of t
    rank[t] = max(rank[t], 1 + rank[s]) // update t's rank
    return t
}
```

domain with $n$ total elements. Observe that the running time of the initialization is proportional to $n$, the number of elements in the set, but this is done only once. Each union takes only constant time, $O(1)$.

In the worst case, find takes time proportional to the height of the tree. The key to the efficiency of this procedure is the following observation, which implies that the tree height is never greater than $\lg m$. (Recall that $\lg m$ denotes the logarithm base 2 of $m$.)

**Lemma:** Using the union-find procedures described above any tree with height $h$ has at least $2^h$ elements.

**Proof:** Given a union-find tree $T$, let $h$ denote the height of $T$, let $n$ denote the number of elements in $T$. We will show that $n \geq 2^h$.

It will help to introduce an intermediate quantity to help drive the proof. Let $u \geq 0$ denote the number of union operations used to build $T$. Our proof is based on induction on $u$. For the basis (no unions, or $u = 0$) we have a tree with $n = 1$ element of height $h = 0$. Since $1 = 2^0$, we have $n \geq 2^h$, which establishes the basis case.

For the induction step, suppose that we form a tree $T$ through $u$ union operations by merging two trees $T'$ and $T''$. Let $n'$ and $n''$ be their respective numbers of elements, let $h'$ and $h''$ be their respective heights, and let $u'$ and $u''$ denote the number of union operations to build each. The number unions to build $T$ is clearly $u = 1 + u' + u''$. Thus, $u'$ and $u''$ are both smaller than $u$, which means that we can apply the induction hypothesis to each of them, implying that $n' \geq 2^{h'}$ and $n'' \geq 2^{h''}$.

Following the merge we have a total $n = n' + n''$ elements. The final height depends on $h'$ and $h''$. We may assume without loss of generality that $h' \leq h''$. (If not, swap the trees $T'$ and $T''$.)

There are two possibilities. If $h' = h''$, then the resulting tree has height $h = h' + 1 = h'' + 1$ (see Fig. 19(a)). The number of elements in the final tree is

$$n \;=\; n' + n'' \;\geq\; 2^{h'} + 2^{h''} \;=\; 2^{h-1} + 2^{h-1} \;=\; 2 \cdot 2^{h-1} \;=\; 2^h.$$

On the other hand, if $h' < h''$, then the final tree has height equal to the larger tree, that is, $h = h''$ (see Fig. 19(b)). The number of elements is

$$n \;=\; n' + n'' \;\geq\; n'' \;\geq\; 2^{h''} \;=\; 2^h.$$

In either case we obtain the desired conclusion.

Since the unions's take $O(1)$ time each, we immediately have the following.

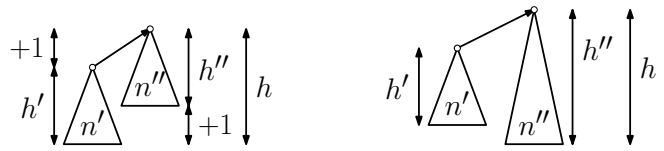



Fig. 19: Proof that $n \geq 2^h$.

**Theorem:** After initialization, any sequence of $m$ union's and find's can be performed in time $O(m \log m)$. In other words, the amortized time for union-find operations is $O(\log m)$.

**Path Compression:** It is possible to apply a very simple heuristic improvement to this data structure which provides a significant improvement in the running time. Here is the intuition. If the user of your data structure repeatedly performs find's on a leaf at a very low level in the tree then each such find takes as much as $O(\log n)$ time. Can we improve on this?

Once we know the result of the find, we can go back and "short-cut" each pointer along the path to point directly to the root of the tree. This only increases the time needed to perform the find by a constant factor, but any subsequent find on this node (or any of its ancestors) will take only $O(1)$ time. The operation of short-cutting the pointers so they all point directly to the root is called *path-compression* and an example is shown below. Notice that only the pointers along the path to the root are altered. We present a slick recursive version below as well. Trace it to see how it works.

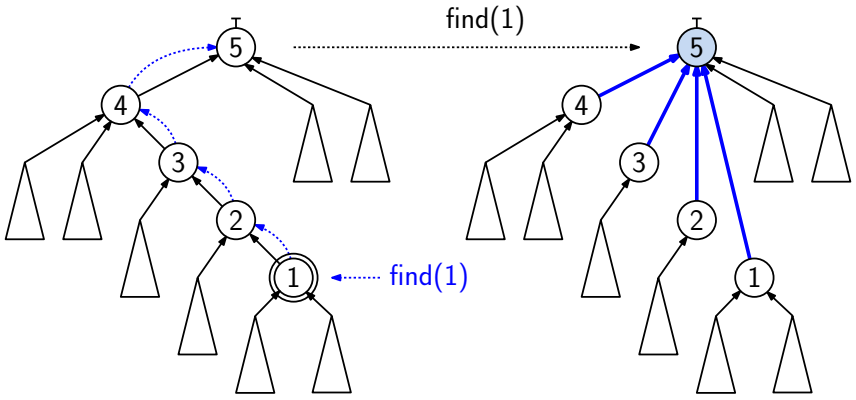



Fig. 20: Find using path compression.

Find operation (with path compression)

```
Set find-compress(Element x) {
    if (parent[x] == null) return x              // return root
    else {
        mySet = find-compress(parent[x])         // find
        parent[x] = mySet                        // compress the link
        return mySet
    }
}
```

The running time of `find-compress` is still proportional to the depth of node being found, but observe that each time you spend a lot of time in a find, you flatten the search path. Thus the work you do provides a benefit for later find operations. (This is the sort of thing that we observed in earlier amortized analyses.)

Does the savings really amount to anything? The answer is yes, but it is not easy to analyze. In 1975, Robert Tarjan proved that the amortized running time is strictly more than $O(1)$, it is much less than $O(\log m)$.

Analyzing this algorithm is quite tricky. In order to create a bad situation you need to do lots of unions to build up a tree with some real depth. But as soon as you start doing finds on this tree, it very quickly becomes very flat again. In the worst case we need to an immense number of unions to get high costs for the finds.

To give the analysis (which we won't prove) we introduce two new functions, $A(i, j)$ and $\alpha(i)$. For $i, j \geq 0$, the function $A(i, j)$ is called *Ackerman's function* (discovered by Wilhelm Ackerman way back in 1928).

$$A(i,j) \;=\; \begin{cases} j+1 & \text{if } i = 0, \\ A(i-1,1) & \text{if } i > 0 \text{ and } j = 0, \\ A(i-1, A(i,j-1)) & \text{otherwise.} \end{cases}$$

It is famous for being just about the fastest growing function imaginable. It is not obvious from the definition, so here are a few examples as the value of $i$ increases

$$\begin{aligned} A(0,j) &= j+1 \\ A(1,j) &= j+2 \\ A(2,j) &= 2j+3 \\ A(3,j) &= 2^{j+3} - 3 \\ A(4,j) &= \left(2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right) - 3, \end{aligned}$$

where the tower of 2's in $A(4, j)$ is of height $j + 3$. As the value of $i$ increases, the function increases to insanely large values very quickly. Even modest values, such as $A(4, 5) \approx 2^{65,536}$, which is already much larger than the number of particles in the observable universe.

Ackerman's function grows unbelievably fast. To create correspondingly slow growing function, we will defined its inverse, which we call $\alpha$. Define

$$\alpha(m,n) \;=\; \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \lg n\}.$$

This definition is somewhat hard to interpret, but the important bottom line is that assuming $\lfloor m/n \rfloor \geq 1$, we have $\alpha(m, n) \leq 4$ as long as $m$ is less than the number of particles in the universe, which is certainly true for any input set your program will ever encounter! The following result shows that an sequence of union-find operations take amortized time $O(\alpha(m, n))$. While we cannot formally state that this is constant time, it is constant time for all practical purposes. (The proof is quite complicated, and we will not present the proof.)

**Theorem:** After initialization, any sequence of $m$ union's and find's (using path compression) on an initial set of $n$ elements can be performed in time $O(m \cdot \alpha(m, n))$ time. Thus the amortized cost of each union-find operation is $O(\alpha(m, n))$.

## Lecture 5: Priority Queues and Heaps

**Priority Queues:** A *priority queue* is an abstract data structure storing key-value pairs. The basic operations involve inserting a new key-value pair (where the key represents the *priority*) and extracting the entry with the smallest priority value. These operations are called *insert* and *extract-min*, respectively. In contrast to a standard queue (first-in, first-out) a priority queue extracts entries according to their priority. As an example, irrespective of the order in which passengers arrive at the gate, airlines often board them according to row number from the rear of the plane.

Priority queues can be implemented in many different ways. For example, you could maintain a simple linear list of key-priority pairs. But how are these to be sorted? If you sort by arrival order, then insertion is fast, but extraction requires checking all the priorities, which takes $O(n)$ time. On the other hand, if you sort by priority, extraction is fast, but insertion involves determining where to put the new item, and this (naively) takes $O(n)$ time.

The question is whether it is possible to achieve both operations in $O(\log n)$ time. There is a collection of related tree-based data structures that support these times. Because they share the same general structure, they are all called *heaps*.

**Heaps:** At its most generic, a *heap* is a rooted (typically binary) tree where each node stores a key-value pair. These data structures all have one common aspect, other than the root node, the priority of every node is greater than or equal to its parent (see Fig. 21). A tree that satisfies this property is said to be in *heap order*. (This is sometimes called a *min heap*. Reversing the order results in a *max heap*. The max heap is usually used in HeapSort.)

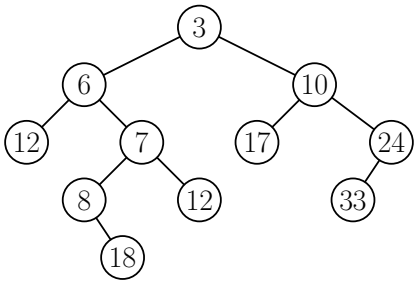



Fig. 21: A (min) heap-ordered binary tree. (Only priorities are shown.)

Note that in such a tree, the smallest priority item is always at the root. They main questions are how to maintain this structure as elements are inserted and extracted.

There are many variants of the heap data structure. If you learned about the sorting algorithm called *HeapSort*, then you no doubt learned the the simplest of these structures, called the *binary heap* (which we will present below). But when additional operations are desired (for example, altering individual priority values or merging two heaps together), there are alternative data structures that are more efficient than the binary heap. This has given rise to data structures with various esoteric names such as *binomial heaps*, *Fibonacci heaps*, *pairing heaps*, *quake heaps*, *leftist heaps*, and *skew heaps*. (For further information, see this Wikipedia article on Heaps.) In this lecture, we will discuss just a couple of these, standard binary heaps and leftist heaps.

**Binary Heaps:** The data structure used in HeapSort is called a *binary heap*. It is a venerable data structure, invented by J. W. J. Williams way back in 1964. It has a number of very

elegant features. Most notably, even though it is a binary tree, it can be stored in an array, without the needs for dynamic memory management or pointers.

Recall from our lecture on trees that a binary tree is *complete* (sometimes called *left-complete*) if every level of the tree is completely filled, except possibly the bottom level, which is filled from left to right. It is easy to verify that a complete binary tree of height $h$ has between $2^h$ and $2^{h+1}-1$ nodes, implying that a tree with $n$ nodes has height $O(\log n)$ (see Fig. 22). (We leave these as exercises involving geometric series.)

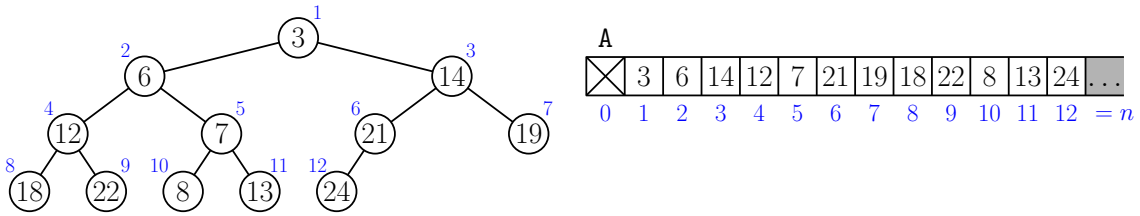



Fig. 22: A heap-ordered complete binary tree and its mapping to an array.

The regular structure allows us to use arithmetic to identify tree relations. Given a complete tree with $n$ elements stored in an array `A[1..n]`, and for any index $1 \le i \le n$, we can access its immediate relations:

- left($i$): if ($2i \le n$) then $2i$, else `null`
- right($i$): if ($2i+1 \le n$) then $2i+1$, else `null`
- parent($i$): if ($i \ge 2$) then $\lfloor i/2 \rfloor$, else `null`

Note that we are effectively wasting element `A[0]` by using this scheme. It is possible to modify the indexing rules to start with index zero, but this makes the access formulas a bit more complicated.

**Binary Heap Insertion:** Let us assume that our binary heap currently contains $n$ elements and is stored in the array `A[1..n]`. (To initialize the structure, we simply set $n \leftarrow 0$.) To insert a new key `x` into the binary heap, we increment $n$ and add the new item at the end of the array. We then "sift" the new key up the tree by swapping it with its parent as long as its priority is smaller than its parent, or until hitting the root, whichever comes first (see Fig. 23).

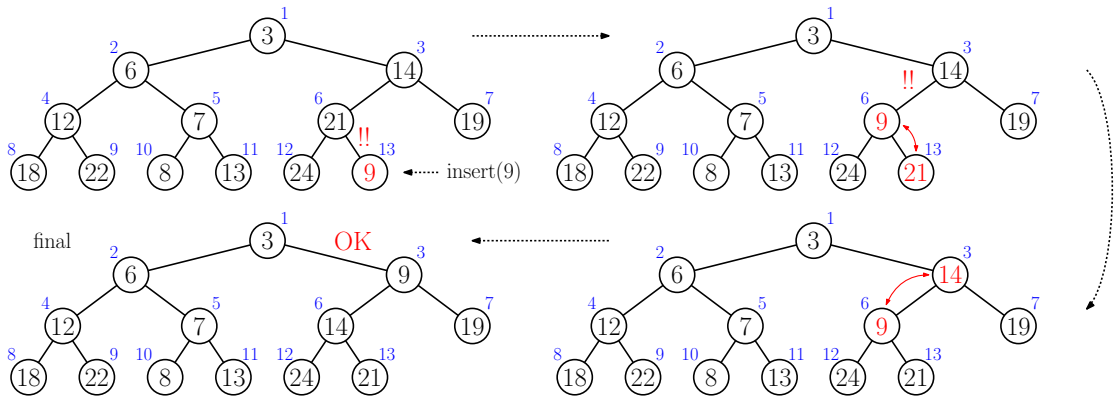



Fig. 23: Example of inserting a new element (9) into a binary heap and sifting up.

The insertion code is presented in the following code block. The code is slightly different from our description. To make the code a bit faster, rather than storing the new key and swapping, we copy entries down and insert the new key at the end. Clearly, the running time is proportional to the height of the tree, which we have shown is $O(\log n)$.

Binary-Heap Insertion

```
void insert(Key x) {                    // insert new key x in heap
    n += 1                              // increment tree size
    i = sift-up(n, x)                   // sift x up the tree to its final position
    A[i] = x                            // store x in the heap
}

int sift-up(int i, Key x) {             // sift x up to its proper position
    while(i > 1 && x < A[parent(i)]) {  // x's parent exists and is too large?
        A[i] = A[parent(i)]             // move parent down (and x up)
        i = parent(i)
    }
    return i
}
```

**Binary Heap Extract-Min:** To perform extract-min, we already observed that the minimum element is at the root. But, if we remove this element, we have a hole that needs to be filled. There is an elegant method to fill this hole. We first save the root element as our final result. We then copy the $n$th element of the array to index 1 and decrement the value of $n$. Finally, we sift the new root element down the tree to restore the heap property. To sift an element down, we first determine which of its two children has the smaller priority. We compare the current node with this smaller child. If the child has a smaller priority than the current node, then we swap them. We repeat this along the we have a child and the swap takes place. The process is illustrated in Fig. 24) and the code is presented just below.

**Mergeable Heaps:** The standard binary heap data structure is an simple and efficient data structure for the basic priority queue operations insert and extract-min. Suppose that we have an application in which in addition to insert and extract-min, we want to be able to *merge* the contents of two different queues into one queue. As an application, suppose that we have a two-processor computer system and each processor has a priority queue of processes waiting to be executed. If one of the processors fails, we need to merge the two queues so that the remaining processor can handle all of them.

Let's introduce a new operation `H = merge(H1, H2)`, which takes two existing priority queues $H_1$ and $H_2$, and merges them into a new priority queue, $H$. This operation is *destructive*, which means that the priority queues $H_1$ and $H_2$ are destroyed in order to form $H$.

We would like to be able to implement `merge` in $O(\log n)$ time, where $n$ is the total number of keys in priority queues $H_1$ and $H_2$. Unfortunately, it does not seem to be possible to do this with the standard binary heap data structure because of its highly rigid array-based structure.

**Leftist Heaps:** We introduce a new data structure called a *leftist heap*, which supports the operations `insert(x)`, `extract-min()`, and `merge(H1,H2)`. Like the binary heap, it is a binary tree, but it is stored as a standard binary tree with left and right child pointers. In order

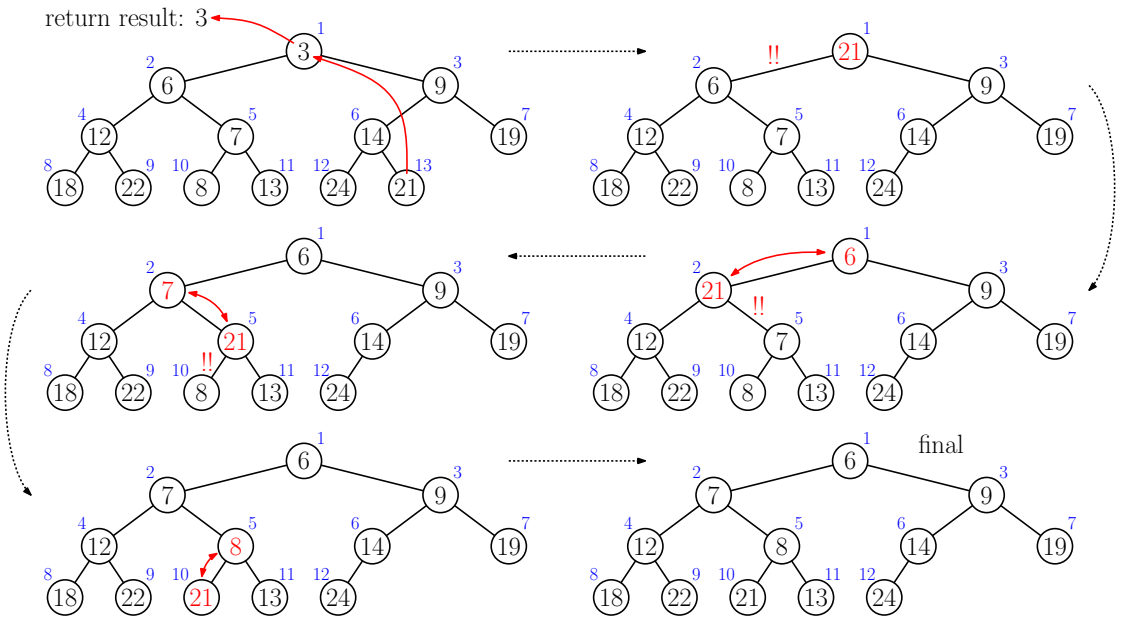



Fig. 24: Example of extract-min. We save the root element (3) as the final result, and copy the last element (21) into the root and decrement $n$. We then repeatedly sift it down by swapping it with the smaller of its two children until either reaching the leaf level or until both of its children are at least as large.

Binary-Heap Extract-Min

```
Key extract-min() {                         // extract the minimum from heap
    if (n == 0) Error - Empty heap
    Key result = A[1]                       // save final result
    Key z = A[n--]                          // get element to sift (and decrement n)
    i = sift-down(1, z)                     // sift z from root to its final position
    A[i] = z                                // move z into its proper position
    return result
}

int sift-down(int i, Key z) {               // sift z down to its proper position
    while (left(i) != null) {               // repeat until arriving at a leaf
        u = left(i);  v = right(i)          // i's left and right children
        if (v != null && A[v] < A[u])       // swap so that u has smaller key
            u = v
        if (A[u] < z) {                     // sift z by moving A[u] up
            A[i] = A[u]
            i = u
        } else break                        // done sifting
    }
    return i                                // return z's proper position
}
```

to support operations in $O(\log n)$ time, we want the tree's height to be $O(\log n)$. Our data structure will have a weaker property, which is where the term "leftist" comes from.

**Leftist Heap Property:** Define the *null path length*, denoted `npl(v)`, of any node $v$ to be the length of the shortest path to reach a node with a `null` child pointer. The value of `npl(v)` can be defined recursively as follows (see Fig. 25).

$$\texttt{npl(v)} = \begin{cases} \texttt{-1} & \text{if } \texttt{v} = \texttt{null}, \\ \texttt{1 + min(npl(v.left), npl(v.right))} & \text{otherwise.} \end{cases}$$

Note that the npl value of any (non-null) node is at least zero. The npl of a node is very different from its height. Trees with very large heights can have very small npl value.

**Leftist:** A node `v` is *leftist* if `npl(v.left)` $\geq$ `npl(v.right)`.

**Leftist heap:** Is a binary tree whose keys are heap ordered (parent key is less than or equal to child key) and whose nodes' npl values all satisfy the leftist property.

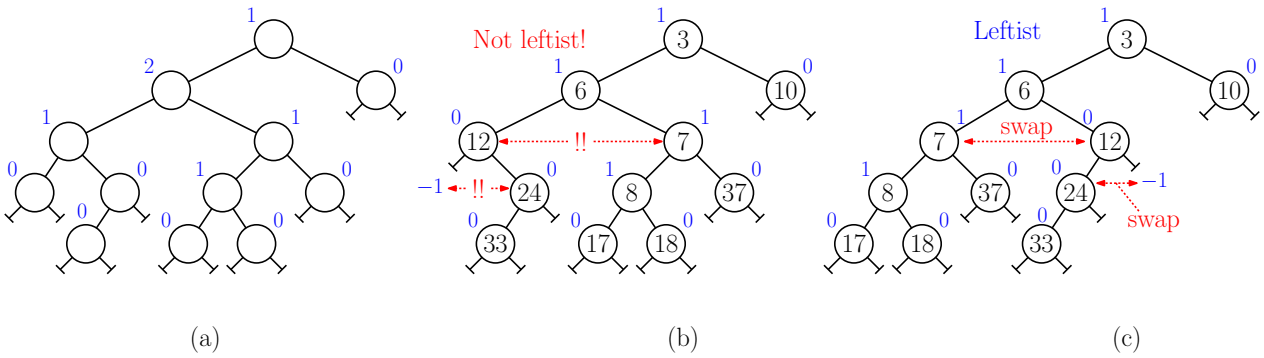



Fig. 25: Null path lengths and the leftist property.

For example, the two trees shown in Fig. 25 are both heap-ordered, but the tree in part (b) is not leftist because node 6's left child has a smaller npl value than its right child. Swapping these two children as shown in (c) yields a valid leftist heap.

Note that any tree that does not satisfy leftist property can always be made to do so by swapping left and right subtrees at any nodes that violate the leftist property. Observe that this does not affect the heap-order property. The key to the efficiency of leftist heap operations is that there exists a short ($O(\log n)$ length) path in every leftist heap, namely the rightmost path. Let $n$ denote the number of nodes in the heap and let $r$ denote the number of nodes on the rightmost path. We want to show that $r = O(\log n)$. More precisely, we can show that $r \leq \lg(n+1)$. By exponentiating both side, this is clearly equivalent to showing that $n \geq 2^r - 1$. The next lemma shows this.

**Lemma:** Consider any leftist heap with $n \geq 1$ nodes and $r$ nodes along its rightmost path. Then $n \geq 2^r - 1$.

**Proof:** The proof is by induction on the number of nodes in the tree.

**Basis:** ($n = 1$) If there is only one node in the tree, then there is only one node on the rightmost path. Thus, $r = 1$. We therefore have, $n = 1 = 2^1 - 1 = 2^r - 1$, as desired.

**Step:** Let $n \geq 2$ denote the number of nodes in the tree, and let us make the (strong) induction hypothesis that any tree with strictly fewer than $n$ nodes satisfies the lemma.

Remove the root of the tree. This results in subtrees. Let $n_L$ be the number of nodes in the left subtree and let $n_R$ denote the number of nodes in the right subtree. Both subtrees have fewer than $n$ nodes, and therefore we can apply the induction hypothesis to them.

Let $r$ denote the number of nodes in the rightmost path of the original tree. It follows that the right subtree has $r-1$ nodes in its rightmost path, and thus by the induction hypothesis, we have $n_R \geq 2^{r-1}-1$. We cannot infer exactly how many nodes are in the rightmost path of the right subtree, but by the leftist property, it must be at least $r-1$. (If it were smaller, then the NPL of the left subtree would be smaller than the NPL of the right subtree, which would violate the leftist property.) Therefore, we have $n_L \geq 2^{r-1}-1$.

Putting this all together, we have

$$n \;=\; 1+n_L+n_R \;\geq\; 1+(2^{r-1}-1)+(2^{r-1}-1) \;=\; 2\cdot 2^{r-1}-2+1 \;=\; 2^r-1,$$

as desired.

**Corollary:** The rightmost path of a leftist binary tree with $n$ nodes has length $O(\log n)$.

**Class Structure:** Before discussing how merging is performed in leftist heaps, it would be good to give an overview of the class structure (see the code block below). The class `LeftistHeap` is generic, and its declaration is templated by the key type `Key`. Because we need to compare keys, we inform the compiler that the key type must implement the `Comparable` interface.

The class consists of three major components, a class declaration for the node type, the private data, and the various public and private methods. The node, called `LHNode` stores the key, the child pointers, and the npl value for this node. `LeftistHeap` has one piece of private data, namely the root of the tree, and the class constructor just sets the root to `null`, thus generating an empty tree. This is followed by the other public and private members. These include the public methods `insert`, `extractMin`, and `mergeWith`. We have omitted all the details, including addition private helper methods.

Leftist Heap Class Structure

```
public class LeftistHeap<Key extends Comparable<Key>> {

    private class LHNode {                  // a node in the tree
        Key key;                            // key
        LHNode left, right;                 // children
        int npl;                            // null path length
    }

    private LHNode root;                    // root of the tree

    public LeftistHeap() { root = null; }   // constructor
    public void insert(Key x) { ... }       // insert
    public Key extractMin() { ... }         // extract-min
    public void mergeWith(LeftistHeap<Key> H2) { ... } // merge with H2
}
```

The public merge method that performs the merger is called `mergeWith`, and it merges this heap with another heap `H2`. It invokes a recursive helper function (given below) and updates the root to point to the resulting tree. Note that `H2` is destroyed in the process.

```
public void mergeWith(LeftistHeap H2) {      // merge this heap with H2
    root = merge(this.root, H2.root)         // invoke the helper and update root
    H2.root = null                           // H2 is destroyed in the process
}
```

**Merging Leftist Heaps:** All that remains, is to show how to implement the helper function `merge(u, v)`. This takes two nodes `u` and `v`, one from each of the heaps, merges the subtrees rooted at these two nodes, and returns a pointer to the root node of the merged tree.

The formal description of the procedure is recursive. However it is somewhat easier to understand in its nonrecursive form, which operates in two separate phases. In the first phase, we walk down the rightmost paths of both subtrees. By the heap ordering, the keys along each of these paths form an increasing sequence. We merge these paths into a single sorted path by selecting the one with the smaller key value (see Fig. 26).

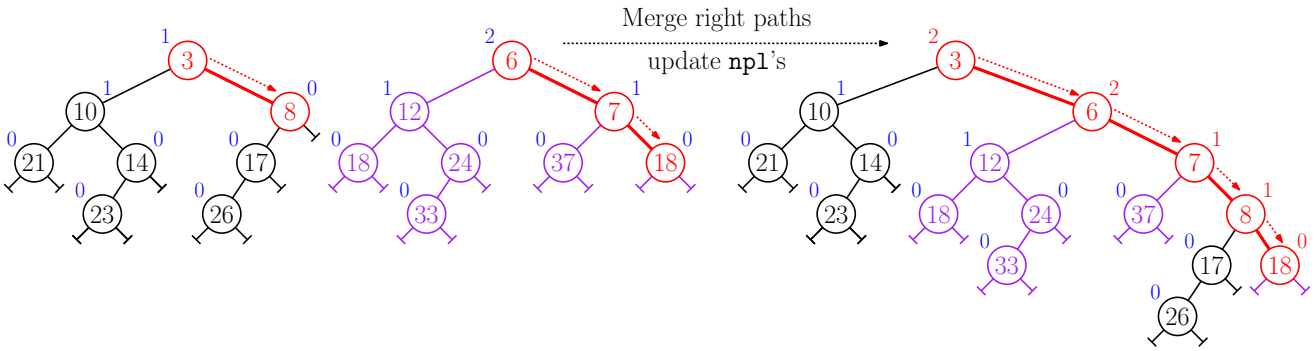



Fig. 26: Merging leftist heaps (Phase-1): Merge rightmost paths by key.

Each node of the tree contains an additional field `npl`, which stores the npl value for this node. After merging the right paths, we update the npl values. After the merger is done, the leftist property may be violated along this rightmost path. To remedy this, we perform a second phase, which swaps left and right children to restore the leftist property. (Recall that this swapping preserves the heap ordering.) This is illustrated in Fig. 27.

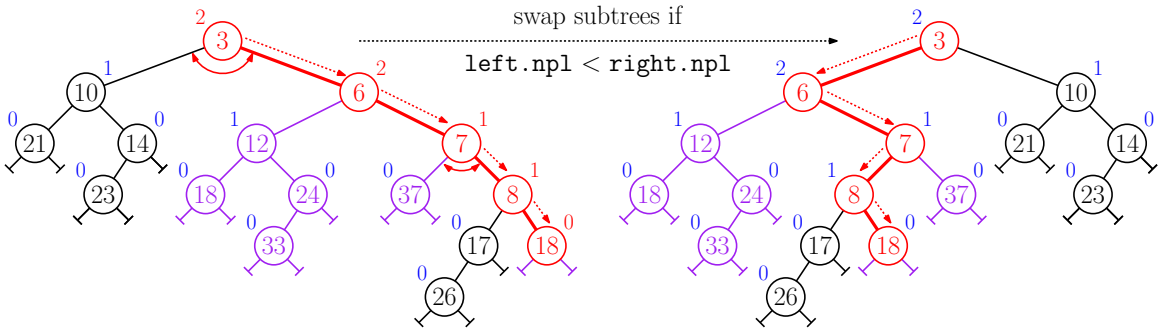



Fig. 27: Merging leftist heaps (Phase-2): Swap left-right to restore leftist property.

The complete code is given in the code block. It is expressed in recursive form, and both phases are combined into a single phase. First, it deals with the trivial cases if either `u` or `v` is empty (`null`), in which case it simply returns the other tree as the answer. It then swaps `u` and `v` so that `u` contains the smaller key. If `u` has no left child (which by leftism implies

Merge two leftist heaps

```
LHNode merge(LHNode u, LHNode v) {          // recursive helper function
    if (u == null) return v                 // if one is empty, return the other
    if (v == null) return u
    if (u.key > v.key) swap(u, v)           // swap so that u has smaller key
    if (u.left == null) u.left = v          // u must be a leaf in this case
    else {                                  // merge v on right and swap if needed
        u.right = merge(u.right, v)         // recursively merge u's right subtree
        if (u.left.npl < u.right.npl) {     // fail the leftist property?
            swap(u.left, u.right)           // swap to restore leftist
        }
        u.npl = u.right.npl + 1             // update npl value
    }
    return u                                // return the root of final tree
}
```

it has no right child), then we attach the entire subtree `v` as the left child of `u`. (Leftism demands that we put it on the left side.) We then recursively merge the remainder of `u`'s right subtree with `v`'s tree. After this returns, we swap the left and right subtrees if needed to satisfy the leftist property at `u`. Finally, we update `u`'s npl value and return `u` as the final result. (I would recommend tracing it out on the above example from Fig. 26 to see that it produces the same result as in Fig. 27.)

**Theorem:** Given two leftist heaps of total size $n$, the above merge procedure runs in time $O(\log n)$.

**Proof:** The merging procedure takes time proportional to the rightmost paths of the two trees. Since both trees have at most $n$ nodes, this is $O(\log n)$.

We have not explained two important operations, `insert` and `extact-min`. We will leave these as an exercise, but as a hint, both can be implemented in $O(\log n)$ time by performing a single invocation of `merge` together with operations that take only constant time.

## Lecture 6: Binary Search Trees

**Searching:** Searching is among the most fundamental problems in data structure design. Our objective is to store a set of *entries* $\{e_1, \ldots, e_n\}$, where each $e_i$ is a pair $(x_i, v_i)$, where $x_i$ is a *key value* drawn from some totally ordered domain (e.g., integers or strings) and $v_i$ is an associated *data value*. The data value is not used in the search itself, but is needed by whatever application is using our data structure.

We assume that keys are unique, meaning that no two entries share the same key. Given an arbitrary search key $x$, the basic search problem is determining whether there exists an entry matching this key value. To implement this, we will assume that we are given two types, `Key` and `Value`. We will also assume that key values can be compared using the usual comparison operators, such as `<`, `>`, `==`. We make no assumptions about `Value`, since it is used by the application only. (We will discuss Java implementation at the end of the lecture.)

**The Dictionary (Map) ADT:** A *dictionary* (also called a *map*) is an ADT that supports the operations insert, delete, and find:

`void insert(Key x, Value v)`: Stores an entry with the key-value pair $(x, v)$. We assume that keys are unique, and so if this key already exists, an error condition will be signaled (e.g., an exception will be thrown).

`void delete(Key x)`: Delete the entry with $x$'s key from the dictionary. If this key does not appear in the dictionary, then an error conditioned is signaled.

`Value find(Key x)`: Determine whether there is an entry matching $x$'s key in the dictionary? If so, it returns a reference to associated value. Otherwise, it returns a `null` reference.

There are a number of additional operations that one may like to support, such as iterating the entries, answering range queries (that is, reporting or counting all objects between some minimum and maximum key values), returning the $k$th smallest key value, and computing set operations such as union and intersection.

There are three common methods for storing dictionaries: sorted arrays, search trees, and hashing. We will discuss the first two in this a subsequent lectures. (Hashing will be presented later this semester.)

**Sequential Allocation:** The most naive approach for implementing a dictionary data structure is to simply store the entries in a linear array without any sorting. To find a key value, we simply run sequentially through the list until we find the desired key. Although this is simple, it is not efficient. Searching and deletion each take $O(n)$ time in the worst case, which is very bad if $n$ (the number of items in the dictionary) is large. Although insertion only involves $O(1)$ to insert a new item at the end of the array (assuming we don't overflow), it would require $O(n)$ time to check that we haven't inserted a duplicate element.

An alternative is to sort the entries by key value. Now, *binary search* can be used to locate a key in $O(\log n)$ time, which is much more efficient. (For example, if $n = 10^6$, $\log_2 n$ is only around 20.) While searches are fast, updates are slow. Insertion and deletion require $O(n)$ time, since the elements of the array must be moved around to make space.

**Binary Search Trees:** In order to provide the type of rapid access that binary search offers, but at the same time allows efficient insertion and deletion of keys, the simplest generalization is called a *binary search tree*. The idea is to store the records in the nodes of a binary tree. Each node, called a `BSTNode`, stores a key, `key`, an associated value, `value`, and pointers to its `left` and `right` children. (See Java implementation below for more information.)

These nodes are organized so that an inorder traversal visits the nodes in increasing key order. In particular, if $x$ is the key stored in some node, then the left subtree contains all keys that are less than $x$, and the right subtree stores all keys that are greater than $x$ (see Fig. 28(a)). (Recall that we assume that keys are distinct, so no other key can be equal to $x$.)

**Search in Binary Search Trees:** The search for a key $x$ proceeds as follows. We start by assigning a pointer $p$ to the root of the tree. We compare $x$ to $p$'s key, that is, `p.key`. If they are equal, we have found it and we are done. Otherwise, if $x$ is smaller, we recursively search $p$'s left subtree, and if $x$ is larger, we recursively visit $p$'s right subtree. The search proceeds until we either find the key (see Fig. 28(b)) or we fall out of the tree (see Fig. 28(b)). The code block below shows a possible pseudocode implementation of the find operation. The initial call is `find(x, root)`, where `root` is the root of the tree. It is easy to see based on the definition of a binary tree why this is correct.

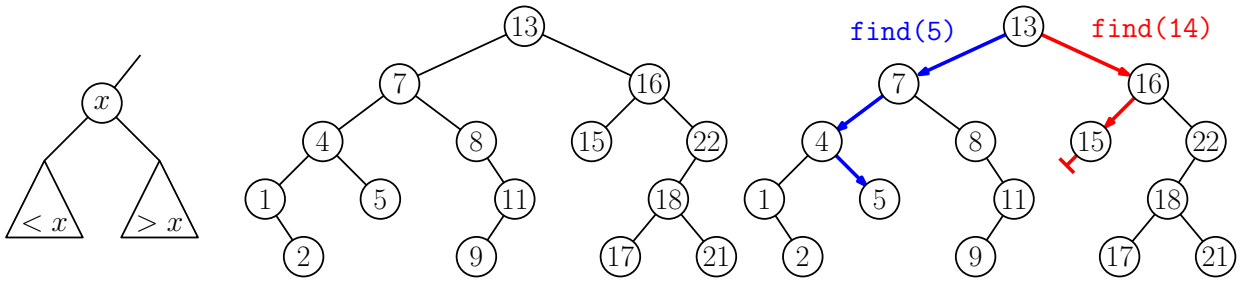



Fig. 28: A possible binary search tree for the key set $\{1, 2, 4, 5, 7, 8, 9, 11, 13, 15, 16, 17, 18, 21, 22\}$ and the searcg paths for `find(5)` and `find(14)`.

Recursive Binary Tree Search

```
Value find(Key x, BSTNode p) {
    if (p == null) return null;          // unsuccessful search
    else if (x < p.key)                  // x is smaller?
        return find(x, p.left);          // ... search left
    else if (x > p.key)                  // x is larger?
        return find(x, p.right);         // ... search right
    else return p.value;                 // successful search
}
```

**Query time:** What is the running time of the search algorithm? Well, it depends on the key you are searching for. In the worst case, the search time is proportional to the height of the tree. The height of a binary search tree with $n$ entries can be as low as $O(\log n)$ for the case of *balanced tree* (see Fig. 29 left) or as large as $O(n)$ for the case of a *degenerate tree* (see Fig. 29 right). However, we shall see that if the keys are inserted in random order, the expected height of the tree is just $O(\log n)$.

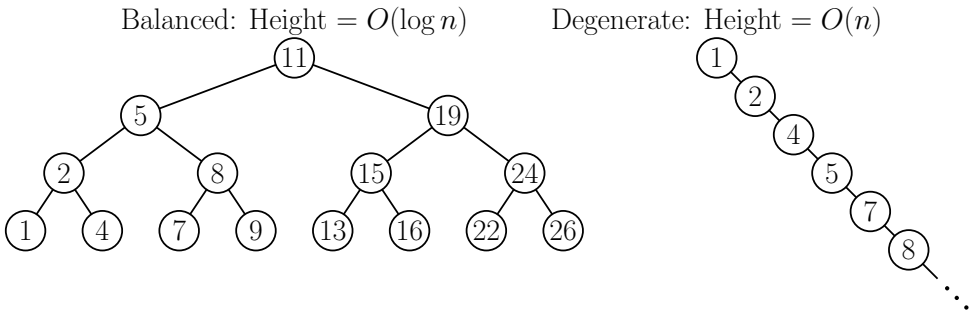



Fig. 29: Balanced and degenerate binary trees.

Can we devise ways to force the tree height to be $O(\log n)$? The answer is yes, and in future lectures we will see numerous approaches for guaranteeing this.

**Insertion:** To insert a new key-value entry $(x, v)$ in a binary search tree, we first try to locate the key in the tree. If we find it, then the attempt to insert a duplicate key is an error. If not, we effectively "fall out" of the tree at some node $p$. We insert a new leaf node containing the desired entry as a child of $p$. It turns out that this is always the right place to put the new node. (For example, in Fig. 30, we fall out of the tree at the left child of node 15, and we insert the new node with key 14 here.)

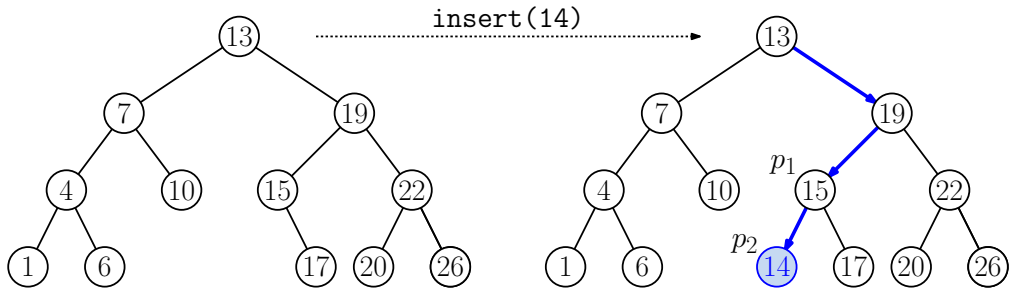



Fig. 30: Binary tree insertion.

This can naturally be implemented recursively. If the inserted key is smaller than the current node's key, we insert recursively on the left. If it is greater, we insert on the right. (It cannot be equal, or we have a duplicate key!) When we fall out of the tree (a `null` link), we create a new node here. The pseudocode is presented in the code block below. The initial call is `root = insert(x, v, root)`. We assume that there is a constructor for the `BSTNode`, which is given the key, value, and the initial values of the left and right child pointers.

Recursive Binary Tree Insertion

```
BSTNode insert(Key x, Value v, BSTNode p) {
    if (p == null)                          // fell out of the tree?
        p = new BSTNode(x, v, null, null);  // ... create a new leaf node here
    else if (x < p.key)                     // x is smaller?
        p.left  = insert(x, v, p.left);     // ...insert left
    else if (x > p.key)                     // x is larger?
        p.right = insert(x, v, p.right);    // ...insert right
    else throw DuplicateKeyException;       // x is equal ...duplicate key!
    return p                                // return ref to current node (sneaky!)
}
```

**Coding Trick:** (*Be sure you understand this!*) You will see one technical oddity in the above pseudocode implementation. Consider the line: "`p.left = insert(x, v, p.left)`". Why does the insert function return a value, and why do we replace the left-child link with this value?

Here is the issue. Whenever we create the new node, we need to "reach up" and modify the appropriate child link in the parent's node. Unfortunately, this is not easy to do in our recursive formulation, since the parent node is not a local variable. To do this, our insert function will return a reference to the modified subtree after the insertion. We store this value in the appropriate child pointer for the parent.

To better understand how our coding trick works, let's refer back to Fig. 30. Let $p_1$ denote the node containing 15. Since $14 < 15$, we effectively invoked the command "`p1.left = insert(x, v, p1.left)`". But since this node has no left child (`p1.left == null`), this recursive call creates a new `BSTNode` containing 14 and returns a pointer to it, call it $p_2$. Since this is the return value from the recursive call, we effectively have executed "`p1.left = p2`", thus linking the 14 node as the left child of the 15 node. Slick!

**Deletion:** Next, let us consider how to delete an entry from the tree. Deletion is a more involed than insertion. While insertion adds nodes at the leaves of the tree, but deletions can occur

at any place within the tree. Deleting a leaf node is relatively easy, since it effectively involves "undoing" the insertion process (see Fig. 31(a)). Deleting an internal node requires that we "fill the hole" left when this node goes away. The easiest case is when the node has just a single child, since we can effectively slide this child up to replace the deleted node (see Fig. 31(b)).

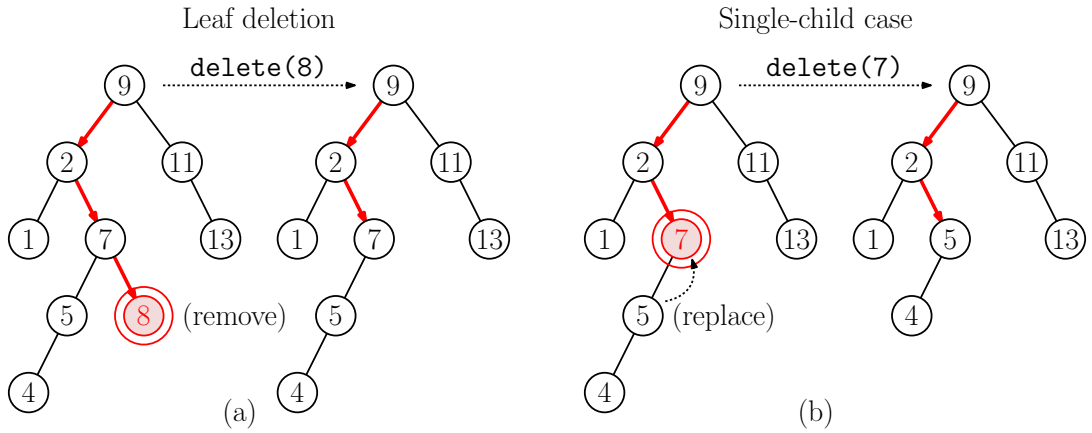



Fig. 31: Deletion: (a) Leaf and (b) single-child case.

The hardest case is when the deleted node that has two children. Let $p$ denote the node to be deleted (see Fig. 32(a)):

- First off, we we need to find a suitable *replacement node*, whose key/value will fill the hole left by deletion. We can either take the largest key from $p$'s left subtree (its inorder predecessor) or the smallest key from its right subtree (its inorder successor). Let's arbitrarily decide to do the latter. Call this node $r$ (see Fig. 32(b)). Note that because $p$ has two children, its inorder successor is the "leftmost" node of $p$'s right subtree. Call $r$ the *replacement node*.
- Copy the contents of $r$ (both key and value) to $p$ (see Fig. 32(c)).
- Recursively delete node $r$ from $p$'s right subtree (see Fig. 32(d)).

It may seem that we have made no progress, because we have just replaced one deletion problem (for $p$) with another (for $r$). However, the task of deleting $r$ is much simpler. The reason is, since $r$ is $p$'s inorder successor, $r$ is the leftmost node of $p$'s right subtree. It follows that $r$ has no left child. Therefore, $r$ is either a leaf or it has a single child, implying that it is one of the two "easy" deletion cases that we discussed earlier.

**Deletion Implementation:** Before giving the code for deletion, we first present a utility function, `findReplacement()`, which returns a pointer to the node that will replace $p$ in the two-child case. As mentioned above, this is the inorder successor of $p$, that is, the leftmost node in $p$'s right subtree. As with the insertion method, the initial call is made to the root of the tree, `delete(x, root)`. Again, we will employ the sneaky trick of returning a pointer to the revised subtree after deletion, and store this value in the child link. See the code fragment below.

The full deletion code is given in the following code fragment. As with insertion, the code is quite tricky. For example, can you see where the leaf and single-child cases are handled

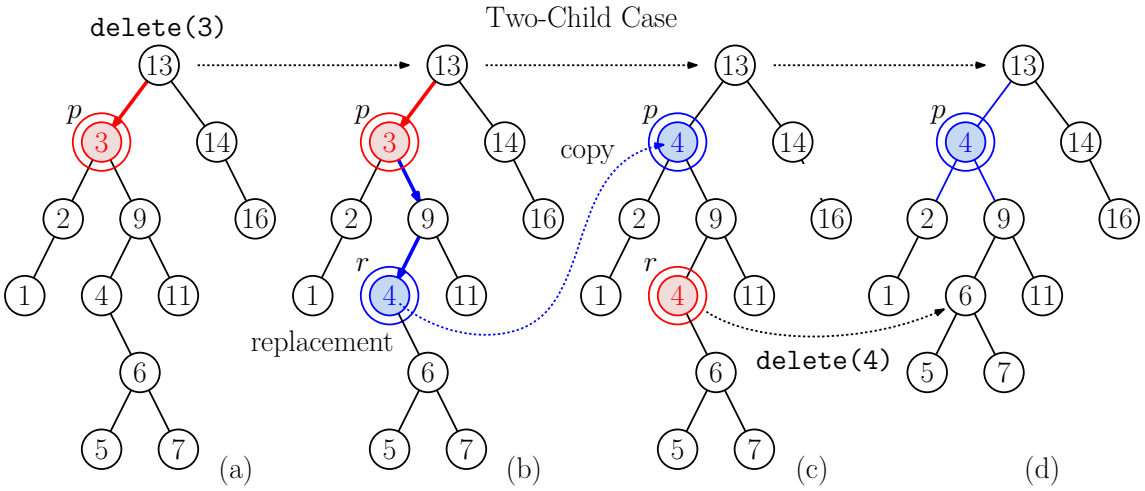



Fig. 32: Deletion: Two-child case.

Replacement Node for the Two-child Case

```
BSTNode findReplacement(BSTNode p) {
    BSTNode r = p.right;
    while (r.left != null) r = r.left;
    return r;
}
```

```
// find p's replacement node
// start in p's right subtree
// go to the leftmost node
```

in the code? We do not have a conditional that distinguishes between these cases. How can that be correct. (But it is!)

**Analysis of Binary Search Trees:** It is not hard to see that all of the procedures `find()`, `insert()`, and `delete()` run in time that is proportional to the height of the tree being considered. (The `delete()` procedure is the only one for which this is not obvious. Because the replacement node is the inorder successor of the deleted node, it is the leftmost node of the right subtree. This implies that the replacement node has no left child, and so it will fall into one of the easy cases, which do not require a recursive call.)

The question is, given a binary search tree $T$ containing $n$ keys, what can be said about the height of the tree? It is easy to devise "terrible" insertion orders so that the tree has the worst possible height of $n - 1$. Conversely, it is possible to devise "perfect" insertion orders, which result in a balanced tree of height $\lfloor \log_2 n \rfloor$. (As an exercise, think about how you would do either of these.)

Since the worst case is obviously very bad, let's consider the expected case. Suppose that the keys are inserted in random order. To be more precise, given $n$ keys, let us assume that each of the $n!$ insertion orders is equally likely. What is the *expected height* of the resulting tree? It turns out to be pretty good!

**Theorem:** Given a set of $n$ keys $x_1 < x_2 < \ldots < x_n$, let $H(n)$ denote the expected height of a binary search tree, under the assumption that every one of the $n!$ insertion orders is equally likely. Then $H(n) = O(\log n)$.

Proving this is not an trivial exercise. (If you saw the expected-case analysis for Quicksort in CMSC 351, it is quite similar.)

```
BSTNode delete(Key x, BSTNode p) {
    if (p == null)                                  // fell out of tree?
        throw KeyNotFoundException;                 // ...error - no such key
    else {
        if (x < p.data)                             // look in left subtree
            p.left = delete(x, p.left);
        else if (x > p.data)                        // look in right subtree
            p.right = delete(x, p.right);
                                                    // found it!
        else if (p.left == null || p.right == null) { // either child empty?
            if (p.left == null) p = p.right;        // return replacement node
            else                p = p.left;
        }
        else {                                      // both children present
            r = findReplacement(p);                 // find replacement node
            copy r's contents to p;                 // copy its contents to p
            p.right = delete(r.key, p.right);       // delete the replacement
        }
    }
    return p;
}
```

**Quick and Dirty Analysis:** (*Optional*)

Rather than give the full proof, we will provide a much simpler one, which will hopefully convince you that the assertion is reasonable. Instead of proving that *every* node of the tree is at expected depth $O(\log n)$, we will prove that the *leftmost* node of the tree (that is, the node associated with the smallest key value) will be at expected depth $O(\log n)$.

**Theorem:** Given a set of $n$ keys $x_1 < x_2 < \ldots < x_n$, let $D(n)$ denote the expected depth of *leftmost* node $x_1$ after inserting all these keys in a binary search tree, under the assumption that all $n!$ insertion orders are equally likely. Then $D(n) \leq 1 + \ln n$, where ln denotes the natural logarithm.

**Proof:** We will track the depth of the leftmost node of the tree through the sequence of insertions. Suppose that we have already inserted $i - 1$ keys from the sequence, and we are in the process of inserting the $i$th key. The only way that the leftmost node changes is when the $i$th key is the lowest key value that has been seen so far in the sequence. That is, the $i$ element to be inserted in the new minimum value among all the keys in the tree.

For example, consider the insertion sequence $S = \langle 9, 5, 10, 6, 3, 4, 2 \rangle$. Observe that the minimum value changes three times, when 5, 3, and 2 are inserted. If we look at the binary tree that results from this insertion sequence we see that the depth of the leftmost node also increases by one with each of these insertions.

To complete the analysis, it suffices to determine (in expectation) the number of times that the minimum in a sequence of $n$ random values changes. To make this formal, for $2 \leq i \leq n$, let $X_i$ denote the random variable that is 1 if the $i$th element of the random sequence is the minimum among the first $i$ elements, and 0 otherwise. (In our sequence $S$ above, $X_2 = X_5 = X_7 = 1$, because the minimum changed when the second, fifth, and seventh elements were added. The remaining $X_i$'s are zero.)

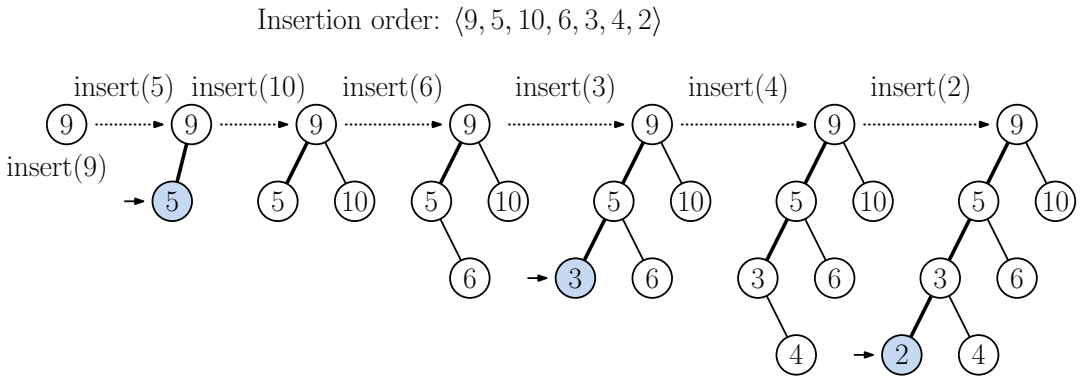



Fig. 33: Length of the leftmost chain.

To analyze $X_i$, let's just focus on the first $i$ elements and ignore the rest. Since every permutation of the numbers is equally likely, the minimum among the first $i$ is equally likely to come at any of the positions first, second, ..., up to $i$th. The minimum changes only if comes last out of the first $i$. Thus, $\Pr(X_i = 1) = \frac{1}{i}$ and $\Pr(X_i = 0) = 1 - \frac{1}{i}$. Whenever this random event occurs ($X_i = 1$), the minimum has changed one more time. Therefore, to obtain the expected number of times that the minimum changes, we just need to sum the probabilities that $X_i = 1$, for $i = 2, \ldots, n$. Thus we have

$$D(n) \;=\; \sum_{i=2}^{n} \frac{1}{i} \;=\; \left(\sum_{i=1}^{n} \frac{1}{i}\right) - 1.$$

This summation is among the most famous in mathematics. It is called the *Harmonic Series*. Unlike the geometric series ($1/2^i$), the Harmonic Series does not converge. But it is known that when $n$ is large, its value is very close to $\ln n$, the natural log of $n$. (In fact, it is not more than $1 + \ln n$.)

Therefore, we conclude that the expected depth of the leftmost node in a binary search tree under $n$ random insertions is at most $1 + \ln n = O(\log n)$, as desired.

**Random Insertions and Deletions:** Interestingly, the analysis of the expected-case tree height breaks down if we perform both insertions and deletions. Suppose that we consider a very long sequence of insertions and deletions, which occur at roughly the same rate so that, in steady state, the tree has roughly $n$ nodes. Let us also assume that insertions are random (drawn say from some large domain of candidate elements) and deletions are random in the sense that a random element from the tree is deleted each time.

It is natural to suppose that the $O(\log n)$ bound should apply, but remarkably it does not! It can be shown that over a long sequence, the height of the tree will converge to a significantly larger value of $O(\sqrt{n})$.[4]

[4]There is an interesting history regarding this question. It was believed for a number of years that random deletions did not alter the structure of the tree. A theorem by T. N. Hibbard in 1962 proved that the tree structure was probabilistically unaffected by deletions. The first edition of D. E. Knuth's famous book on data structures, quotes this result. In the mid 1970's, Gary Knott, a Ph.D. student of Knuth and later a professor at UMD, discovered a subtle flaw in Hibbard's result. While the structure of the tree is probabilistically the same, the distribution of keys is not. However, Knott could not resolve the asymptotic running time. The analysis showing that $O(\sqrt{n})$ bound was due to Culberson and Munro in the mid 1980's.

The reason has to do with the fact that the replacement element was chosen in a "biased" manner, always taking the inorder successor. Over the course of many deletions, this repeated bias causes the tree's structure to skew away from the ideal. This can be remedied by selecting the replacement node in an unbiased manner, choosing randomly between the inorder successor or inorder predecessor. It has been shown experimentally that this resolves the issue, but (to the best of my knowledge) it is not known whether the expected height of this balanced version of deletion matches the expected height for the insertion-only case (see Culberson and Munro, Algorithmica, 1990).

**Java Implementation:** So far, we have been expressing our functions in pseudocode. Defining a binary search tree object in a modern object-oriented language like Java would be done in a slightly different manner. First, rather than fixing specific types for the key and value, we would make the parameterized (generic) types. Let's call these `Key` and `Value`, respectively. We should not assume that we can simply apply operators such as "`<`", "`>`", and "`==`" for comparing keys. For this reason, we will assume that our `Key` object implements the `Comparable<Key>` interface, which means that it defines a method `compareTo`, which compares two objects of type `Key`. (By the way, this is not the only way to achieve this. Another approach is by associating a `Comparator` with the tree. Comparators offer some added flexibility, since multiple comparators can be defined for the same object, allowing you to sort them according to different criteria.)

In order to implement nodes, the class `BinarySearchTree` can define an *inner class*, called `BSTNode`. In Java, we can use the default access modifier. This means that the `BSTNode` members are accessible within the same package but hidden from the outside. Public functions, like `find`, `insert`, and `delete`, each invoke corresponding local functions, each of which is invoked on the root of the tree. A partial example (showing just the `find` function) is shown in the following code fragment.

## Lecture 7: AVL Trees

**Balanced Binary Trees:** The binary search trees described in the previous lecture are easy to implement, but they suffer from the fact that if nodes are inserted in a poor order (e.g., increasing or decreasing) then the height of the tree can be much higher than the ideal height of $O(\log n)$. This raises the question of whether we can design a binary search tree that is *guaranteed* to have $O(\log n)$ height, irrespective of the order of insertions and deletions.

Today we will consider the oldest, and perhaps best known example of such a data structure is the famous AVL tree, which was discovered way back in 1962 by G. Adelson-Velskii and E. Landis (and hence the name "AVL").

**AVL Trees:** AVL tree's are height-balanced binary search trees. For any node $v$ of the tree, let height($v$) denote the height of the subtree rooted at $v$ (shown in blue in Fig. 34(a)).

Formally, we can define *height* recursively as follows

$$\text{height}(v) \;=\; \begin{cases} -1 & \text{if } v = \texttt{null} \\ 1 + \max(\text{height}(v.\text{left}), \text{height}(v.\text{right})) & \text{otherwise} \end{cases}$$

Setting the height of `null` to $-1$ is a convenient trick to force leaves to have a height of zero.

(Partial) Java Binary Tree

```
public class BinarySearchTree<Key extends Comparable<Key>, Value> {

    class BSTNode {                          // a node of the tree
        Key key;
        Value value;
        BSTNode left;
        BSTNode right;

        // ... other utility methods omitted
    }

    Value find(Key x, BSTNode p) {           // find helper function
        if (p == null) return null;          // unsuccessful search
        else if (x.compareTo(p.key) < 0)     // x is smaller?
            return find(x, p.left);          // ... search left
        else if (x.compareTo(p.key) > 0)     // x is larger?
            return find(x, p.right);         // ... search right
        else return p.value;                 // successful search
    }

    // ... other methods (insert, delete, ...) omitted

    private BSTNode root;                    // root of tree (private)

    public Value find(Key x) {               // public find key
        return find(x, root);                // invoke find helper
    }

    // ... other public functions
}
```

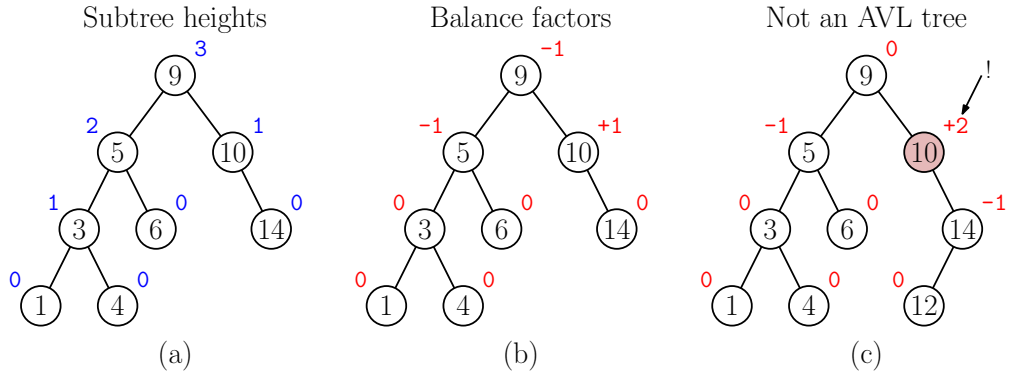



Fig. 34: AVL-tree balance condition.

In an absolutely ideal height-balanced tree, the two children of any internal node would have equal heights, but it is not generally possible to achieve this goal while efficiently processing insertions and deletions. The most natural relaxation of this condition is to allow a height difference of one. Define the *AVL balance condition* to be that for every node in the tree, the absolute difference in the heights of its left and right subtrees is at most 1. An *AVLtree* is a binary search tree that satisfies the AVL balance condition.

To formalize this, define the *balance factor* of a node $v$ (see Fig. 34(b)) to be

$$\text{balance}(v) \;=\; \text{height}(v.\text{right}) - \text{height}(v.\text{left}).$$

The AVL balance condition is equivalent to the requirement that $\text{balance}(v) \in \{-1, 0, +1\}$ for all nodes $v$ of the tree. (Thus, Fig. 34(b) is an AVL tree, but the tree of Fig. 34(c) is not because node 10 has a balance factor of $+2$.) If $\text{balance}(v) < -1$, we say that the subtree is *left heavy* and if $\text{balance}(v) > +1$, we say that the subtree is *right heavy*.

**Worst-case Height:** Before discussing how we maintain this balance condition we should consider the question of whether this condition is strong enough to guarantee that the height of an AVL tree with $n$ nodes is $O(\log n)$. Interestingly, the famous Fibonacci numbers $(0, 1, 1, 2, 3, 5, 8, \ldots)$ will arise in the analysis. Recall that for $h \geq 0$, the $h$th *Fibonacci number*, denoted $F_h$ is defined by the following recurrence:

$$F_0 \;=\; 0, \quad F_1 \;=\; 1, \quad \text{and} \quad F_h \;=\; F_{h-1} + F_{h-2}, \quad \text{for } h \geq 2.$$

An important and well-known property of the Fibonacci numbers is that they grow exponentially. In particular, $F_h \approx \varphi^h/\sqrt{5}$, where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$ is the famous *Golden Ratio*.[5]

**Lemma:** An AVL tree of height $h \geq 0$ has $\Omega(\varphi^h)$ nodes, where $\varphi = (1 + \sqrt{5})/2$.

**Proof:** For $h \geq 0$, let $N(h)$ denote the minimum possible number of nodes in binary tree of height $h$ that satisfies the AVL balance condition. We will prove that $N(h) = F_{h+3} - 1$ (see Fig. 35). The result will then follow from the fact that $F_{h+3} \approx \varphi^{h+3}/\sqrt{5}$, which is equal to $\varphi^h$ up to constant factors (since $\varphi$ itself is a constant).

Our proof is by induction on $h$. First, observe that a tree of height zero consists of a single root node, so $N(0) = 1$. Also, the smallest possible AVL tree of height one consists of a root and a single child, so $N(1) = 2$. By definition of the Fibonacci numbers, we have $F_{0+3} = 2$ and $F_{1+3} = 3$, and thus $N(i) = F_{i+3} - 1$, for these two basis cases.

For $h \geq 2$, let $h_L$ and $h_R$ denote the heights of the left and right subtrees, respectively. Since the tree has height $h$, one of the two subtrees must have height $h - 1$, say, $h_L$. To minimize the overall number of nodes, we should make the other subtree as short as possible. By the AVL balance condition, this implies that $h_R = h - 2$. Adding a $+1$ for the root, we have $N(h) = 1 + N(h-1) + N(h-2)$. We may apply our induction hypothesis to conclude that

$$\begin{aligned} N(h) \;&=\; 1 + N(h-1) + N(h-2) \;=\; 1 + (F_{h+2} - 1) + (F_{h+1} - 1) \\ &=\; F_{h+2} + F_{h+1} - 1 \;=\; F_{h+3} - 1, \end{aligned}$$

[5]Here is a sketch of a proof. Let us conjecture that $F_h \approx \varphi^h$ for some constant $\varphi$. Since the function grows very (exponentially) fast, we may ignore the tiny contribution of the $+1$ in the definition for large $h$. Substituting our conjectured value for $F_h$ into the above recurrence, we find the $\varphi$ satisfies $\varphi^h = \varphi^{h-1} + \varphi^{h-2}$. Removing the common factor of $\varphi^{h-2}$, we have $\varphi^2 = \varphi + 1$, that is, $\varphi^2 - \varphi - 1 = 0$. This is a quadratic equation, and by applying the quadratic formula, we conclude that $\varphi = (1 + \sqrt{5})/2$.

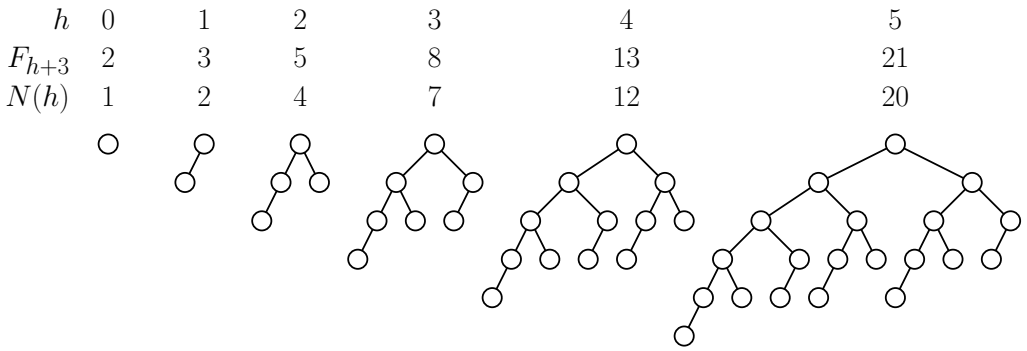



Fig. 35: Minimal AVL trees and Fibonacci numbers.

as desired.

**Corollary:** An AVL tree with $n$ nodes has height $O(\log n)$.

**Proof:** Let lg denote logarithm base 2. From the above lemma, up to constant factors we have $n \geq \varphi^h$, which implies that $h \leq \log_\varphi n = \lg n / \lg \varphi$. Since $\varphi > 1$ is a constant, so is $\log \varphi$. Therefore, $h$ is $O(\log n)$. (If you work through the math, the actual bound on the height is roughly $1.44 \lg n$. In other words, in the worst case, an AVL tree is suboptimal with respect to height by a factor of at most 1.44)

Since the height of the AVL tree is $O(\log n)$, it follows that the `find` operation takes this much time. All that remains is to show how to perform insertions and deletions in AVL trees, and how to restore the AVL balance condition efficiently after each insertion or deletion.

**Rotation:** In order to maintain the tree's balance, we will employ a simple operation that locally modifies subtree heights, while preserving the tree's inorder properties. This operation is called *rotation*. It comes in two symmetrical forms, called a *right rotation* and a *left rotation* (see Fig. 36(a) and (b)).

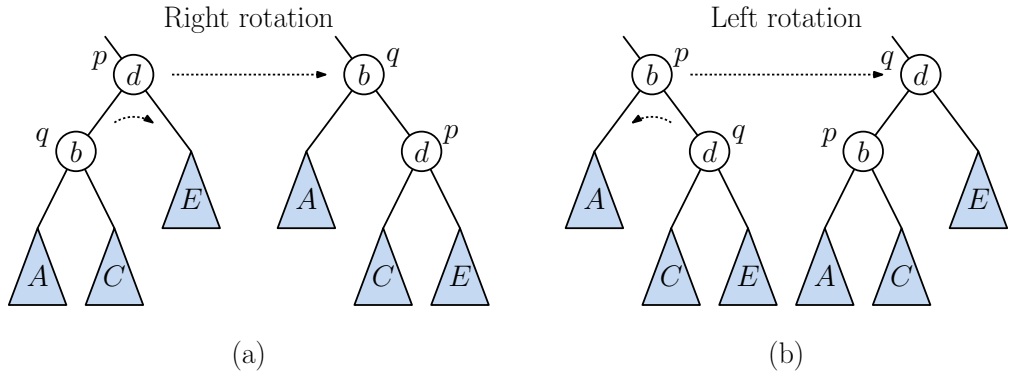



Fig. 36: (Single) Rotations. (Triangles denote subtrees, which may be null.)

We have intentionally labeled the elements of Fig. 36 to emphasize the fact that the inorder properties of the tree are preserved. That is, $A < b < C < d < E$.

Unfortunately, a single rotation is not always be sufficient to rectify a node that is out of balance. To see why, observe that the single rotation does not alter the height of subtree $C$. If it is too heavy, we need to do something else to fix matters. This is done by combining

two rotations, called a *double rotation*. They come in two forms, *left-right rotation* and *right-left rotation* (Fig. 37). To help remember the name, note that the left-right rotation, called `rotateLeftRight(p)`, is equivalent to performing a left rotation to the `p.left` (labeled $b$ in Fig. 37(a)) followed by a right rotation to `p` (labeled $d$ in Fig. 37(a)). The right-left rotation is symmetrical (see Fig. 37(b)).

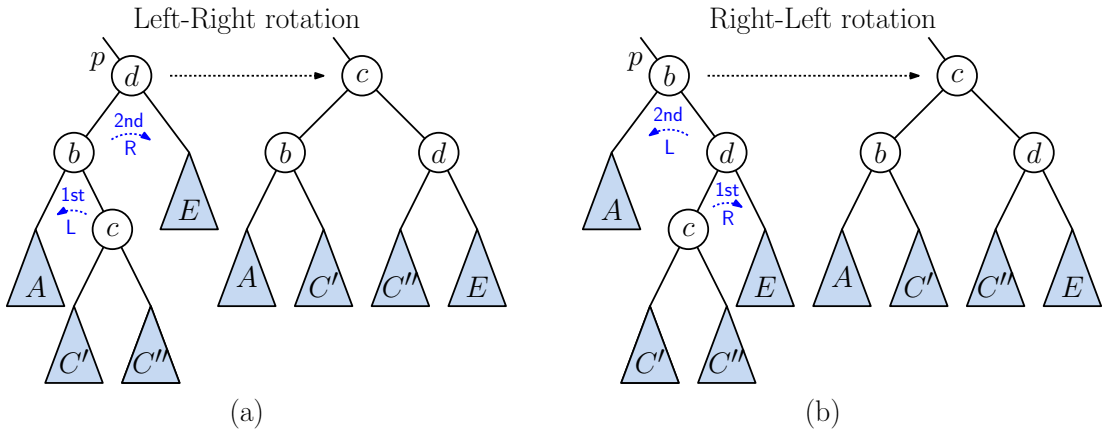



Fig. 37: Double rotations (`rotateLeftRight(p)` and `RotateRightLeft(p)`).

**Insertion:** The insertion routine for AVL trees starts exactly the same as the insertion routine for standard (unbalanced) binary search trees. In particular, we search for the key and insert a new node at the point that we fall out of the tree. After the insertion of the node, we must update the subtree heights, and if the AVL balance condition is violated at any node, we then apply rotations as needed to restore the balance.

The manner in which rotations are applied depends on the nature of the imbalance. An insertion results in the addition of a new leaf node, and so the balance factors of the ancestors can be altered by at most $\pm 1$. Suppose that after the insertion, we find that some node has a balance factor of $-2$. For concreteness, let us consider the naming of the nodes and subtrees shown in Fig. 38, and let the node in equation be $d$. Note that this node must be along the search path for the inserted node, since these are the only nodes whose subtree heights may have changed. Clearly, $d$'s left subtree, is too deep relative to $d$'s right subtree $E$. Let $b$ denote the root of $d$'s left subtree.

At this point there are two cases to consider. Either $b$'s left child is deeper or its right child is deeper. (The subtree that is deeper will be the one into which the insertion took place.)

**Left-left heavy:** Let's first consider the case where the insertion took place in the the subtree $A$ (see Fig. 38(b)). In this case, we can restore balance by performing a right rotation at node $d$. This operation pulls the deep subtree $A$ up by one level, and it pushes the shallow subtree $E$ down by one level (see Fig. 38(c)). Observe that the depth of subtree $C$ is unaffected by the operation. It follows that the balance factors of the resulting subtrees rooted at $b$ and $d$ are now both zero. The AVL balance condition is satisfies by all nodes, and we are in good shape.

**Left-right heavy:** Next, us consider the case where the insertion occurs within subtree $C$ (see Fig. 39(b)). As observed earlier, the rotation at $d$ does not alter $C$'s depth, so we will need to do something else to fix this case. Let $c$ be the root of the subtree $C$, and

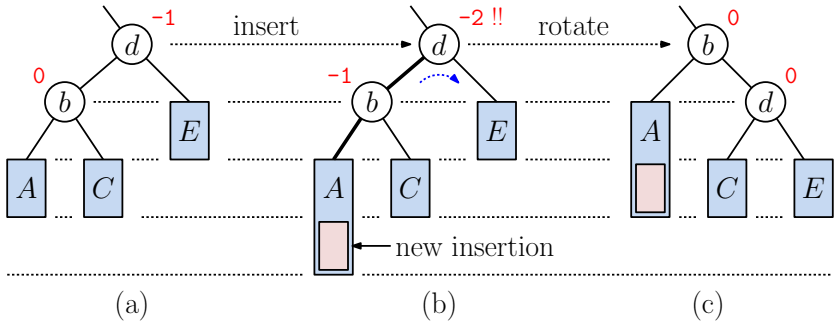



Fig. 38: Restoring balance after insertion through a single rotation.

let $C'$ and $C''$ be its two subtrees (either of these might be `null`). The insertion took place into either $C'$ or $C''$. (We don't care which, but the "?" in the figure indicate our uncertainty.) We restore balance by performing two rotations, first a left rotation at $b$ and then a right rotation at $d$ (see Fig. 39(c)). This *double rotation* has the effect of moving the subtree $E$ down one level, leaving $A$'s level unchanged, and pulling both $C'$ and $C''$ up by one level.

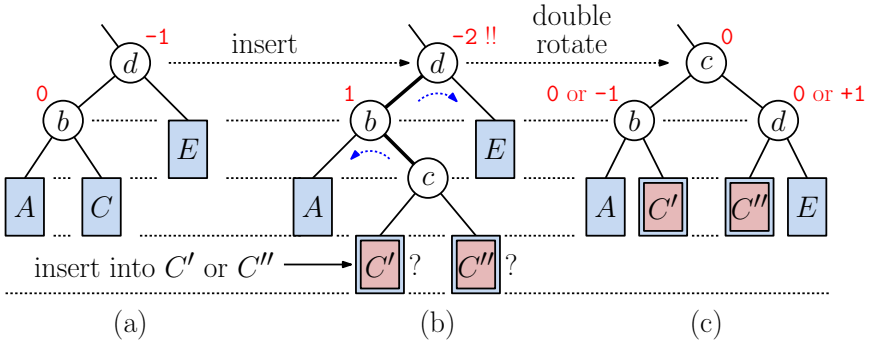



Fig. 39: Restoring balance after insertion through a double rotation.

The balance factors at nodes $b$ and $d$ will depend on whether the insertion took place into $C'$ or $C''$, but irrespective of which, they will be in the range from $-1$ to $+1$. The balance factor at the new root node $c$ is now 0. So, again we are all good with respect to the AVL balance condition.

**Utilities:** Before giving the code for insertion and deletion, let's introduce a few utilities. We assume that we store the height of each node in a field $p$.height. We first define utility functions `height` and `updateHeight`, which access and update height information. Next, `balanceFactor` computes the balance factor from the child heights. Finally, we give code for single and double rotations.

**AVL Insertion:** The insert function is virtually identical to the insert function for standard binary search trees, but with one change. After the insertion, we invoke a function `rebalance`, which updates heights, checks balance factors, and applies the appropriate rotations to restore the AVL balance properties. To determine which rotation to apply, we first determine whether we are left or right heavy. If we are left heavy, then we determine whether we are left-left heavy or left-right heavy. In the former case, we do a single rotation, and in the latter we do a double rotation. The right-heavy cases are symmetrical.

AVL Tree Utilities

```
int height(Node p) { return p == null ? -1 : p.height }
void updateHeight(Node p) { p.height = 1 + max(height(p.left), height(p.right)) }
int balanceFactor(Node p) { return height(p.right) - height(p.left) }

Node rotateRight(Node p) {              // right single rotation
    Node q = p.left
    p.left = q.right                    // swap inner child
    q.right = p                         // bring q above p
    updateHeight(p)                     // update subtree heights
    updateHeight(q)
    return q                            // q replaces p
}

Node rotateLeft(Node p) { ... symmetrical to rotateRight ... }

Node rotateLeftRight(Node p) {          // left-right double rotation
    p.left = rotateLeft(p.left)
    return rotateRight(p)
}
Node rotateRightLeft(Node p) { ... symmetrical to rotateLeftRight ... }
```

AVL Tree Insertion

```
Node insert(Key x, Value v, Node p) {   // insert (x,v) into p's subtree
    if (p == null) {                    // fell out of tree -- create new node
        p = new Node(x, v, null, null)
    }
    else if (x < p.key) {               // x is smaller - insert left
        p.left = insert(x, p.left)      // ... insert left
    else if (x > p.key) {               // x is larger - insert right
        p.right = insert(x, p.right)    // ... insert right
    }
    else throw DuplicateKeyException    // key already in the tree?
    return rebalance(p)                 // ONLY CHANGE FROM STANDARD BST INSERT
}

Node rebalance(Node p) {                // rebalance subtree at p
    if (p == null) return p             // null - nothing to do
    if (balanceFactor(p) < -1) {        // left heavy?
        if (height(p.left.left) >= height(p.left.right)) { // left-left heavy?
            p = rotateRight(p)          // fix with single rotation
        else                            // left-right heavy?
            p = rotateLeftRight(p)      // fix with double rotation
    } else if (balanceFactor(p) > +1) { // right heavy?
        if (height(p.right.right) >= height(p.right.left)) { // right-right heavy?
            p = rotateLeft(p)           // fix with single rotation
        else                            // right-left heavy?
            p = rotateRightLeft(p)      // fix with double rotation
    }
    updateHeight(p)                     // update p's height
    return p                            // return link to updated subtree
}
```

An interesting feature of the insertion algorithm (which is not at all obvious) is that whenever rebalancing is required, the height of the modified subtree is the same as it was before the insertion. This implies that no further rotations are required. (This is not the case for deletion, however.)

**Deletion:** After having put all the infrastructure together for rebalancing trees, deletion is actually relatively easy to implement. As with insertion, deletion process is the same as for standard unbalanced binary search trees. The only difference is that as we are backing out of the recursion following the deletion, we need to update heights, check balance factors, and perform the necessary rotations.

As with insertion, there are multiple cases. If a node is out of balance, it is either left-heavy or right-heavy (but here, the heaviness is due to a deletion from the opposite side of the tree). We need only consider left-heaviness, since right-heaviness is symmetrical.

We say that the node `p` is left-left heavy if its left-left grandchild's height is greater than or equal to its left-right grandchild (see Fig. 40). If so, we remedy this with a single right rotation at `p`.

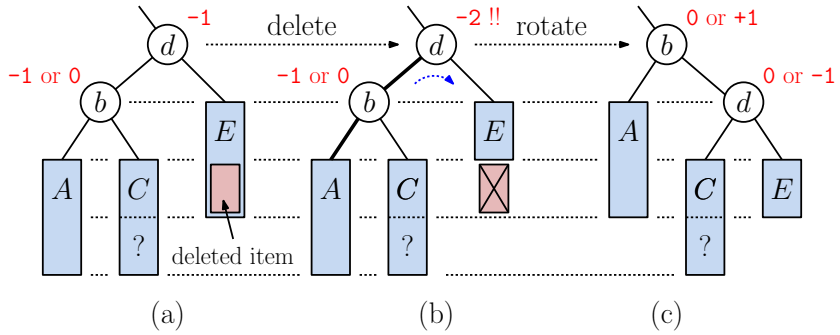



Fig. 40: Restoring balance after deletion in the left-left heavy case. Node $d$ is left heavy following the deletion from its right subtree. The left-left grandchild $A$ is at least as tall as the left-right grandchild $C$. ()The "?" illustrate places where the subtree's height is not fully determined.) A single right rotation at $d$ restores balance.

The other possibility is that `p` is left-right heavy, meaning that its left-right grandchild is strictly taller than its left-left grandchild (see Fig. 41). In this case, we remedy the balance through a left-right double rotation.

Note that in the case of the double rotation, the height of the entire tree rooted at $d$ has decreased by 1. This means that further ancestors need to be checked for the balance condition. Unlike insertion, where at most one rebalancing operation is needed, deletion could result in a cascade of $O(\log n)$ rebalancing operations.

**AVL Deletion:** The pseudocode for deletion is presented below. The code is identical to the deletion code for standard (unbalanced) binary search trees, except for the final call to `rebalance`.

**Lazy Deletion:** (Optional) The deletion code for standard binary search tree (and, by extension, AVL trees and other balanced search trees) is generally more complicated than the insertion code. An intriguing alternative for avoiding coding up the deletion algorithm is called *lazy deletion*. For each node, we maintain a boolean value indicating whether this element is *alive* or *dead*. When a key is deleted, we simply declare it to be dead, but leave it in the tree. If

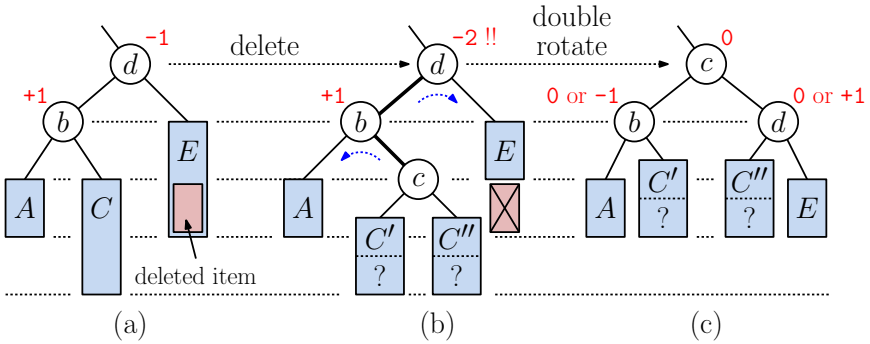



Fig. 41: Restoring balance after deletion in the left-right heavy case. Node $d$ is left heavy following the deletion from its right subtree. The left-right grandchild $C$ is strictly taller than its left-left grandchild $A$. A left-right rotation at $d$ restores balance.

AVL Tree Deletion

```
Node delete(Key x, Node p) {                  // delete x from p' subtree
    if (p == null)                            // fell out of tree?
        throw KeyNotFoundException            // ...error -- no such key
    else {
        if (x < p.data)                       // delete from left subtree
            p.left = delete(x, p.left)
        else if (x > p.data)                  // delete from right subtree
            p.right = delete(x, p.right)
                                              // found it!
        else if (p.left == null || p.right == null) { // 0 or 1 child
            if (p.left == null) p = p.right
            else                p = p.left
        }
        else {                                // both children present
            r = findReplacement(p)            // find replacement node
            copy r's contents to p            // copy its contents to p
            p.right = delete(r.key, p.right) // delete the replacement
        }
    }
    return rebalance(p)                       // ONLY CHANGE FROM STANDARD BST DELETE
}
```

an attempt is made to insert a value that comes in the same relative order as a dead key, we store the new key-value pair in the dead node and declare it to now be alive. Of course, your tree may generally fill up with lots of dead nodes, so lazy deletion is usually applied only in circumstances where the number of deletions is expected to be significantly smaller than the number of insertions. Alternatively, if the number dead nodes gets too high, you can invoke a *garbage collection* process, which builds an entirely new search tree containing just the alive nodes.

## Lecture 8: 2-3 Trees

**2-3 Trees:** In the previous lecture, we presented one way to establish balance in a binary search tree, namely through the AVL tree's height-balance condition. Today, we will explore an alternative approach which is achieved by allowing nodes to variable "widths." In particular, we will allow internal nodes to have either two or three children, called *2-nodes* and *3-nodes*, respectively (see Fig. 42(a) and (b)). When a node has three children, it stores two keys. Given the two key values $b$ and $d$, the three subtrees $A$, $C$, and $E$ must satisfy the *generalized inorder property* that

$$A < b < C < d < E.$$

(That is, all the keys in subtree $A$ are smaller than $b$ which is smaller than any of the keys of $C$, and so on.)

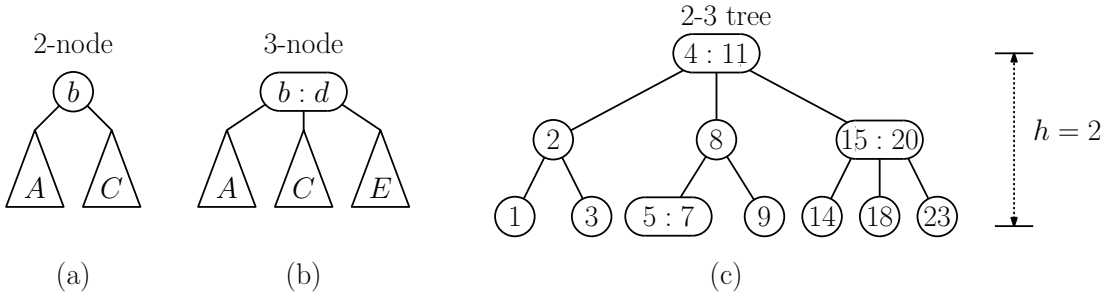



Fig. 42: (a) 2-node, (b) 3-node, and (c) a possible 2-3 tree of height 2.

A *2-3 tree* of height $h \geq -1$ is defined recursively as follows (see Fig. 42(c)). Its keys satisfy the generalized inorder property and it is either:

- empty (i.e., `null`) when $h = -1$, or
- its root is a 2-node, and its two subtrees are each 2-3 trees of heights $h - 1$, or
- its root is a 3-node, and its three subtrees are each 2-3 trees of heights $h - 1$.

By this definition, all the leaves are at the same level. The densest possible 2-3 tree is a complete 3-ary tree of height $h$ and the sparsest possible tree is a complete binary tree of height $h$. This implies the following:

**Theorem:** A 2-3 tree with $n$ nodes has height $O(\log n)$.

It is easy to see how to perform the operation `find(x)` in a 2-3 tree. We apply the usual recursive descent as for a standard binary tree, but whenever we come to a 3-node, we need

to check the relationship between $x$ and the two key values in this node in order to decide which of the three subtrees to visit. The important issues are insertion and deletion, which we discuss next.

For conceptual purposes, it will be convenient to temporarily allow for the existence of "invalid" *1-nodes* and *4-nodes*. A 1-node has one child and no keys. A 4-node has four children and 3 keys. As soon as one of these exceptional nodes comes into existence, we will take action to replace them with proper 2-nodes and 3-nodes.

**Insertion into a 2-3 tree:** The insertion procedure follows the general structure that we have established with prior binary search trees. We first search for the insertion key, and make a note of the last node we visited just before falling out of the tree. Because all leaf nodes are at the same level, we always fall out at the lowest level of the tree. We insert the new key into this leaf node. If the node was a 2-node, it now becomes a 3-node, and we are fine. If it was a 3-node, it now becomes a 4-node, and we need to fix it.

While the initial insertion takes place at a leaf node, we will see that the restructuring process can propagate to internal nodes. So, let us consider how to remedy the problem of a 4-node in a general setting. A 4-node has three keys, say $b$, $d$, and $f$ and four children, say $A$, $C$, $E$, and $G$ (see Fig. 43(a)). In the leaf case, think of $A$, $C$, $E$, and $G$ has empty (`null`) trees.

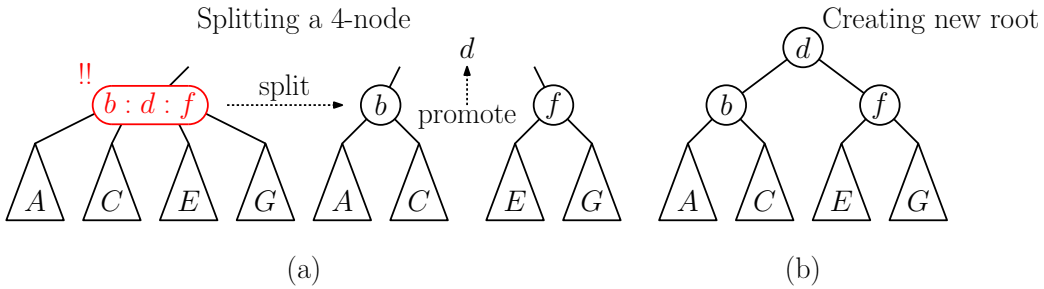



Fig. 43: 2-3 tree insertion: (a) splitting a 4-node into two 2-nodes and (b) creating a new root.

To resolve the problem we *split* this node into two 2-nodes: one for $b$ with $A$ and $C$ as subtrees, and the other for $f$ with $E$ and $G$ as subtrees. We then *promote* the middle key $d$ by inserting it (recursively) into the appropriate location within parent node. What if there is no parent, because the current node is the root? We create a new root node storing $d$ whose two children are $b$ and $f$ (see Fig. 43(b)). It is easy to check that this process preserves the 2-3 tree structural properties, and so we won't bother giving a proof.

In the figure below, we present an example of the result of inserting key 6 into a 2-3 tree. It requires two splits to restore the tree's structure.

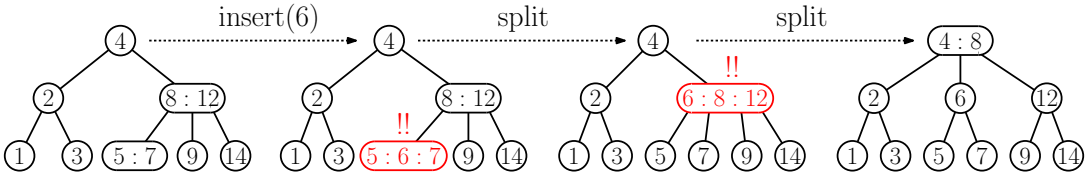



Fig. 44: 2-3 tree insertion involving two splits.

**Deletion from a 2-3 tree:** Consistent with our experience with binary search trees, deletion is more complicated than insertion. The general process follows the usual pattern. First, we

find the key to be deleted. If it does not appear in a leaf node, then we identify a replacement key as the inorder successor. (The inorder predecessor would work equally well.) We copy the replacement key-value pair to replace the deleted key entry, and then we recursively delete the replacement key from its subtree. In this manner, we can always assume that we are deleting a key from a leaf node. So, let us focus on this.

As you might imagine, since insertion resulted in an overfull 4-node being split into two 2-nodes, the process of deletion will involve merging "underfull" nodes. This is indeed the case, but it will also be necessary to consider another restructuring primitive in which keys are taken or "adopted" from a sibling. This adoption process is also called *key rotation*, and I will use both terms interchangeably.

More formally, let us consider the deletion of an arbitrary key from an arbitrary node in the tree. If this is a 3-node, then the deletion results in a 2-node, which is fine. However, if this is a 2-node, the deletion results in an illegal 1-node, which has one subtree and zero keys. We remedy the situation in one of two ways.

**Adoption (Key Rotation):** Consider the left and right siblings of this node (if they exist). If either sibling is a 3-node, then it gives up an appropriate key (and subtree) to convert us back to 2-node. The tree is now properly structured.

Suppose, for the sake of illustration that the current node is an underfull 1-node, and it has a right sibling that is a 3-node (see Fig. 45(a)). Then, we adopt the leftmost key and subtree from this sibling, resulting in two 2-nodes. (A convenient mnemonic is the equation $1 + 3 = 2 + 2$.)

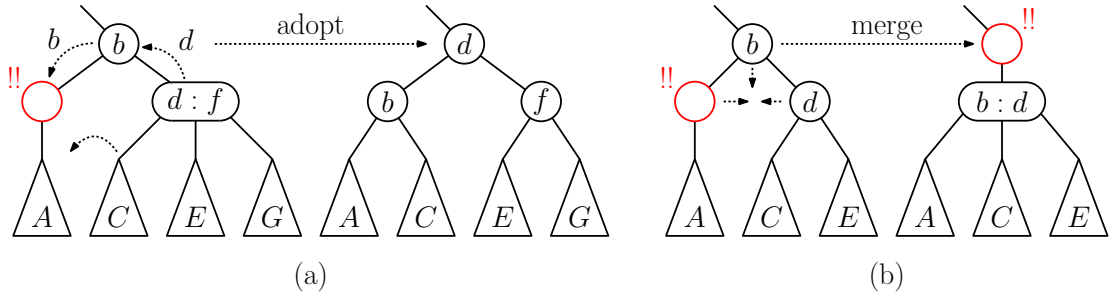



Fig. 45: 2-3 tree deletion: (a) adopting a key/subtree from a sibling and (b) merging two nodes.

**Merging:** On the other hand, if neither sibling can offer a key, then it follows that at least one sibling is a 2-node. Suppose for the sake of illustration that it is the right sibling (see Fig. 45(b)). In this case, we merge the 1-node with this 2-node to form a 3-node. (A convenient mnemonic is the equation $1 + 2 = 3$.)

But, we need a key to complete this 3-node. We take this key from our parent. If the parent was a 3-node, it now becomes a 2-node, and we are fine. If the parent was a 2-node, it has now become a 1-node (as in the figure), and the restructuring process continues on up the tree. Finally, if we ever arrive at a situation where the 1-node is the root of the tree, we remove this root and make its only child the new root.

An example of the deletion of a key is shown in Fig. 46. In this case, the initial deletion was from a 2-node, which left it as a 1-node. We merged it with its sibling to form a 3-node ($1 + 2 = 3$). This involved demoting the key 6 from the parent, which caused the parent to decrease from a 2-node to a 1-node. Since the parent has a 3-node sibling, we can adopt from

it. (By the way, there is also a sibling which is a 2-node, containing the key 2. Could we have instead merged with this node? The answer is "yes", but it is not in our interest to do this. This is because merging results in more disruptions to ancestors of the tree, whereas a single adoption terminates the restructuring process.)

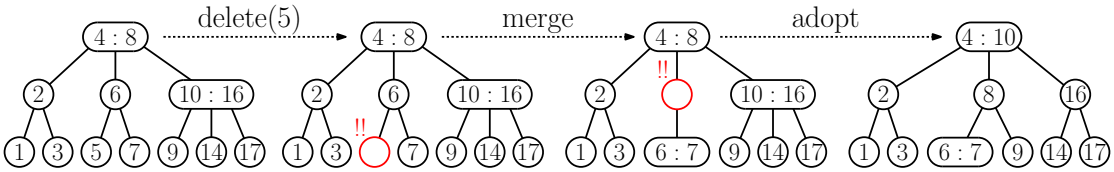



Fig. 46: 2-3 tree deletion involving a merge and an adoption.

In Fig. 47 we illustrate two more examples of deletions from a 2-3 tree. In the upper example, we delete the key 3. Since its only sibling is a 2-node, we cannot perform an adoption, so we merge with the parent. (You might wonder, can't we adopt from our cousin, the (6 : 7) node? The answer is "no." According to the 2-3 deletion algorithm, you can only adopt from your siblings. If we were to allow adoptions from arbitrarily distant cousins, the adoption process would take more than $O(1)$ time.) The parent node is now critical, and so we merge with its parent.

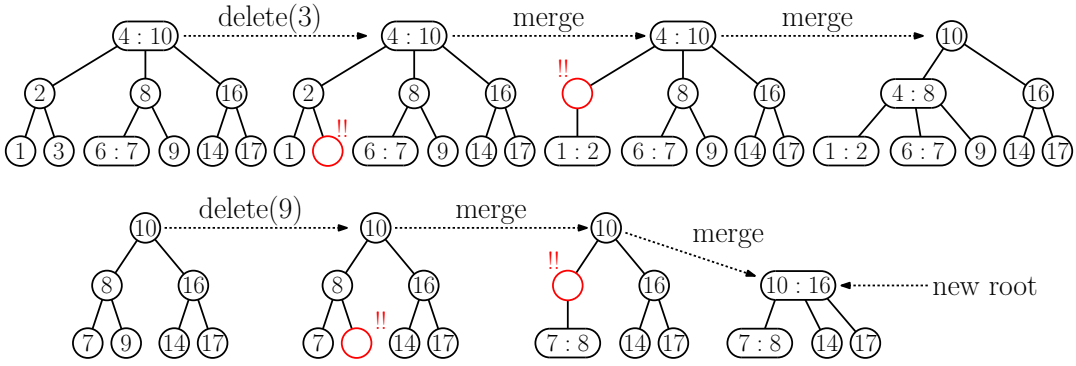



Fig. 47: Two more examples of 2-3 tree deletions.

In the second example in Fig. 47, we illustrate a case where the root node becomes critical. In this case, we remove the old root, and make its only child the new root.

**Implementation?** We will not discuss the 2-3 tree implementation in detail, since we will be presenting a number of related data structures, including red-black trees, AA trees, and B-trees in later lectures. The fact that there are two types of nodes might suggest using two different node classes (e.g., `TwoNode` and `ThreeNode`, which are both subclasses of a generic node class). However, because we often convert between one node type and the other (changing a 2-node into a 3-node and vice versa), it may be preferred to have just one node type that is capable of representing both, and adjusting the number of children/keys dynamically. An example is shown in the following code fragment.

In our examples, we temporarily created a 4-node, which we immediately split into two 2-nodes. This is really just a conceptual trick. In reality, a 4-node would never be generated. Instead, as soon as you see that you are about to create a 4-node, you would immediately invoke a split procedure to remedy the situation.

```
class TwoThreeNode {
    int             nChildren;      // number of children (2 or 3)
    TwoThreeNode    child[3];       // children pointers
    Key             key[2];         // keys
    Value           value[2];       // values
}
```

Another tricky implementation issue is that the current node will need to access its siblings on the left and right sides. To make this operation easier, it may be desirable to add a parent link to our node structure. Alternatively, the various recursive functions can be passed two node pointers as arguments, one to the current node and the other to its parent.

## Lecture 9: Red-black and AA trees

**"A rose by any other name . . . ":** In the previous lecture, we presented the 2-3 tree, which allows nodes of variable widths. In this lecture, we will explore two variations on this idea, red-black trees and AA trees. Both of these structures arise by converting variable width 2-3 nodes into the classical binary tree structure. (Before reading this lecture, please review the material on 2-3 trees from the earlier lecture.)

**Red-Black Trees:** While 2-3 trees provide an optimal $O(n)$ space and $O(\log n)$ time solution to the ordered dictionary problem, they suffer from the shortcoming that they are *not* binary trees. This makes programming these trees a bit messier.

As we saw earlier in the semester, there are ways of representing arbitrary trees as binary trees. This inspires the question, "Can we encode a 2-3 tree as an equivalent binary tree?" Unfortunately, the first-child, next-sibling approach presented earlier in the semester will not work. (Can you see why not? At issue is whether the inorder properties of the tree hold under this representation.)

Here is a simple approach. First, there is no need to modify 2-nodes, since they are already binary-tree nodes. To represent a 3-node as a binary-tree node, we create a two-node combination, as shown in Fig. 48(a) below. Observe that the 2-3 tree ordering ($A < b < C < d < E$) is preserved in the binary version. The 2-3 tree shown in the inset would be encoded in the manner shown in Fig. 48(b).

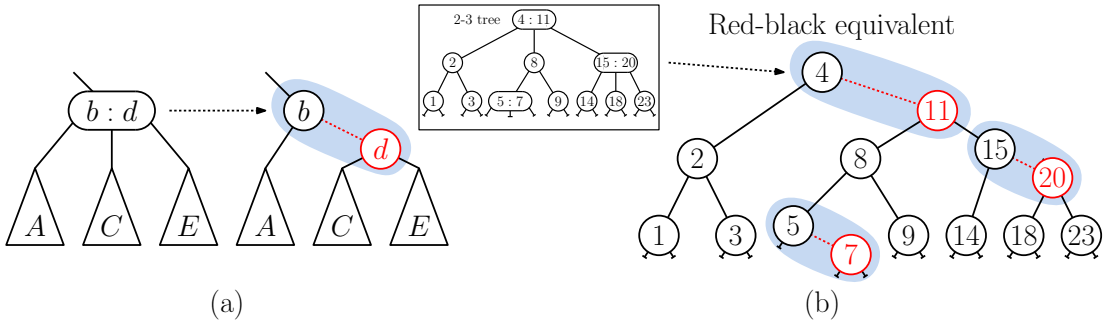



Fig. 48: Encoding a 2-3 tree as an equivalent binary tree.

In the figure, we highlighted node pairs from 3-nodes. To implement this, we will label each of "second nodes" of the 3-nodes as *red* and label all the other nodes as *black*, we obtain a binary tree with both red and black nodes. (Of course, this label can be as simple as a status bit, which indicates whether the node is red or black.) It is easy to see that the resulting binary tree satisfies the following *red-black tree properties*:

(1) Each node is either red or black.

(2) The root is black. (Since it either arises as a 2-node or the top node of a 3-node.)

(3) All `null` pointers are treated *as if* they point to black nodes (a conceptual convenience).

(4) If a node is red, then both its children are black. (Since the children of a 3-node are either 2-nodes or the top of another 3-node pair.)

(5) Every path from a given node to any of its `null` descendants contains the same number of black nodes. (Since all leaf nodes in a 2-3 tree are at the same depth.)

A binary search tree that satisfies these conditions is called a *red-black tree*. It is easy to see that the above encoding of any 2-3 tree to this binary form satisfies all of these properties. (On the other hand, if you just saw this list of properties without having seen the 2-3 tree, it would seem to be very arcane!) Because 2-3 trees have $O(\log n)$ height, the following is an immediate consequence:

**Lemma:** A red-black tree with $n$ nodes has height $O(\log n)$.

**Equivalent? Well, no:** While our 2-3 tree encoding yields red-black trees, not every red-black tree corresponds to our binary encoding of 2-3 trees. There are two issues. First, the red-black conditions do not distinguish between left and right children, so a 3-node could be encoded in two different ways in a red-black tree (see Fig. 49(a)). In addition, the red-black condition allows for the sort of structure in Fig. 49(b), which clearly does not correspond to a node of a 2-3 tree.

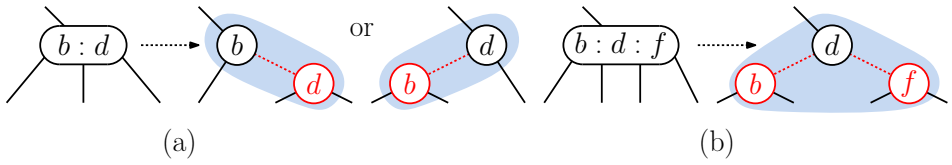



Fig. 49: Color combinations allowed by the red-black tree rules.

So, red-black trees are not equivalent to 2-3 trees, but they are in fact equivalent to a minor variant, called 2-3-4 trees. A 2-3-4 tree is a simple generalization of a 2-3 tree to allow for 4-nodes (that is, having four children and three keys). We will leave as an exercise the fact that split, merge, and adoption (key rotation) utilities for 2-3 trees can be easily generalized to 2-3-4 trees. Red-black trees as defined above correspond 1–1 with 2-3-4 trees. Red-black trees are important as the basis of `TreeMap` class in the `java.util` package.

While red-black trees have a reputation as being the fastest of the balanced binary-tree data structures, they are a bit messy to code, owing to the various cases that can arise. In this lecture, we will study a simplified variant of red-black trees, called AA trees.

**AA trees (red-black trees simplified):** In an effort to simplify the complicated cases that arise with the red-black tree, in 1993 Arne Anderson developed a restriction of the red-black tree.

He called his data structure a BB tree (for "Binary B-tree"), but over time the name has evolved into AA trees, named for the inventor (and to avoid confusion with another popular but unrelated data structure called a BB[$\alpha$] tree).

Anderson's idea was to allow the conversion described above between 2-3 trees and red-black trees, and to rule out the alternative forms shown in Fig. 49. The additional rule that enforces this is:

(6) Each red node can arise only as the right child of a black node.

The edge between a red node and its black parent is called a *red edge*, and is shown as a dashed red edge in our figures. Note that, while AA trees are simpler to code, experiments show that are a bit slower than red-black trees in practice.

Arne-Anderson was focused on making the code for the AA tree as simple as possible. To help, the AA tree has the following two noteworthy features:

**No `null` pointers:** Instead, we create a special *sentinel node*, called `nil` (see Fig. 50(a)), and every `null` pointer is replaced with a pointer to `nil`. This node is declared to be black and `nil.left == nil` and `nil.right == nil` While a tree may have many `null` pointers, there is only one `nil` node allocated, with potentially many incoming pointers. Why do this? This simplifies the code because the left and right children always refer to actual nodes. For example, for any node `p`, we could write `p.right.right.right.right` without worrying about our program aborting due to dereferencing a `null` pointer.

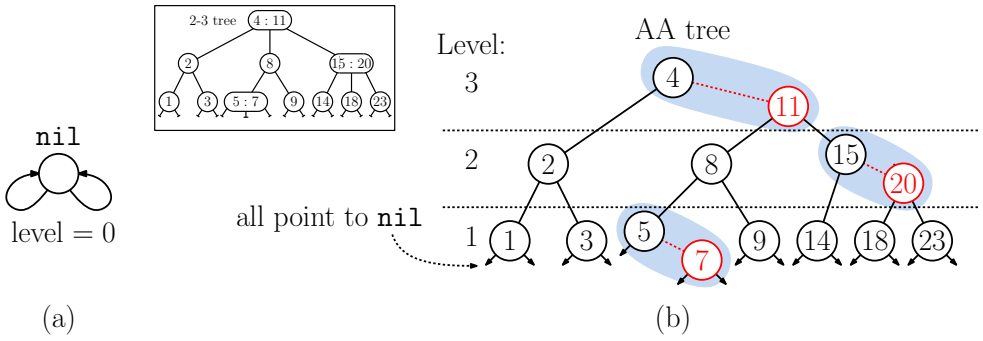



Fig. 50: AA trees: (a) the `nil` sentinel node, (b) the AA tree for a 2-3 tree.

**Levels, not colors:** Rather than using colors to distinguish node types, each node `p` stores a *level number*, denoted `p.level` (see Fig. 50(b)). Intuitively, the level number encodes the level of the associated node in the 2-3 tree. Formally, `nil` node is at level zero.[6] We can use levels to define colors, by the following rule:

A node is *red* if and only if it's at the *same level as its parent*. (The root is never red)

When drawing AA trees, we will draw broken horizontal lines to indicate where levels change. How do we know a node's color? We declare a node `p` to be *red* if `p` and its parent are both on the same level. **When we draw our figures, we will include the colors to emphasize this (see Fig. 50(b)), but the colors are not actually stored in the tree.**

[6] You might protest with this choice. We have been very careful so far in defining the height of an empty tree as $-1$ and `nil` is basically playing the same role as an empty tree. Here I am just following Arne Anderson's convention. If you prefer to set `nil`'s level to $-1$, feel free. Then the other levels will start from 0.

**AA tree operations:** Since an AA tree is essentially a binary search tree, the `find` operation is exactly the same as for any binary search tree. Insertions and deletions are performed in essentially the same way as for AVL trees: first the key is inserted or deleted at the leaf level, and then we retrace the search path back to the root and restructure the tree as we go. As with AVL trees, restructuring essentially involves rotations. For AA trees the two restructuring operations go under the special names `skew` and `split`. They are defined as follows:

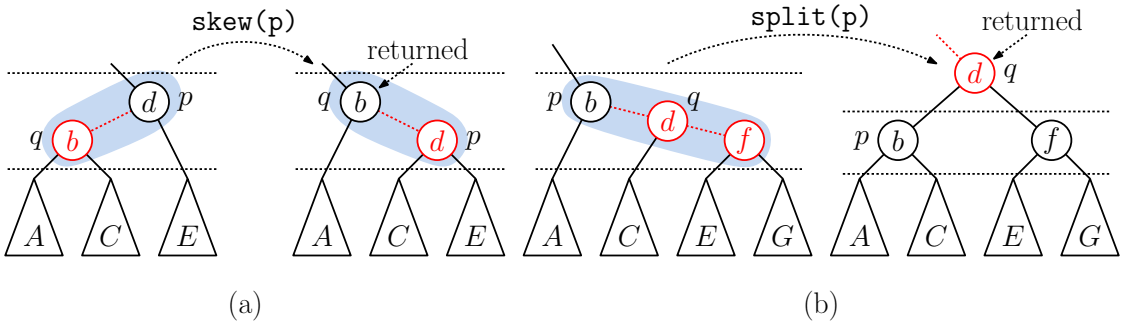



Fig. 51: AA restructuring opertions (a) skew and (b) split. (Afterwards `q` may be red or black.)

`skew(p)`: If `p` is black and has a red left child, rotate so that the red child is now on the right (see Fig. 51(a)). The level of these two nodes are unchanged. Return a pointer to upper node of the resulting subtree.

`split(p)`: If `p` is black and has a chain of two consecutive red nodes to its right (that is, `p.level == p.right.right.level`), split this triple by performing a left rotation at `p` and promoting `p`'s right child, call it `q`, to the next higher level (see Fig. 51(b)). Because $q$ is pushed up a level, it is quite likely to be red now. (The only exception is if $p$ was the root, and now $q$ is the new root.)

In the figure, we have shown `p` as a black node, but in the context of restructuring `p` may be either red or black. As a result, the node `q` that is returned from the operations may either be red or black. The implementation of these two operations is shown in the code block below.

**AA-tree insertion:** As mentioned above, we insert a node just as for a standard binary-search tree and then work back up the tree restructuring as we go. What sort of restructuring is needed? Recall first that (following the policies of 2-3 trees) all leaves should be at the same level of the tree. To achieve this, when the new node is inserted, we assign it the same level number as its parent. This is equivalent to saying that the newly inserted node is red (see Fig. 52(a)).

The first problem might arise is that this newly inserted red node is a left child, which is not allowed (see Fig. 52(b)). Letting `p` denote the node's parent, this is easily remedied by performing `skew(p)` (see Fig. 52(c)). Let `q` be the pointer to the resulting subtree.

Next, it might be that `p` already had a right child that was red, and the skew could have resulted in a right-right chain starting from `q`. (This is equivalent to having a 4-node in a 2-3 tree.) We remedy this by invoking the split operation on `q` (see Fig. 52(d)). Note that the split operation moves the middle node of the chain up to the next level of the tree. The problems that we just experienced may occur with this promoted node, and so the skewing/splitting process generally propagates up the tree to the root.

```
AANode skew(AANode p) {
    if (p == nil) return p;
    if (p.left.level == p.level) {          // red node to our left?
        AANode q = p.left;                  // do a right rotation at p
        p.left = q.right;
        q.right = p;
        return q;                           // return pointer to new upper node
    }
    else return p;                          // else, no change needed
}

AANode split(AANode p) {
    if (p == nil) return p;
    if (p.right.right.level == p.level) {   // right-right red chain?
        AANode q = p.right;                 // do a left rotation at p
        p.right = q.left;
        q.left = p;
        q.level += 1;                       // promote q to next higher level
        return q;                           // return pointer to new upper node
    }
    else return p;                          // else, no change needed
}
```

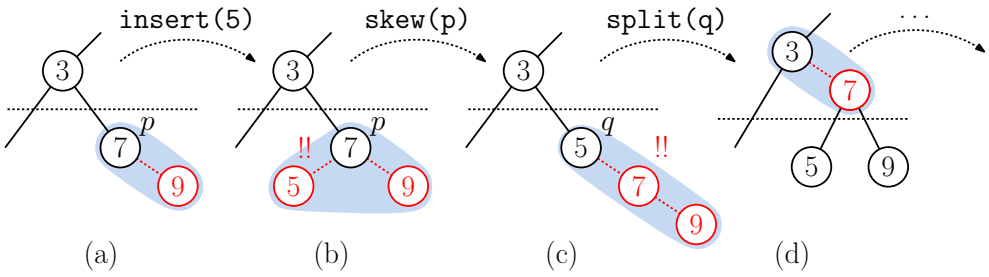



Fig. 52: AA insertion: (a) Initial tree, (b) after insertion, (c) after skewing, (d) after splitting.

The insertion function is provided in the following code block. Observe that (as with the AVL tree) the function is almost the same as the standard (unbalanced) binary tree insertion except for the final rebalancing step, which is performed by the call "`return split(skew(p))`". (This simplicity is the principle appeal of AA trees over traditional red-black trees.)

AA Tree Insertion

```
AANode insert(Key x, Value v, AANode p) {
    if (p == nil)                           // fell out of the tree?
        p = new AANode(x, v, 1, nil, nil);  // ... create a new leaf node here
    else if (x < p.key)                     // x is smaller?
        p.left = insert(x, v, p.left);      // ...insert left
    else if (x > p.key)                     // x is larger?
        p.right = insert(x, v, p.right);    // ...insert right
    else
        throw DuplicateKeyException;        // duplicate key!
    return split(skew(p));                  // restructure and return result
}
```

An example of insertion is shown in Fig. 53. (See the lecture on 2-3 trees for the analogous process.)

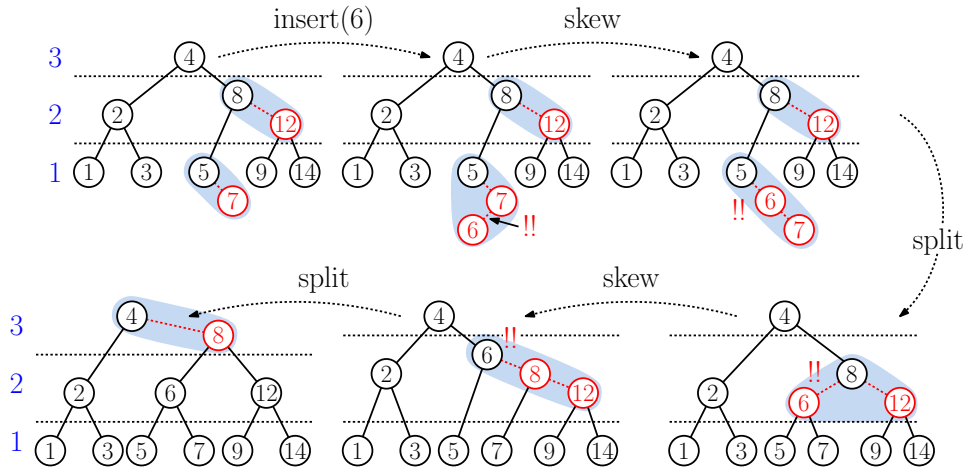



Fig. 53: Example of AA-tree insertion. (Remember, a node is red if it is at the same level as its parent.)

**AA-tree deletion:** As usual deletion is more complex than insertion. If this is not a leaf node, we find a suitable replacement node. (This will either be the inorder predecessor or inorder successor, depending on the tree's structure.) We copy the contents of the replacement node to the deleted node and then we proceed to delete the replacement. After deleting the replacement node (which must be a leaf), we retrace the search path towards the root and restructure as we go.

Before discussing deletion, let's first consider a useful utility function. In the process of deletion, a node can lose one of its children. As a result, we may need to decrease this node's level in the tree. To assist in this process we define two functions. The first, called `updateLevel(p)`, updates the level of a node `p` based on the levels of its children. Every node has at least one black child, and therefore, the ideal level of any node is one more than

the minimum level of its two children. If we discover that `p`'s current level is higher than this ideal value, we set it's level to the proper value. If `p`'s right child is a red node (that is, `p.right.level == p.level` prior to the adjustment), then the level of `p.right` needs to be decreased as well. This is shown in the code block below.

AA-Tree update level utility

```
void updateLevel(AANode p) {                    // update p's level
    int idealLevel = 1 + min(p.left.level, p.right.level);
    if (p.level > idealLevel) {                 // is p's level too high?
        p.level = idealLevel;                   // decrease its level
        if (p.right.level > idealLevel)         // is p's right child red?
            p.right.level = idealLevel;         // ...pull it down as well
    }
}
```

When the restructuring process arrives at a node `p`, we first fix its level using `updateLevel(p)`. Next we need to skew to make sure that any red children are to its right. Deletion is complicated in that we may generally need to perform up to three skew operations to achieve this: one on `p`, one on `p.right`, and one on `p.right.right` (see Fig. 54). After this, `p` may generally be at the top of a right-leaning chain consisting of `p` followed by four red nodes. To remedy this, we perform two splits, one at `p`, and the other to its right-right grandchild, which becomes its right child after the first split (see Fig. 54). Whew! These splits may not be needed, but remember that the split function only modifies the tree if needed. The restructuring function, called `fixAfterDelete`, is presented in the following code fragment. As an exercise, you might draw the equivalent 2-3 tree both before and after the deletion. Note that tree you get by following the 2-3 tree deletion algorithm may not agree with the tree you get by following the AA-tree deletion algorithm

AA-Tree Deletion Utility

```
AANode fixAfterDelete(AANode p) {
    updateLevel(p);                             // update p's level
    p = skew(p);                                // skew p
    p.right = skew(p.right);                    // ...and p's right child
    p.right.right = skew(p.right.right);        // ...and p's right-right grandchild
    p = split(p);                               // split p
    p.right = split(p.right);                   // ...and p's (new) right child
    return p;
}
```

**Tracing an Example:** Let's trace through the steps taken in the deletion process of Fig. 54. After deleting the leaf node 1, we return to node 2. We first invoke `updateLevel(2)`. Because its left child (now `nil`) is at level 0, this pulls 2 down to level 1. Next, we apply the three skews and two skews at the subtree rooted at 2, but everything is fine in this subtree, and none of the rotations are triggered. So, we return from 2 up to node 4.

Now, 4's left child (2) is at level 1, which pulls 4 down to level 2, and in the process, its right child (13) is pulled down as well. Now we have a mess consisting of nodes 4, 6, 9, 13, and 16 all at level 2. The node `p` points to the top node (4). First, we invoke `skew(p)` (at 4), but since 4's left child is a level lower, this does nothing. Next, we invoke `skew(p.right)` (at 13). This performs a right rotation at 13, which pulls 6 up as 4's new right child, and

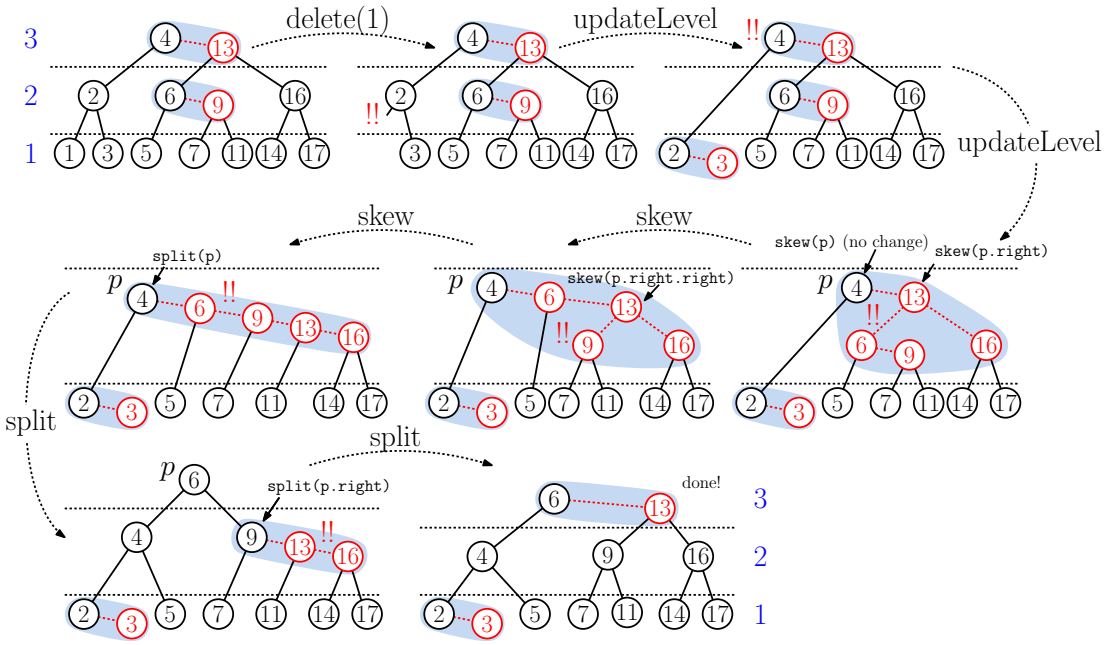



Fig. 54: Example of AA-tree deletion. (Remember, a node is red if it is at the same level as its parent.)

now 13 is `p.right.right` and its left child is 9. Next, we invoke `skew(p.right.right)` (13). This performs a right rotation at 13, which pulls 9 up as 6's new right child, and now 13 is `p.right.right.right`.

Now that we are done with the skews, we next invoke `split(p)` (4). This performs a left rotation at 4 and pushes 6 up to level 3. The split operation returns 6 which is assigned to `p`. We next invoke `split(p.right)` (9). This performs a left rotation at 9 and pushes 13 up to level 3, as the new right child of 6.

We finally return the current value of `p` (6) to the calling procedure. However, the calling procedure was the initial call, namely `root = delete(1, root)`. Therefore, node 6 is assigned as the new root of the tree, and we are done.

**Deletion Code:** Finally, we can present the full deletion code. It looks almost the same as the deletion code for the standard binary search tree, but after deleting the leaf node, we invoke `fixAfterDelete` to restructure the tree. We will omit the (messy) details showing that after this restructuring, the tree is in valid AA-form. (We refer you to Anderson's original paper.)

**Analysis:** All of these algorithms take $O(1)$ time per level of the tree, which implies that the running time of all the dictionary operations is $O(h)$ where $h$ is the height of the tree. As we saw abow, the tree's height is $O(\log n)$ height, which implies that all the dictionary operations run in $O(\log n)$ time.

## Lecture 10: Point quadtrees and kd-trees

**Geometric Data Structures:** In today's lecture we move in a new direction by covering a number of data structures designed for storing multi-dimensional geometric data. Geometric data

```
AANode delete(Key x, AANode p) {
    if (p == nil)                                  // fell out of tree?
        throw KeyNotFoundException;                // ...error - no such key
    else {
        if (x < p.key)                             // look in left subtree
            p.left = delete(x, p.left);
        else if (x > p.key)                        // look in right subtree
            p.right = delete(x, p.right);
        else {                                     // found it!
            if (p.left == nil && p.right == nil)// leaf node?
                return nil;                        // just unlink the node
            else if (p.left == nil) {              // no left child?
                AANode r = inorderSuccessor(p); // get replacement from right
                p.copyContentsFrom(r);             // copy replacement contents here
                p.right = delete(r.key, p.right); // delete replacement
            }
            else {                                 // no right child?
                AANode r = inorderPredecessor(p); // get replacement from left
                p.copyContentsFrom(r);             // copy replacement contents here
                p.left = delete(r.key, p.left); // delete replacement
            }
        }
        return fixAfterDelete(p);                  // fix structure after deletion
    }
}
```

structures are fundamental to the efficient processing of data sets arising from myriad applications, including spatial databases, automated cartography (maps) and navigation, computer graphics, robotics and motion planning, solid modeling and industrial engineering, particle and fluid dynamics, molecular dynamics and drug design in computational biology, machine learning, image processing and pattern recognition, computer vision.

Fundamentally, our objective is to store a large datasets consisting of geometric objects (e.g., points, lines and line segments, simple shapes (such as balls, rectangles, triangles), and complex shapes such as surface meshes) in order to answer queries on these data sets efficiently. While some of our explorations will involve delving into geometry and linear algebra, fortunately most of what we will cover assumes no deep knowledge of geometric objects or their representations. Given a collection of geometric objects, there are numerous types of queries that we may wish to answer.

**Nearest-Neighbor Searching:** Store a set of points so that qiven a query point $q$, it is possible to find the closest point of the set (or generally the closest $k$ objects) to the query point (see Fig. 55(a)).

**Range Searching:** Store a set of points so that given a query region $R$ (e.g., a rectangle or circle), it is possible to report (or count) all the points of the set that lie inside this region (see Fig. 55(b)).

**Point location:** Store the subdivision of space into disjoint regions (e.g., the subdivision of the globe into countries) so that given a query point $q$, it is possible determine the region of the subdivision containing this point efficiently (see Fig. 55(c)).

**Intersection Searching:** Store a collection of geometric objects (e.g., rectangles), so that given a query consisting of an object $R$ of this same type, it is possible to report (or count) all of the objects of the set that intersect the query object (see Fig. 55(d)).

**Ray Shooting:** Store a collection of object so that given any query ray, it is possible to determine whether the ray hits any object of the set, and if so which object does it hit first.

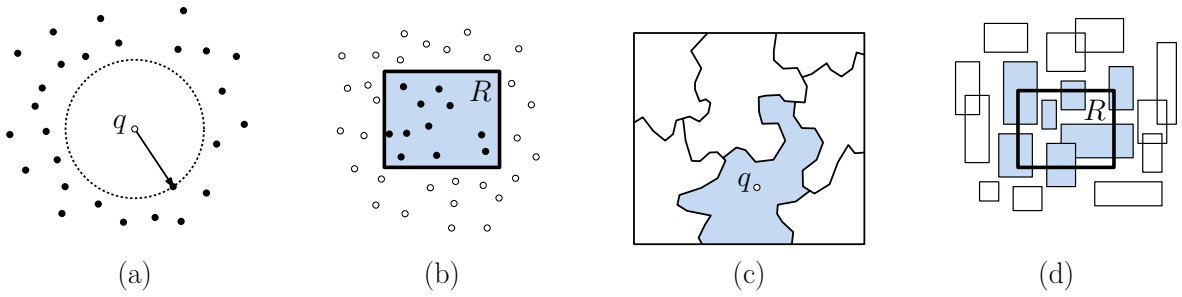



Fig. 55: Common geometric queries: (a) nearest-neighbor searching, (b) range searching, (c) point location, (d) intersection searching.

In all cases, you should imagine the size $n$ of the set is huge, consisting for example of millions of objects, and the objective is to answer the query in time that is significantly smaller than $n$, ideally $O(\log n)$. We shall see that it is not always possible to achieve efficient query times with storage that grows linearly with $n$. In such instances, we would like the storage to slowly, for example, $O(n \log n)$. As with 1-dimensional data structures, it will also be desirable to provide dynamic updates, allowing for the insertion and deletion of objects.

**No Total Ordering:** While we shall see that many of the ideas that we applied in the design of 1-dimensional data structures can be adapted to the design of multi-dimensional data structure, there is one fundamental challenge that we will face. Almost all 1-dimensional data structures exploit the fact that the data are drawn from a total order. The existence of such a total ordering is critical to all the tree-based search structures we studied as well as skip lists.

The only exception to this is hashing. But hashing is applicable only when we are searching for exact matches. In typical geometric queries (as all the ones described above) exact matching does not apply. Instead we are interested in notions such as "close to" or "contained within" or "overlapping with," none of which are amenable to hashing.

**Point representations:** Let's first say a bit about representation and notation. We will assume that each point $p_i$ is expressed as a $d$-element vector, that is $p_i = (p_{i,1}, \ldots, p_{i,d})$. To simplify our presentation, we will usually describe our data structures in a 2-dimensional context, but the generalization to higher dimensions will be straightforward. For this reason, we may sometimes refer to a point's in terms of its $(x, y)$-coordinates, for example, $p = (p_x, p_y)$, rather than $p = (p_1, p_2)$.

While mathematicians label indices starting with 1, programming languages like Java prefer to index starting with 0. Therefore, in Java, each point `p` is represented as a $d$-element vector:

```
float[][] p = new float[n][d]; // array of n points, each a d-element vector
```

In this example, the points are `p[0]`, `p[1]`, `p[n-1]`, and the coordinates of the $i$th point are given by `p[i][0]`, `p[i][1]`, `p[i][d-1]`.

A better approach would be to define a class that represents a point object. An example of a simple `Point` object can be found in the code block below. We will assume this in our examples. Java defines a 2-dimensional point object, called `Point2d`.

Simple Point class

```
public class Point {
    private float[] coord; // coordinate storage

    public Point(int dim) {  /* construct a zero point */ }

    public int getDim() { return coord.length; }
    public float get(int i) { return coord[i]; }

    public void set(int i, float x) { coord[i] = x; }

    public boolean equals(Point other) { /* compare with another point */ }
    public float distanceTo(Point other) { /* compute distance to another point */ }

    public String toString() { /* convert to string */  }
}
```

Now, your point set could be defined as an array of points, for example, `Point[] pointSet = new Point[n]`. Note that although we should use `pt.get(i)` to get the $i$th coordinate of a point `pt`, we will often be lazy in our code samples, and just write `pt[i]` instead.

**Point quadtree:** Let us first consider a natural way of generalizing unbalanced binary trees in the 1-dimensional case to a $d$-dimensional context. Suppose that we wish to store a set $P = \{p_1, \ldots, p_n\}$ of $n$ points in $d$-dimensional space. In binary trees, each point naturally splits the real line in two. In two dimensions if we run a vertical and horizontal line through the point, it naturally subdivides the plane into four *quadrants* about this point. (In general $d$-dimensional space, we consider $d$ axis-parallel hyperplanes passing through the point. These subdivide space into $2^d$ *orthants*.)

To simplify the presentation, let us assume that we are working in 2-dimensional space. The resulting data structure is called a *point quadtree*. (In dimension three, the corresponding structure is naturally called an *octtree*. As the dimension grows, it is too complicated to figure out the proper term for the number of children, and so the term *quadtree* is often used in arbitrary dimensions, even though the outdegree of each node is $2^d$, not four.)

Each node has four (possibly null) children, corresponding to the four quadrants defined by the 4-way subdivision. We label these according to the compass directions, as `NW`, `NE`, `SW`, and `SE`. In terms of implementation, you can think of assigning these the values 0, 1, 2, 3, and use them as indices to a 4-element array of children pointers.

As with standard (unbalanced) binary trees, points are inserted one by one. We descend through the tree structure in a natural way. For example, we compare the newly inserted point's $x$ and $y$ coordinates to those of the root. If the $x$ is larger and the $y$ is smaller, we recurse on the `SE` child. The insertion of each point results in a subdivision of a rectangular region into four smaller rectangles. Consider the insertion of the following points (see Fig. 56):

$$(35, 40), (50, 10), (60, 75), (80, 65), (85, 15), (5, 45), (25, 35), (90, 5).$$

The final subdivision and tree structure are shown in Fig. 57.

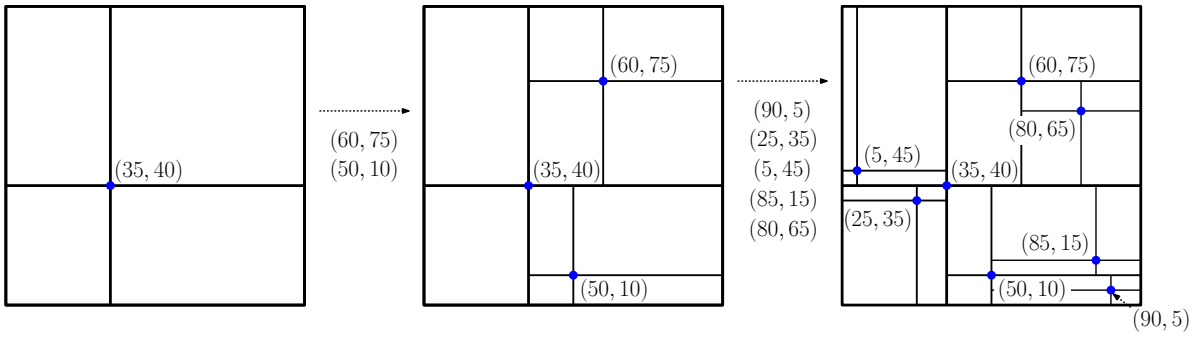



Fig. 56: Point quadtree subdivision.

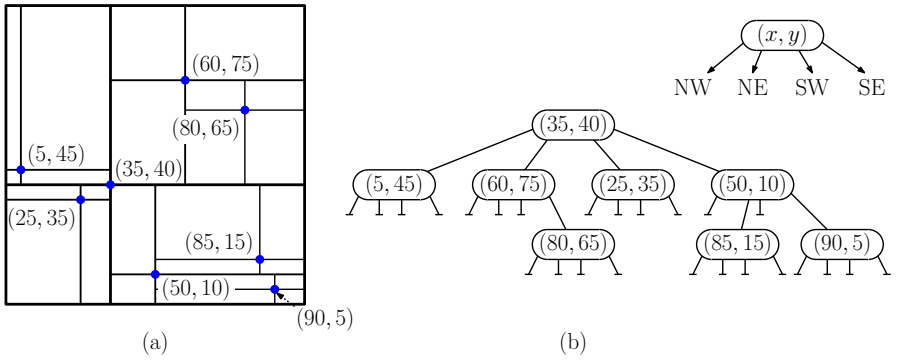



Fig. 57: Point quadtree.

Each node in the tree is naturally associated with a rectangular region of space, which we call its *cell*. Note that some rectangles are special in that they extend to infinity. Since semi-infinite rectangles sometimes bother people, it is not uncommon to assume that everything is contained within one large bounding rectangle, which may be provided by the user when the tree is first constructed.

We will not discuss algorithms for the point quad-tree in detail. Instead, we will defer this discussion to point kd-trees, and simply note that for each operation on a kd-tree, there is a similar algorithm for quadtrees.

**Point kd-tree:** As observed above, point quadtrees can be generalized to higher dimensions, the number of children grows exponentially in the dimension, as $2^d$. For example, if you are working in 20-dimensional space, every node has $2^{20}$, or roughly a million children! Clearly, the simple quadtree idea is not scalable to very high dimensions. Next, we describe an alternative data structure, that always results in a binary tree.

As in the case of a quadtree, the cell associated with each node is an axis-aligned (hyper-)rectangle. When a new point is inserted into some leaf node's cell, we split the cell by a horizontal or vertical *splitting line*, which passes through this point (or generally a $(d-1)$-dimensional axis-aligned hyperplane). The split is specified by its *cutting dimension*, `cutDim`, which can be represented as an integer from 0 to $d-1$ and its *cutting value*. As with quadtrees, the cut is made through the point, so the cutting value is `p.point[cutDim]`. By convention, points whose coordinate value is *strictly smaller* than the cutting value (`pt[cutDim] < point[cutDim]`) are stored in the left subtree and those with values *greater than or equal* are in the right subtree.

```
class KDNode {                              // node in a kd-tree
    Point point                             // splitting point
    int cutDim                              // cutting dimension (optional)
    KDNode left, right                      // children

    KDNode(Point point, int cutDim) {       // constructor
        this.point = point
        this.cutDim = cutDim
        left = right = null
    }

    boolean inLeftSubtree(Point pt) {       // is pt in left subtree?
        return pt[cutDim] < point[cutDim]
    }
}
```

The resulting data structure is called a *point kd-tree*. Actually, this is a bit of a misnomer. The data structure was named by its inventor Jon Bentley to be a *2-d tree* in the plane, a *3-d tree* in 3-space, and a *k-d tree* in dimension $k$. However, over time the name "kd-tree" became commonly used irrespective of dimension. Thus it is common to say a "kd-tree in dimension 3" rather than a "3-d tree".

**How to Cut Space?** There are a number of ways to select the cutting dimension. The most common is just to alternate (or generally cycle) among the possible axes at each new level of the tree. For example, at the root node we cut orthogonal to the $x$-axis (or 0th coordinate), for its children we cut orthogonal to $y$ (or 1st coordinate), for the grandchildren we cut again along $x$, and so on. In higher dimensions, we cycle through the various dimensions. An example is shown in Fig. 58. Note that when this is done, we do not need to explicitly store the cutting dimension in each node, since it can be computed on the fly as we traverse the tree.

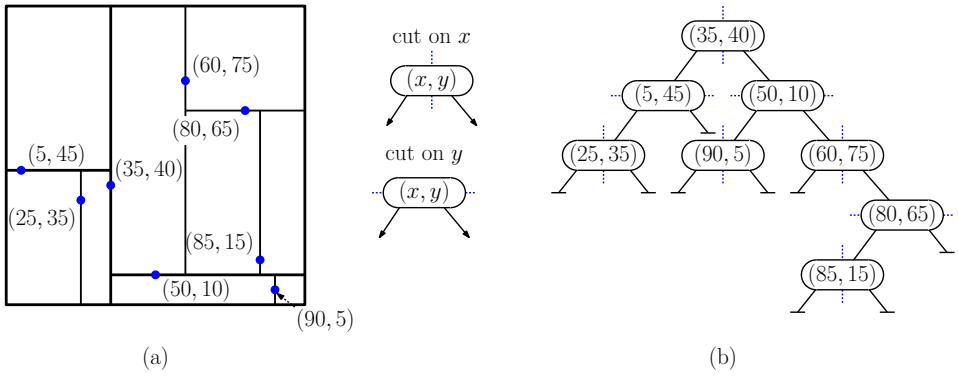



Fig. 58: Point kd-tree decomposition.

We will assume this cyclic method of choosing the cutting dimension in our examples, but there are better ways. For example, Bentley proposed that the cutting dimension should be chosen based on the point distribution. The cut should be made perpendicular to the longest side of the point set's bounding box. He also suggested that the cut should be made through the median point of the set. This tends to produce nice "fat" cells and has height $O(\log n)$.

As with unbalanced binary search trees, it is possible to prove that if keys are inserted in

random order, then the expected height of the tree is $O(\log n)$, where $n$ is the number of points in the tree.

**Insertion into kd-trees:** Insertion operates as it would for a regular binary search tree. We descend the tree until falling out, and then we create a node containing the point and assign its cutting dimension by whatever policy is used by the tree. The principal utility function is presented in the following code block. The function takes three arguments, the point `pt` being inserted, the current node `p`, and the cutting dimension of the newly created node. The initial call is `root = insert(pt, root, 0)`.

kd-tree Insertion

```
KDNode insert(Point pt, KDNode p, int cd) { // insert pt in subtree p with cutDim cd
    if (p == null) {                                    // fell out of tree
        p = new KDNode(pt, cd)                          // create new leaf
    } else if (p.point.equals(pt)) {
        throw Exception("Error - duplicate point")
    } else if (p.inLeftSubtree(pt)) {                   // insert into left subtree
        p.left = insert(pt, p.left, (cd + 1) % dim)
    } else {                                            // insert into right subtree
        p.right = insert(pt, p.right, (cd + 1) % dim)
    }
    return p
}
```

Observe that whenever we make a recursive call, we advance the cutting dimension by one (`cd + 1`) but we wrap around by modding this with the dimension of the space. An example is shown in Fig. 59, where we insert the point $(50, 90)$ into the kd-tree of Fig. 58. We descend the tree until we fall out on the left subtree of node $(60, 75)$. We create a new node at this point, and the cutting dimension cycles from the parent's $x$-cutting dimension (`cutDim = 0`) to a $y$-cutting dimension (`cutDim = 1`).

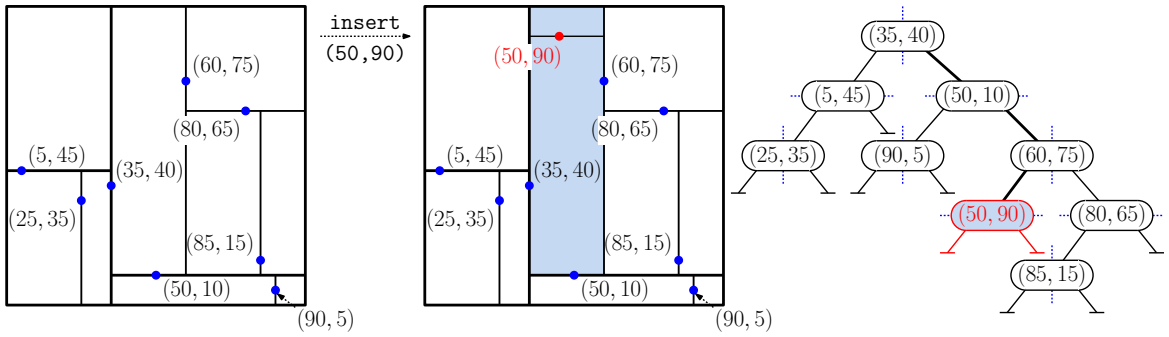



Fig. 59: Inserting $(50, 90)$ into the point kd-tree of Fig. 58.

**Analysis:** The space needed by the kd-tree to store $n$ points in $d$-dimensional space is $O(n)$, which is optimal. (We treat $d$ like a constant, independent of $n$. In fact, it takes $O(dn)$ space to store the coordinates, but since $d$ is a constant, we can ignore the $d$ factor.)

The height analysis of the kd-tree is essentially the same as that of the unbalanced binary tree. If $n$ points are inserted in random order, then the height of the tree will be $O(\log n)$ in expectation. (Below, we discuss deletion. Because we chose replacement nodes in a biased

way, always from the right subtree, it is reasonable that the same systematic bias issues that leads to $O(\sqrt{n})$ tree height over a long series of random insertions and deletions.)

Probably the best way to maintain balance in a kd-tree is to simply rebuild unbalanced subtrees, similar to what we did with *scapegoat trees*. This raises the question of how to build a well-balanced tree for a fixed set of points. Suppose that we use the method of cycling the cutting dimension from level to level of the tree to build a kd-tree for a point set $P$. At the root level, we could choose the splitting point to be the median of $P$ according to $x$-coordinates. Then, after partitioning the set about this point, into say $P_L$ and $P_R$, the splitting value for each set would be the respective medians but according to the $y$-coordinates. By doing this, we guarantee that the number of points in each subtree is essentially half that of its parent, and this implies that the overall tree height is $O(\log n)$.

By the way, this raises an interesting computational question. We know that it is possible to build a 1-dimensional tree from a sorted point set in $O(n \log n)$ time, by repeatedly splitting on the median. Can you generalize this to construct a perfectly balanced 2-dimensional kd-tree also in $O(n \log n)$ time. The tricky issue is that sorting on $x$ does not help you in finding the $y$-splits, and sorting on $y$ does not help you with the $x$ splits. This is an interesting computational problem to think about. (The answer is that it is possible to build such a tree in $O(n \log n)$ time, but it takes a bit of cleverness. We will leave this as an exercise.)

**Finding Replacements for Deletion:** We will next discuss deletion from kd-trees. As we saw with deletion in standard binary search trees, an issue that will arise when deleting a point from the middle of the tree is what to use in place of this node, that is, the *replacement point*. It is not as simple as selecting the next point in an inorder traversal of the tree, since we need a point that satisfies the necessary geometric conditions.

Suppose that we wish to delete a point in node `p` where `p.cutDim == 0` (that is, vertical cut). An appropriate choice for the replacement point is the point of `p.right` that has the smallest $x$-coordinate. (What do we do if the right child is `null`? We'll come to this later.) Finding such a point is a nice exercise, since it illustrates how programming is done with kd-trees.

Let us derive a procedure `findMin(p, i)` that computes the point in the subtree rooted at node $p$ that has the smallest $i$th coordinate. The procedure operates recursively. When we arrive at a node $p$, if the cutting dimension matches $i$, given that the subtrees are ordered by the $i$th coordinate, we know that the minimum cannot lie in the right subtree. If the left child is non-null, then the desired point is there, and we recursively search this subtree (see Fig. 60(a)). If the left child is null, we return $p$'s associated point (see Fig. 60(b)).

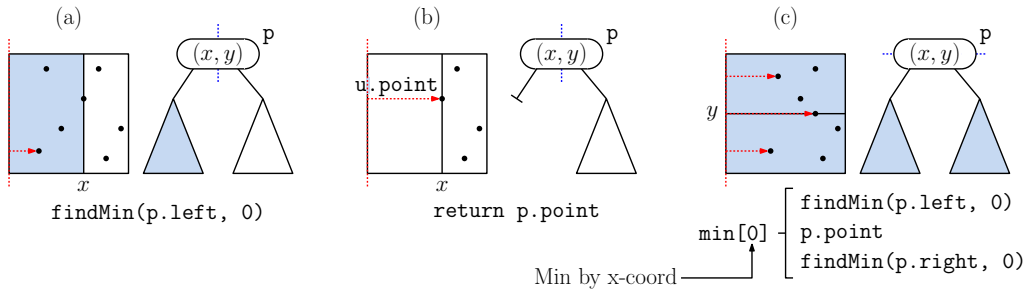



Fig. 60: Cases in `findMin`.

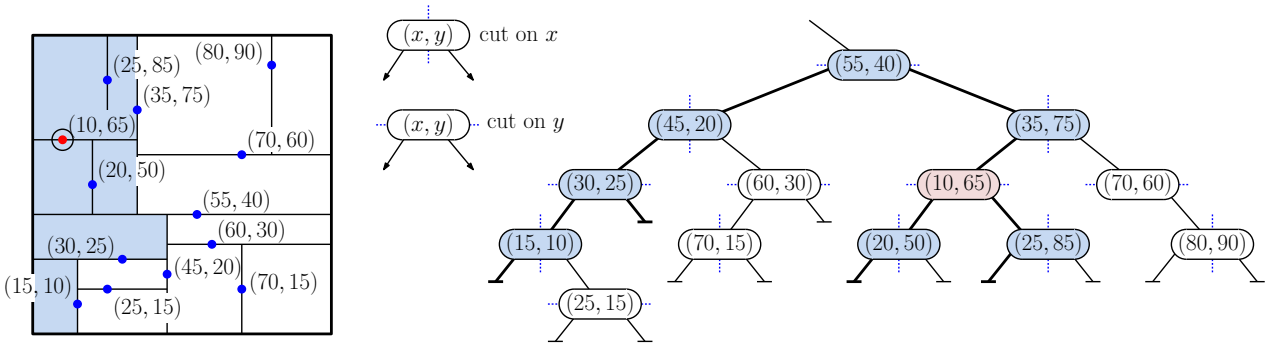
Finally, if the cutting dimension does not equal $i$, then the minimum may either in the left/right subtrees or might be `p.point` itself. We check all three cases and return the

minimum along the $i$th coordinate (see Fig. 60(c)). The code is presented below. Let's imagine that we have utility function `min[i](q1, q2, q3)`, which returns whichever point among `q1`, `q2`, and `q3` that has the smallest `i`-th coordinate. (A null point is never favored over a non-null point.)

Find the minimum point in subtree along $i$th coordinate

```
Point findMin(KDNode p, int i) {                    // get min point along dim i
    if (p == null) return null                      // fell out of tree?
    if (p.cutDim == i) {                            // cutting dimension matches i?
        if (p.left == null)                         // no left child?
            return p.point                          // use this point
        else
            return findMin(p.left, i)               // get min from left subtree
    } else {                                        // it may be in either side
        return min[i](findMin(p.left), p.point, findMin(p.right))
    }
}
```

Fig. 61 presents an example of the execution of this algorithm to find the point with the minimum $x$-coordinate in the subtree rooted at $(55, 40)$. Since this node splits horizontally, we need to visit both of its subtrees to find their minimum $x$ values. (These will be $(15, 10)$ for the left subtree and $(10, 65)$ for the right subtree.) These values are then compared with the point at the root to obtain the overall $x$-minimum point, namely $(10, 65)$. Observe that because the subtrees at $(45, 20)$ and $(35, 75)$ both split on the $x$-coordinate, and we are looking for the point with the minimum $x$-coordinate, we do not need to search their right subtrees. The nodes visited in the search are shaded in blue.




(a) (b)

Fig. 61: Example of `findMin` when $i = 0$ (the $x$-coordinate) on the subtree rooted at $(55, 40)$. The function returns $(10, 65)$.

**Deletion from a kd-tree:** As with insertion, deletion is similar as for unbalanced binary search trees. However, there is an interesting twist here. Recall that in the 1-dimensional case we needed to consider a number of different cases. If the node is a leaf we just delete the node. Otherwise, its deletion would result in a "hole" in the tree. We need to find an appropriate *replacement*. In the 1-dimensional case, we were able to simplify this if the node has a single child (by making this child the new child of our parent). However, this would move the child from an even level to an odd level, or vice versa, and this would violate our assumption that

the cutting dimensions cycle among the coordinates. (Note this might not be an issue if the cutting dimension were selected by some other policy, but to keep things as clean as possible, our deletion procedure will not alter a node's cutting dimension.)

**The Replacement Problem:** Let us assume first that the right subtree is non-empty. Recall that in the 1-dimensional case, the replacement key was taken to be the smallest key from the right child, and after using the replacement to fill the hole, we recursively deleted the replacement. How do we generalize this to the multi-dimensional case? The proper thing to do is to find the point whose coordinate along the current cutting dimension is minimum. Thus, if the cutting dimension is the $x$-axis, say, then the replacement is the point with the smallest $x$-coordinate in the right subtree. We use the `findMin()` function (above) to do this.

On the other hand, what if the right subtree is empty? At first, it might seem that the right thing to do is to select the maximum node from the left subtree. However, there is a subtle trap here. Recall that we maintain the invariant that points whose coordinates are equal to the cutting dimension are stored in the right subtree. If we select the replacement point to be the point with the maximum coordinate from the left subtree, and if there are other points with the same coordinate value in this subtree, then we will have violated our invariant.

As an example, of the problem, consider Fig. 62. Suppose that we delete $(50, 60)$. Since the right subtree is empty, we take the rightmost point from the left subtree. There are two candidates, $(30, 45)$ and $(30, 20)$. Suppose we take $(30, 20)$ as the replacement, which we copy to the deleted node. Next, we recursively delete $(30, 20)$ from the left subtree (see Fig. 62(c)). Observe that the point $(30, 45)$ is the left subtree of the root, but according to our convention, if a node has the same coordinate as the cutting value, it should go in the right subtree, not the left. Thus, the result is *not* a valid kd-tree by our convention.

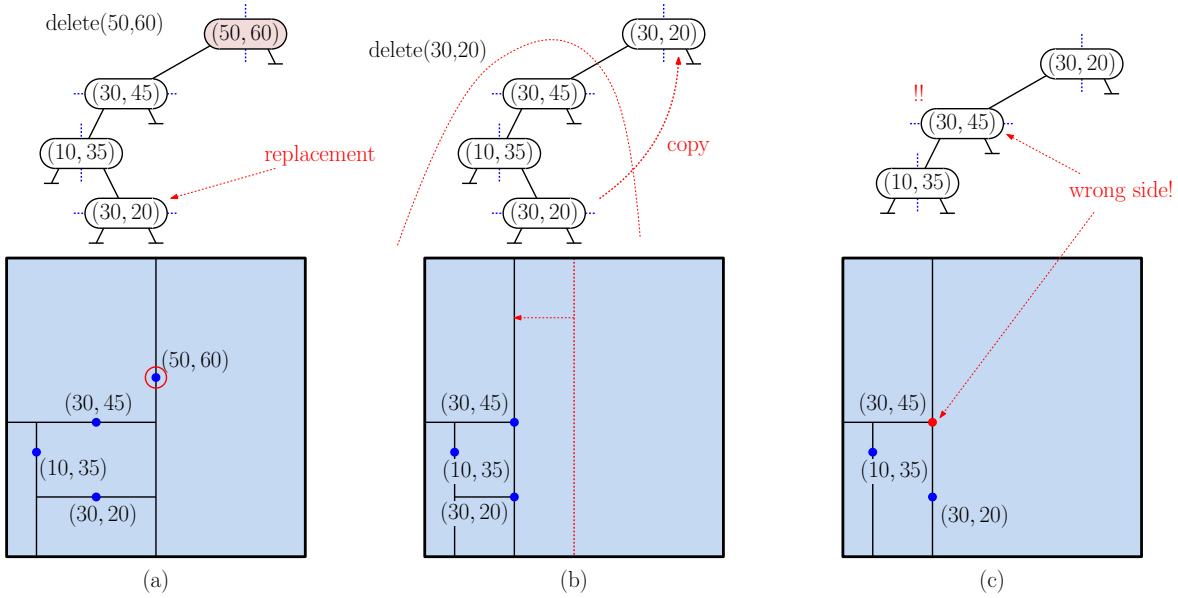



Fig. 62: The problem with deletion.

**Fixing the Replacement Problem:** There is a clever trick for fixing the replacement problem. Rather than selecting the point with the maximum coordinate from the left subtree, we will instead select the point with the *minimum* coordinate. Huh? To make this work, we also

move the left subtree over to become the new right subtree, and the left child pointer is set to `null`. An example is given in Fig. 63. **This is tricky! Be sure you understand why this works.** The deletion helper is given in the code block below.

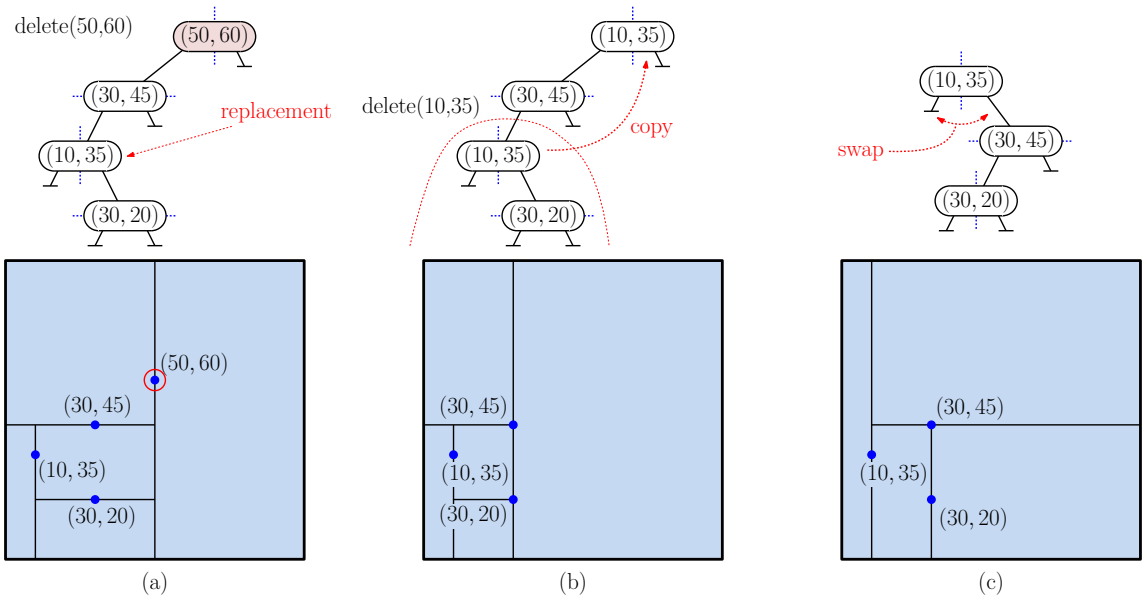



Fig. 63: Fixing the replacement problem.

**Deletion Example:** An example of the operation of this deletion algorithm is presented in Fig. 64. The original objective is to delete the point $(35, 60)$. This is at the root of the tree. Because the cutting dimension is vertical, we search its right subtree to find the point with the minimum $x$-coordinate, which is $(50, 30)$. The point is copied to the root. Note that when this happens, the associated subdivision of space immediately changes. (See the top right of Fig. 64). The picture is a bit wonky, because $(50, 30)$ appears twice in the tree. Once as a vertical splitted at the root and later as a horizontal splitter.

We then recursively delete $(50, 30)$ from the root's right subtree. This recursive call then seeks the node $p$ containing $(50, 30)$. Note that this node has no right child, but unlike standard binary search trees, we cannot simply unlink it from the tree (for the reasons described above). Instead, we observe that its cutting dimension is horizontal, and we search for the point with the minimum $y$-coordinate in $p$'s left subtree, which is $(60, 10)$. We copy $(60, 10)$ to $p$. Again, the subdivision changes. Node that the vertical splitting line through $(70, 20)$, which was blocked by $(50, 30)$ now extends all the way to the top of the cell.

We now recursively delete $(60, 10)$ from $p$'s left subtree. It is a leaf, so it may simply be unlinked from the tree. (We don't see the change in the subdivision, since $(60, 10)$ already exists in the tree.) Finally, when we return to the parent of $(70, 20)$, since this was a left-side deletion, we need to move the subtree rooted at $(70, 20)$ over to the right side of its parent $(60, 10)$. Following this, we return all the way back to the root without any further changes. (Whew!)

```
KDNode delete(Point pt, KDNode p) {
    if (p == null) {                              // fell out of tree?
        throw Exception("Error - Point does not exist");
    } else if (p.point.equals(pt)) {              // found it
        if (p.right != null) {                    // can replace from right
            p.point = findMin(p.right, p.cutDim)  // find and copy replacement
            p.right = delete(p.point, p.right)    // delete from right
        } else if (p.left != null) {              // can replace from left
            p.point = findMin(p.left, p.cutDim)   // find and copy replacement
            p.right = delete(p.point, p.left)     // delete left but move to right!!
            p.left = null                         // left subtree is now empty
        } else {                                  // deleted point in leaf
            p = null                              // remove this leaf
        }
    } else if (p.inLeftSubtree(pt)) {
        p.left = delete(pt, p.left)               // delete from left subtree
    } else {                                      // delete from right subtree
        p.right = delete(pt, p.right)
    }
    return p
}
```

## Lecture 11: Answering Queries with kd-trees

**Recap:** In our previous lecture we introduced kd-trees, a multi-dimensional binary partition tree that is based on axis-aligned splits. We have shown how to perform the operations of insertion and deletion from kd-trees. In this lecture, we will investigate an important geometric query using kd-trees: orthogonal range search queries.

**Range Queries:** Given any point set, a fundamental type of query is called a *range query* or more properly, an *orthogonal range query*. To motivate this sort of query, suppose that you querying a biomedical database with millions of records. Each medical record is encoded as a vector of health statistics, such as height, weight, blood pressure, etc. Each coordinate is the numeric value of some statistic, such as a person's height, weight, blood pressure, etc. Suppose that you want to anwwer queries of the form "how many patients whose range 70–80 kilograms, heights in the range 160–170 centimeters, etc." This is equivalent to finding the number of points in the database that lie within an axis-orthogonal rectangle, defined by the intersection of these intervals (see Fig. 65).

More formally, given a set $S$ of points in $d$-dimensional real space, $\mathbb{R}^d$, we wish to store these points in a kd-tree so that, given a query consisting of an axis-aligned rectangle, denoted $R$, we can efficiently count or report the points of $S$ lying within $R$. Listing all the points lying in the range is called a *range reporting query*, and counting all the points in the range is called a *range counting query*. The solutions for the two problems are often similar, but some tricks can be employed when counting, that do not apply when reporting.

**A Rectangle Class:** Before we get into a description of how to answer orthogonal range queries with the kd-tree tree, let us first define a simple class, called `Rect` for storing a multi-dimensional rectangle, or *hyperrectangle* for short. The private data consists of two points `low` and `high`. A point $q$ lies within the rectangle if $\text{low}[i] \leq q[i] \leq \text{high}[i]$, for $0 \leq i \leq d-1$


Fig. 64: Deletion from a kd-tree.

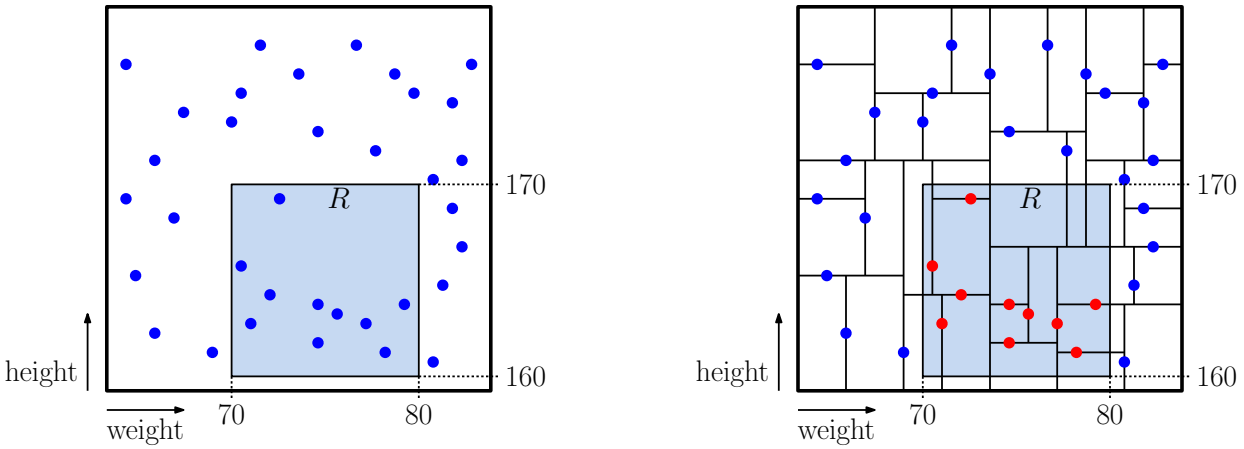



Fig. 65: Orthogonal range query.

(assuming Java-like indexing). In addition to a constructor, the class provides a few useful geometric primitives (illustrated in Fig. 66).

`boolean contains(Point q)`: Returns `true` if and only if point $q$ is contained within this rectangle (using the above inequalities).

`boolean contains(Rect c)`: Returns `true` if and only if this rectangle contains rectangle $c$. This boils down to testing containment on all the intervals defining each of the rectangles' sides:

$$\big[c.\text{low}[i], c.\text{high}[i]\big] \subseteq \big[\text{low}[i], \text{high}[i]\big], \quad \text{for all } 0 \le i \le d-1.$$

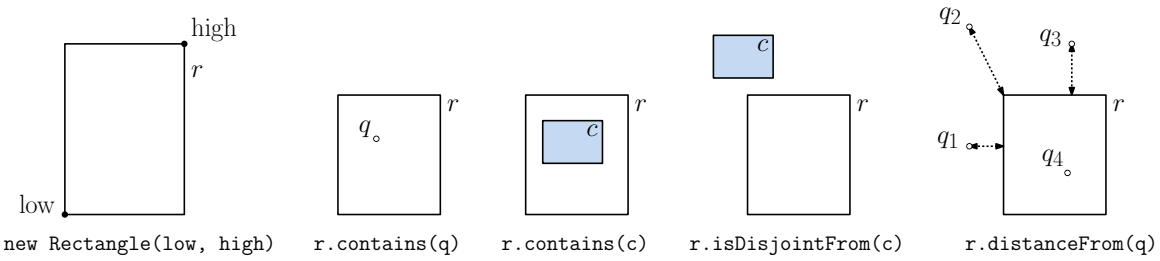



Fig. 66: An axis-parallel rectangle methods.

`boolean isDisjointFrom(Rect c)`: Returns `true` if and only if rectangle $c$ is disjoint from this rectangle. This boils down to testing whether any of the defining intervals are disjoint, that is

$$r.\text{high}[i] < c.\text{low}[i] \text{ or } r.\text{low}[i] > c.\text{high}[i], \quad \text{for any } 0 \le i \le d-1.$$

`double distTo(Point q)`: Returns the minimum Euclidean distance from $q$ to any point of this rectangle. This can be computed by computing the distance from the coordinate $q[i]$ to this rectangle's $i$th defining interval, taking the sums of squares of these distances, and then taking the square root of this sum:

$$\sqrt{\sum_{i=0}^{d-1} (\text{distance}(q[i], \big[\text{low}[i], \text{high}[i]\big]))^2}$$

There is one additional function worth discussing, because it is used in many algorithms that involve kd-trees. The function is given a rectangle $r$ and a splitting point $s$ lying within the rectangle. We want to cut the rectangle into two sub-rectangles by a line that passes through the splitting point. These are used in a context where the rectangle $r$ represents the cell associated with a given kd-tree node, and by cutting the cell through the splitter, we generate the cells associated with the node's left and right children.

`Rect leftPart(int cd, Point s):` (and `rightPart(int cd, Point s)`) These are both given a cutting dimension `cd` and a point `s` that lies within the rectangle. The first returns the subrectangle lying to the left (below) of $s$ with respect to the cutting dimension, and the other returns the subrectangle lying to the right (above) of $s$ with respect to the cutting dimension (see Fig. 66). More formally, `leftPart(cd, s)`, returns a rectangle whose low point is the same as r.low and whose high point is the same as r.high except that the cd-th coordinate is set to $s$[cd]. Similarly, `rightPart(cd, s)`, returns a rectangle whose high point is the same as r.high and whose low point is the same as r.low except that the cd-th coordinate is set to $s$[cd].

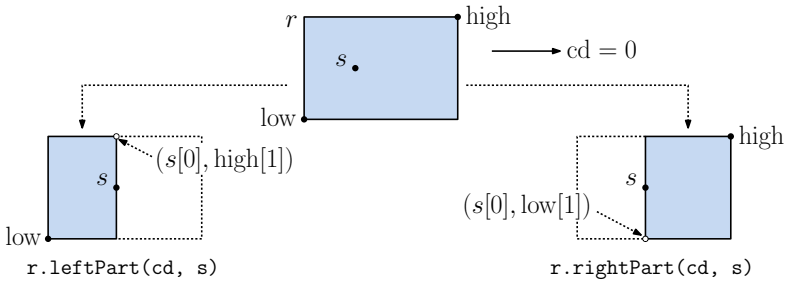



Fig. 67: The functions `leftPart` and `rightPart`.

The following code block provides a high-level overview of the `Rect` class (without defining any of the functions).

A skelton of a simple Rect class

```
public class Rect {
    private Point low;                          // lower left corner
    private Point high;                         // upper right corner

    public Rect(Point low, Point high) ...      // constructor
    public boolean contains(Point q) ...        // do we contain q?
    public boolean contains(Rect c) ...         // do we contain rectangle c?
    public boolean isDisjointFrom(Rect c) ...   // disjoint from rectangle c?
    public double distTo(Point q) ...           // minimum distance to point q
    public Rect leftPart(int cd, Point s) ...   // left part from s
    public Rect rightPart(int cd, Point s) ...  // right part from s
}
```

**Anwering the Range Query:** In order to answer range counting queries, let us first assume that each node `p` of the tree has been augmented with a member `p.size`, indicating the number of points lying within the subtree rooted at `p`. This can easily be updated as points are inserted to and deleted from the tree. The counting function, `rangeCount(R, p, cell)` operates recursively. The first argument `R` is the query range, the second argument `p` is the node

currently visited, and `cell` is its associated cell. It returns a count of the number of points within `p`'s subtree that lie within `R`. The initial call is `rangeCount(R, root, boundingBox)`, where `boundingBox` is the bounding box of the entire kd-tree.

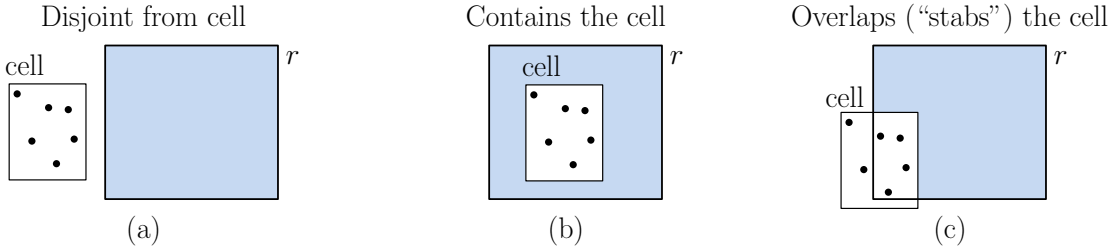



Fig. 68: Cases arising in orthogonal range searching.

The function operates recursively, working from the root down to the leaves. First, if we fall out of the tree then there is nothing to count. Second, if the current node's cell and the range are completely disjoint, we may return 0, because none of this node's points lie within the range (see Fig. 68). Next, if the query range completely contains the current cell, we can count all the points of `p` as lying within the range, and so we return `p.size`. Otherwise, the range partially overlaps the cell. We say that the range *stabs* the cell. In this case, we apply the function recursively to each of our two children. The function is presented in the code block below.

kd-tree Range Counting Query

```
int rangeCount(Rect R, KDNode p, Rect cell) {
    if (p == null) return 0             // empty subtree
    else if (R.isDisjointFrom(cell))    // no overlap with range?
        return 0
    else if (R.contains(cell))          // the range contains our entire cell?
        return p.size                   // include all points in the count
    else {                              // the range stabs this cell
        int count = 0
        if (R.contains(p.point))        // consider this point
            count += 1
                                        // apply recursively to children
        count += rangeCount(R, p.left,  cell.leftPart(p.cutDim, p.point))
        count += rangeCount(R, p.right, cell.rightPart(p.cutDim, p.point))
        return count
    }
}
```

**An Example:** Fig. 69 shows an example of a range search. Next to each node we store the size of the associated subtree in blue. We say that a node is *visited* if a call to `rangeCount()` is made on this node. We say that a node is *processed* if both of its children are visited. Observe that for a node to be processed, its cell must overlap the range without being contained within the range. In the example, the shaded nodes are those that are not processed. For example the cell associated with node $h$ (shaded in pink) is entirely contained within the range, and any points in its subtree can be safely included in the count. (In this case, this includes the three points $t$, $h$, and $p$.) The cells associated with nodes $j$ and $g$ (shaded in gray) are entirely disjoint from the query, and the subtrees rooted at these nodes can be completely

ignored. The nodes with red squares surrounding them those whose points have been added individually to the count (by the condition `R.contains(p.point)`). There are four such nodes $d$, $f$, $l$, and $q$. Combined with the three points of $h$'s subtree, the total count returned is 7.

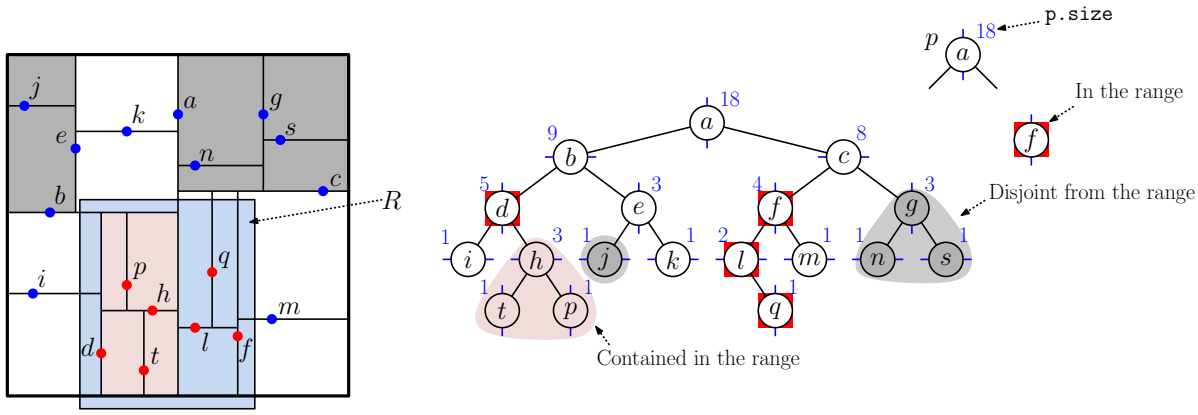



Fig. 69: Range search in kd-trees. The subtree rooted at $h$ is counted entirely. The subtrees rooted at $j$ and $g$ are excluded entirely. The other points are checked individually.

**Analysis of query time:** How many nodes does this method visit altogether? We claim that the total number of nodes is $O(\sqrt{n})$ assuming a balanced kd-tree (which is a reasonable assumption in the average case).

**Theorem:** Given a balanced kd-tree with $n$ points in $\mathbb{R}^2$ (where the cutting dimension alternates between $x$ and $y$), orthogonal range counting queries can be answered in $O(\sqrt{n})$ time.

Recall from the discussion above that a node is processed (both children visited) if and only if the range partially overlaps or "stabs" the cell. To bound the total number of nodes that are processed in the search, it suffices to count the total number of nodes whose cells are stabbed by the query rectangle. Rather than prove the above theorem directly, we will prove a simpler result, which illustrates the essential ideas behind the proof. Rather than using a 4-sided rectangle, we consider an orthogonal range having a only one side, that is, an orthogonal halfplane. In this case, the query stabs a cell if the vertical or horizontal line that defines the halfplane intersects the cell.

**Lemma:** Given a balanced kd-tree with $n$ points in $\mathbb{R}^2$ (where the cutting dimension alternates between $x$ and $y$), any vertical or horizontal line stabs the cells of $O(\sqrt{n})$ nodes of the tree.

**Proof:** It will simplify the analysis to assume that the tree is "perfectly balanced", which means that if a subtree contains $m$ points then each of its two subtrees contains at most $m/2$ points. (The proof generally works as long as the height of the tree is $O(\log n)$, but it is a bit more complicated.)

By symmetry, it suffices to consider a horizontal line. Consider a processed node which has a cutting dimension along $x$. A horizontal line can stab the cells of both its children. On the other hand, if the cutting dimension is along $y$, a horizontal line either stabs

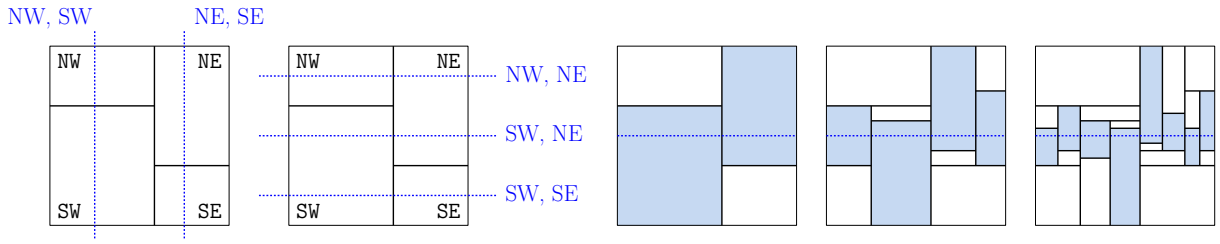



Fig. 70: An axis-parallel line in 2D can stab at most two out of four cells in two levels of the kd-tree decomposition. In general, it stabs $2^i$ cells at level $2i$.

the upper cell or the lower cell, but not both. (If our horizontal line coincides with the cutting line, then we consider it to stab the upper cell and not the lower cell.)

Since we alternate splitting on $x$ then $y$, this means that after descending two levels of the tree, we may stab the cells of at most two of the possible four grandchildren of the original node. (This is illustrated in Fig. 70.) By our assumption that the tree is balanced, if the parent node has $n$ points, each of its two children has at most $n/2$ points, and each of the four grandchildren has at most $n/4$ points. Therefore, the total number of nodes whose cells are stabbed satisfies the following recurrence:

$$T(n) = \begin{cases} 2 + 2T(n/4) & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

We can solve this recurrence either by appealing to the *Master Theorem* (see the CLRS Algorithms book), but it is easy enough to solve directly. By expanding the recurrence and observing the trend, we obtain:

$$\begin{aligned} T(n) &= 2 + 2T(n/4) \\ &= 2 + 2(2 + T(n/16)) \;=\; 2 + 4 + 4T(n/16) \\ &= (2 + 4) + 4(2 + 2T(n/64)) \;=\; (2 + 4 + 8) + 8T(n/64) \\ &= \cdots \\ &= \sum_{i=1}^{k} 2^i + 2^k T(n/4^k). \end{aligned}$$

To get to the basis case of $T(1)$, we set $n/4^k = 1$, which yields $k = \log_4 n = (\lg n)/(\lg 4) = (\lg n)/2$. The summation term (which is the dominant term) in $T(n)$ is:

$$\sum_{i=1}^{k} 2^i \;\approx\; 2 \cdot 2^k \;=\; 2 \cdot 2^{(\lg n)/2} \;=\; 2 \cdot (2^{\lg n})^{1/2} \;=\; 2 \cdot n^{1/2} \;=\; 2\sqrt{n} \;=\; O(\sqrt{n}).$$

This completes the proof.

We have shown that any (infinite) vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. This upper bound clearly holds for any finite vertical or horizontal line segment. Thus, if we apply it to the four line segments that bound $R$, it follows that the the total number of cells stabbed by the query range $R$ is $O(4\sqrt{n}) = O(\sqrt{n})$. The total query time is determined by the sum of nodes visited, which is dominated by the sum of the nodes that are stabbed by

the query. Therefore, the overall running time (assuming a balanced kd-tree and alternating cutting dimensions) is $O(\sqrt{n})$. This completes the proof of the above lemma.

To see whether you understand this, you might try generalizing this analysis to arbitrary dimensions $d$ (where $d$ is constant). As a hint, the query time in general will be $O(n^{1-1/d})$. In the case where $d = 2$, this is $O(\sqrt{n})$. Observe that as $d$ gets larger and larger, the query time approaches $O(n)$. Unfortunatey, $O(n)$ is the same time as brute-force search (since we can simply test every point one-by-one to see whether it lies in the range). Thus, kd-trees are efficient only for fairly small values of $d$.

**Nearest-Neighbor Queries:** Next we consider how to perform an important retrieval query on a kd-tree. Nearest neighbor queries are among the most important queries. We are given a set of points $S$ stored in a kd-tree, and a query point $q$, and we want to return the point of $S$ that is closest to $q$. Let's assume that distances are measured using Euclidean distances. In particular, given two points $s = (s_1, \ldots, s_d)$ and $q = (q_1, \ldots, q_d)$, their Euclidean distance is

$$\text{dist}(s, q) \;=\; \sqrt{(s_1 - q_1)^2 + \cdots + (s_d - q_d)^2}.$$

An example is shown in Fig. 71. Observe that the circle centered at $q$ and passing through its nearest neighbor $s$ contains no other points. However, every leaf cell of the kd-tree whose cell overlaps the interior of this circle (shaded in the figure) may need to be visited in the search, since each might contribute a point that could be closer to $q$ than $s$ is. What makes the search efficient is that the number of such nodes is usually much smaller than the total number of nodes in the tree. Of course, finding these nodes is the key issue in answering nearest neighbor queries.

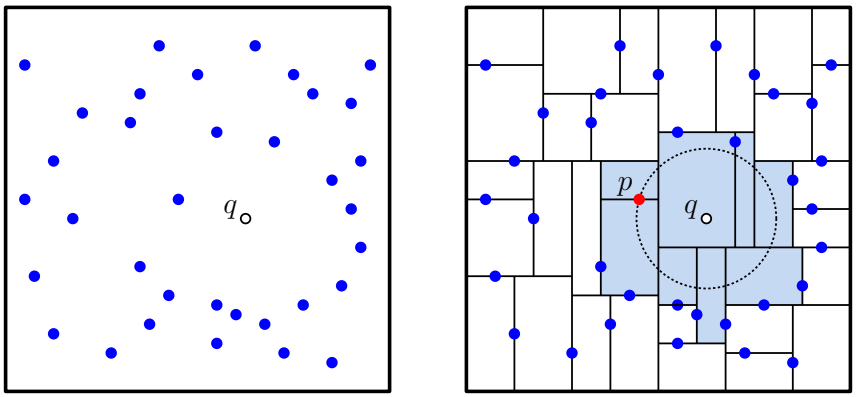



Fig. 71: Nearest-neighbor searching using a kd-tree.

An intuitively appealing approach to nearest neighbor queries would be to find the leaf node of the kd-tree that contains $q$ and then search this and the neighboring cells of the kd-tree. The problem is that the nearest neighbor may actually be very far away, in the sense of the tree's structure. For example, in Fig. 72, many of the points are at nearly the same distance from the query point $q$. It would be necessary to visit almost all the nodes of the tree to determine which of these points is the actual nearest neighbor.

We will need a more systematic approach to finding nearest neighbors. Nearest neighbor queries illustrate three important elements of range and nearest neighbor processing.

**Partial results:** Store the intermediate results of the query and update these results as the query proceeds.

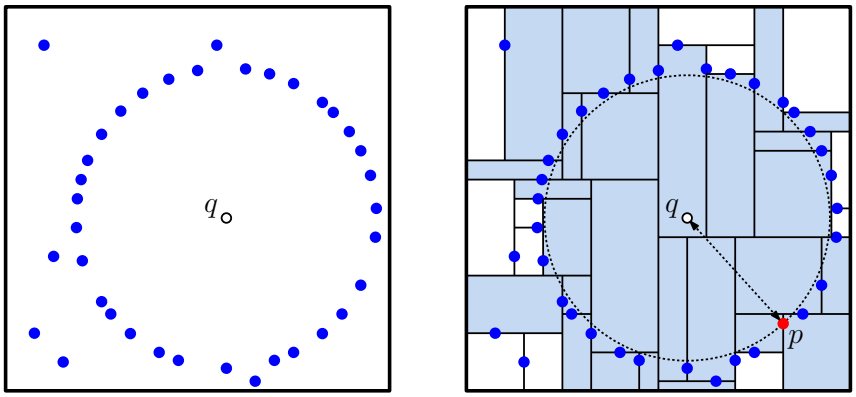



Fig. 72: A (nearly) worst-case scenario for nearest-neighbor searching. Almost all the nodes of the tree need to be visited, since any might be the nearest neighbor.

**Traversal order:** Visit the subtree first that is more likely to be relevant to the final results.

**Pruning:** Do not visit any subtree that be judged to be irrelevant to the final results.

**Nearest-neighbor Utilities:** Before presenting the code for nearest-neighbor searching, we introduce a few helpful utilities. First, recall that every cell of the kd-tree is associated with an axis-parallel rectangle, called its *cell*. (For $d \geq 3$ the generalization of a rectangle is called a *hyperrectangle*, but we will just use the term "rectangle" for simplicity.) A convenient way to represent a rectangle in any $d$-dimensional space is to give two points `low` and `high`. In 2D, these represent the lower-left and upper-right corners of the rectangle, respectively. In general, the rectangle consists of all points $q$ such that $\text{low}_i \leq q_i \leq \text{high}_i$ (see Fig. 66(a)). A possible implementation, without any details, is outlined in the code block below. (We make use of the `Point` object, which was introduced in the previous lecture.)

**Nearest-neighbor Code:** Our procedure for returning the nearest neighbor returns the nearest point to the query point `q`. As usual, we employ a recursive utility function that works on an individual node `p` of the tree. The function `nearNeigh(q, p, cell, best)` is given four parameters:

- the query point `q`
- the current node `p` of the tree
- the rectangular cell associated with this node, `cell`, and
- the closest point to `q` so far, called `best`.

The procedure works as follows:

- First, if `p` is `null`, we must have fallen out of the tree, and we just return the current `best` as the answer.
- Otherwise, we determine whether `p.point` is closer to `q` than is `best`. If so, `p` becomes the new `best` (see Fig. 73(a)).
- Next, we need to search the subtrees for possibly closer points:
  - We invoke `leftPart` and `rightPart` to determine the cells of the left and right subtrees, respectively.

– Next, we check on which side of of the splitting plane the query point lies. If `q` lies on the left side, we are more likely to find the nearest neighbor in the left subtree, so we search that first. Otherwise, we search the right subtree first.

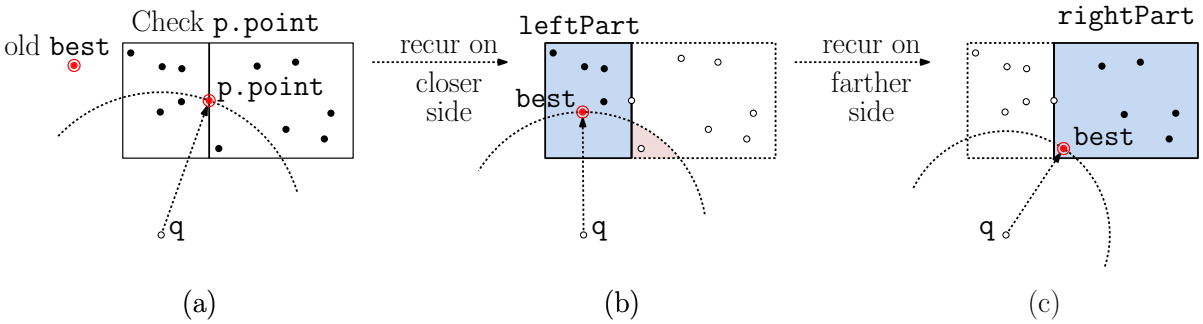



Fig. 73: Nearest-neighbor searching.

– We recursively visit the closer subtree first (see Fig. 73(b)), and update `best` accordingly.
– After returning from this call, we compute `q`'s distance to the right subtree cell. Observe that if this distance is greater than `q`'s distance to `best`, there is no chance that the other subtree contains the nearest neighbor, and so there is no need to visit this subtree. Otherwise, we apply the search recursively to the right subtree (see Fig. 73(c)) and update `best` accordingly.

Given a query point $q$, the initial call is `nearNeigh(q, root, rootCell, null)`, where `rootCell` is the rectangle that encloses the entire tree contents. Let us assume that the `distTo` function return $+\infty$ (e.g., `Double.POSITIVE_INFINITY` in Java) if the argument is `null`. The code is presented below.

Nearest-neighbor helper

```
Point nearNeigh(Point q, KDNode p, Rect cell, Point best) {
    if (p == null) return best
    if (q.distTo(p.point) < q.distTo(best)) best = p        // new closest point
    cd = p.cutDim;                                          // cutting dimension
    Rect leftCell = cell.leftPart(cd, p.point)              // left child's cell
    Rect rightCell = cell.rightPart(cd, p.point)            // right child's cell

    if (q[cd] < p.point[cd]) {                              // q is closer to left
        best = nearNeigh(q, p.left, leftCell, best)         // search left subtree
        if (rightCell.distTo(q) < q.distTo(best))           // is right viable?
            best = nearNeigh(q, p.right, rightCell, best)
    } else {                                                // q is closer to right
        best = nearNeigh(q, p.right, rightCell, best)       // search right subtree
        if (leftCell.distTo(q) < q.distTo(best))            // is left viable?
            best = nearNeigh(q, p.left, leftCell, best)
    }
    return best;
}
```

An example of the algorithm in action is shown in Fig. 74. The algorithm starts by descending to the leaf node (the upper child of $(70, 30)$), computing distances to all the points seen along

the way. At this point $(70, 30)$ is `best`. Because the lower child of $(70, 30)$ overlaps the best-neighbor ball (from `q` to `best`), we need to inspect this subtree. When we visit $(50, 25)$, we discover that it is even closer. We visit both its children. However, observe that when we arrive at $(60, 10)$, we visit the closer of its two children (the empty subtree lying above this point), but there is no need to visit its lower child, because it lies entirely outside of the best-neighbor ball. We then return from the recursion. On returning to $(80, 40)$ and $(70, 80)$, we see that the cells of their other children lie entirely outside the best-neighbor ball, and so we do not need to visit them. On returning to the root at $(35, 90)$ we see that its left subtree does overlap the best-neighbor ball, and so we recurse on that subtree as well. We continue until arriving at the closest leaf to the query point, namely the right child of $(25, 10)$. We compute distances too all the points associated with the nodes visited, and we discover along the way that $(25, 50)$ is closer to the query point. After this, all the remaining cells (shaded in white in the figure) lie outside the best-neighbor ball, and so we can terminate the search.

**Analysis:** How efficient is this procedure? The worst-case performance can be bad because as seen in Fig. 72, there are cases where we may need to visit almost every node of the tree. However, this is really a very pathological example. In most instances, the typical running time is much closer to $O(2^d + \log n)$, where $d$ is the dimension of the space. Generally, you expect to visit some set of nodes that are in the neighborhood of the query point (giving rise to the $2^d$ term) and require $O(\log n)$ time to descend the tree to find these nodes.

## Lecture 12: Scapegoat Trees

**Overview:** In this lecture, we will discuss an ordered (tree-based) dictionary data structure that is efficient, $O(\log n)$ time, but in an amortized sense. Recall that this means that, over a series of operations, the average cost per operation is $O(\log n)$, even though the cost of any individual operation can be much higher. The worst case for find will be $O(\log n)$, but the worst case for insert and delete can both be as high as $O(n)$.

This data structure has the unusual name of *scapegoat tree*. The idea underlying this data structure was due initially to Arne Anderson (of AA trees) in 1989. This idea was rediscovered by Igal Galperin and Ron Rivest in 1993, who made some refinements and gave it the name "scapegoat tree," which we will explain below. (By the way, Rivest is quite famous in cryptography. The "R" in the RSA public crypto system comes from his name.)

**Wreck it Ralph:** While amortized data structures often interesting in their own right, there is a particular motivation for studying the scapegoat tree. So far, all of the binary search trees that we have studied achieve balance "incrementally" through the use of rotation operation. Scapegoat trees are unique in that they do not rely on rotation. This is significant because there exist binary trees that cannot be balanced through the use of rotations. (A prime example is the kd-tree data structure for storing geometric objects.) The scapegoat tree achieves good balance in a very different way—when a subtree is imbalanced, it is tossed out and rebuilt from scratch (or "wrecked and fixed").

Below, we will discuss the details of how the scapegoat tree works. Here is a high-level overview. A scapegoat tree is a binary search tree, which does not need to store any additional information in the nodes, other than the key, value, and left and right child pointers. Nonetheless, the height of a tree with $n$ nodes will always be $O(\log n)$. (Note that this is not the case for splay trees, whose height can grow to as high as $O(n)$.)

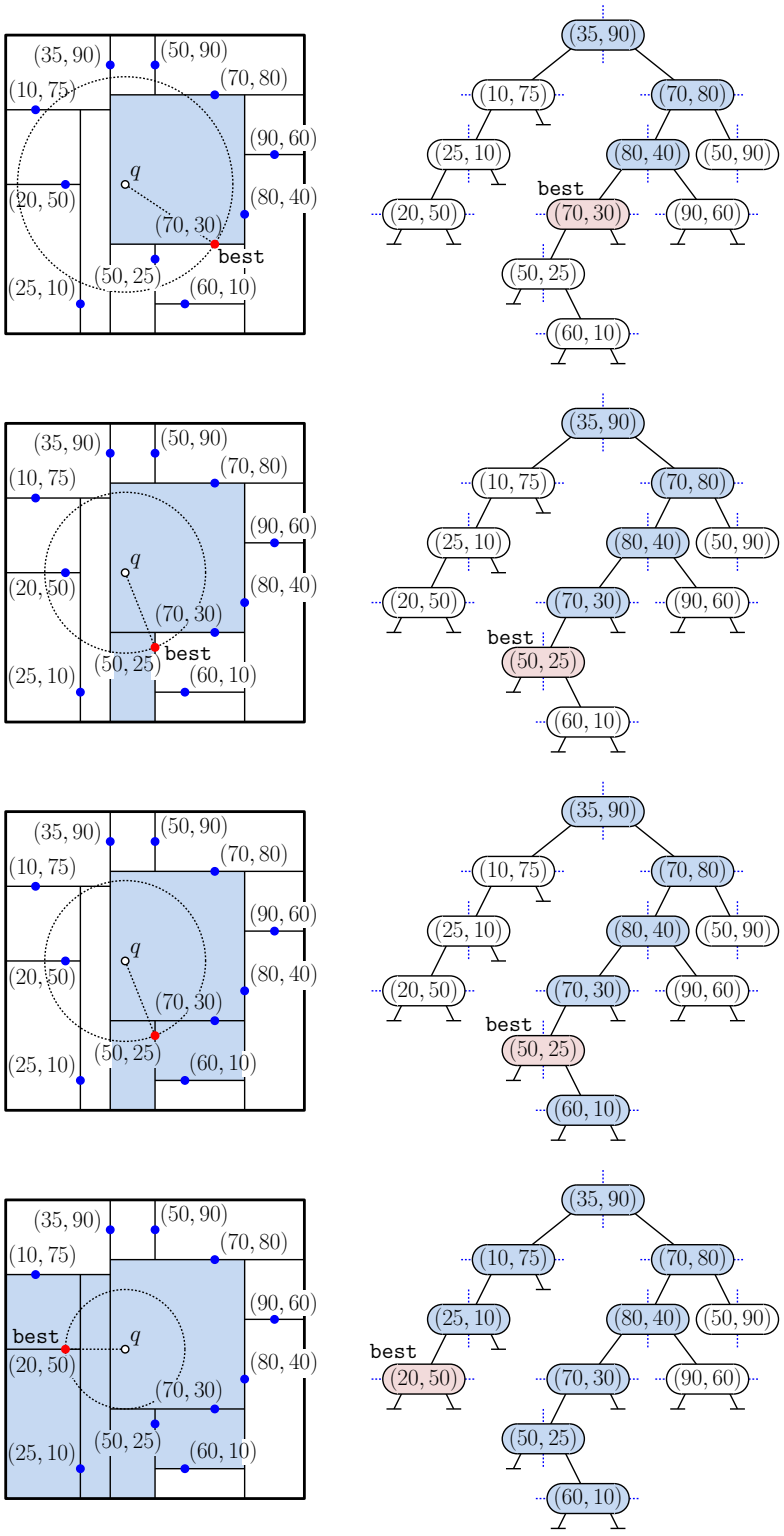



Fig. 74: Nearest-neighbor search.

In an ideally balanced tree, each child has almost nearly exactly half as many nodes as its parent. Define the *size* of a node to be the total number of nodes in its subtree. A node is said to be *weight balanced* if the sizes of its two subtrees are within a constant fraction of either other (e.g., split no worse than $\frac{1}{3} \cdots \frac{2}{3}$). The scapegoat tree attempts to maintain this property. Insertion and deletions work roughly as follows.

**Insertion:**

- The key is first inserted just as in a standard (unbalanced) binary tree
- We monitor the depth of the inserted node after each insertion, and if it is too high, we can infer there must be at least one node on the search path that has poor weight balance (that is, its left and right children have very different sizes).
- In such a case, we find such a node, called the *scapegoat*,[7] and we completely rebuild the subtree rooted at this node so that it is perfectly balanced.

**Deletion:**

- The key is first deleted just as in a standard (unbalanced) binary tree
- Once the number of deletions performed is sufficiently large relative to the entire tree size, rebuild the entire tree so it is perfectly balanced.

**Why the Asymmetry?** You might wonder why there is a notable asymmetry between the rebuilding rules for insertion and deletion. The existence of a single very deep node is proof that a tree is out of balance. Thus, for insertion, we can use the fact that the inserted node is too deep to trigger rebuilding. However, observe that the converse does not work for deletion.

The natural counterpart would be "if the depth of the leaf node containing the deleted key is too small, then trigger a rebuilding operation." But the fact that a single node has a low depth, does not imply that the rest of the tree is out of balance. (It may just be that a single search path has low depth, but the rest of the tree is perfectly balanced.)

Could we instead apply the deletion rebuilding trigger to work for insertion? Again, this will not work. The natural counterpart would be, "given a newly rebuild tree with $n$ keys, we will rebuild it after inserting roughly $n/2$ new keys." However, if we are very unlucky, all these keys may fall along a single search path, and the tree's height would be as bad as $O((\log n) + n/2) = O(n)$, and this is unacceptably high.

**How to Rebuild a Subtree:** Before getting to the details of how the scapegoat tree works, let's consider the basic operation that is needed to maintain balance, namely rebuilding subtrees into balanced form. We shall see that if the subtree contains $n$ keys, this operation can be performed in $O(n)$ time (see Fig. 134). Letting `p` denote the root node of the subtree to rebuild, call this function `rebuild(p)`:

- Perform an inorder traversal of `p`'s subtree, copying the keys to an array `A[0..k-1]`, where `k` denotes the number of nodes in this subtree. Note that the elements of $A$ are sorted.
- Invoke the following recursive subtree-building function: `buildSubtree(A)`
  - Let `k = A.length`.
  - If `k == 0`, return an empty tree, that is, `null`.

[7] The colorful term "scapegoat" refers to an individual who is assigned the blame when something goes wrong. In this case, the unbalanced node takes the blame for the tree's height being too great.

– Otherwise, let `x` be the median key, that is, `A[j]`, where `j` $= \lfloor$`k/2`$\rfloor$. Recursively invoke `L = buildSubtree(A[0..j-1])` and `R = buildSubtree(A[j+1..k-1])`. Finally, create an internal node containing $x$ with left subtree $L$ and right subtree $R$. Return a pointer to $x$.

Note that if `A` is implemented as a Java `ArrayList`, there is a handy function called `subList` for performing the above splits, without the need to explicitly copy elements into new arrays. The `buildSubtree` function is given in the code block below.

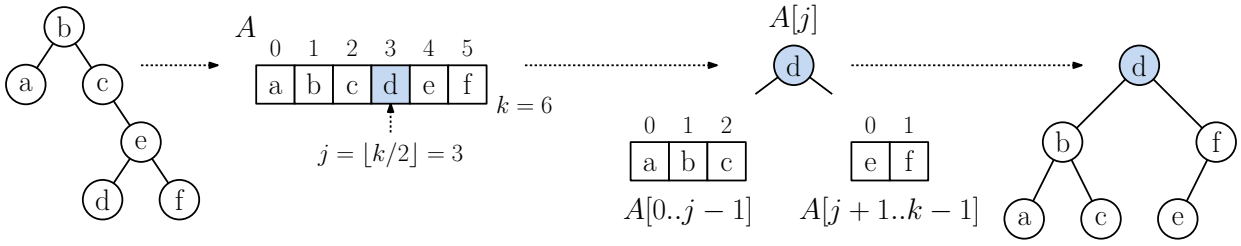



Fig. 75: Recursively building a balanced tree from a sorted array of keys.

Building a Balanced Tree from an Array

```
Node buildSubtree(Key[] A) {                    // A is a sorted array of keys
    k = A.length
    if (k == 0) return null                     // empty array
    else {
        j = floor(k/2)                          // median of the array
        Node p = new Node(A[j])                 // ...this is the root
        p.left = buildSubtree(A[0..j-1])        // build left subtree recursively
        p.right = buildSubtree(A[j+1..k-1])     // build right subtree recursively
        return p                                // return root of the subtree
    }
}
```

Ignoring the recursive calls, we spend $O(1)$ time in each recursive call, so the overall time is proportional to the size of the tree, which is $k$, so the total time is $O(k)$.

**Scapegoat Tree Operations:** In addition to the nodes themselves, the scapegoat tree maintains two integer values. The first, denoted by $n$, is just the actual number of keys in the tree. The second, denoted by $m$, is a special parameter, which is used to trigger the event of rebuilding the entire tree.

In particular, whenever we insert a key, we increment $m$, but whenever we delete a key we do not decrement $m$. Thus, $m \geq n$. The difference $m - n$ intuitively represents the number of deletions. When we reach a point where $m > 2n$ (or equivalently $m - n > n$) we can infer that the number of deletions exceeds the number of keys remaining in the tree. In this case, we rebuild the entire tree in balanced form.

We are now in a position to describe how to perform the dictionary operations for a scapegoat tree.

Initialization: $n \leftarrow m \leftarrow 0$ and the root is initialized to `null`.

`find(Key x)`: The find operation is performed exactly as in a standard (unbalanced) binary search tree. We will show that the height of the tree never exceeds $\log_{3/2} n \approx 1.7 \lg n$, so this is guaranteed to run in $O(\log n)$ time.

`delete(Key x)`: This operates exactly the same as deletion in a standard binary search tree. After deleting the node, decrement $n$ (but do not change $m$). If $m > 2n$, rebuild the entire tree by invoking `rebuild(root)`, and set $m \leftarrow n$.

`insert(Key x, Value v)`: First, increment both $n$ and $m$. We start by applying the insertion process a standard (unbalanced) binary search tree. As we are tracing the search path to the insertion point, keep track of our depth in the tree. (Recall that depth is the number of edges to root.) If the depth of the inserted node exceeds $\log_{3/2} m$ then we trigger a *rebuilding event*. This involves the following:

- Walk back up along the insertion search path towards the root. Let `p` be the current node that is visited, and let `p.child` be the child of `p` that lies on the search path.
- Let `size(p)` denote the *size* of the subtree rooted at `p`, that is, the number of nodes in this subtree.
- If
$$\frac{\texttt{size(p.child)}}{\texttt{size(p)}} > \frac{2}{3},$$
then rebuild the subtree rooted at `p` by invoking `rebuild(p)`. The node `p` is the *scapegoat*. After the rebuild is done, we terminate the insertion process. Even if `p` has an ancestor that satisfies the scapegout condition, we do not do a second rebuild as part of this insertion.

An example of insertion is shown in Fig. 135. After inserting 5, the tree has $n = 11$ nodes. The newly inserted node is at depth 6, and since $6 > \log_{3/2} 11$ (which is approximately 5.9), we trigger the rebuilding event. We walk back up the search path. We find node 9 whose size is 7, but the child on the search path has size 6, and $6/7 > 2/3$, so we invoke rebuild on the node containing 9.

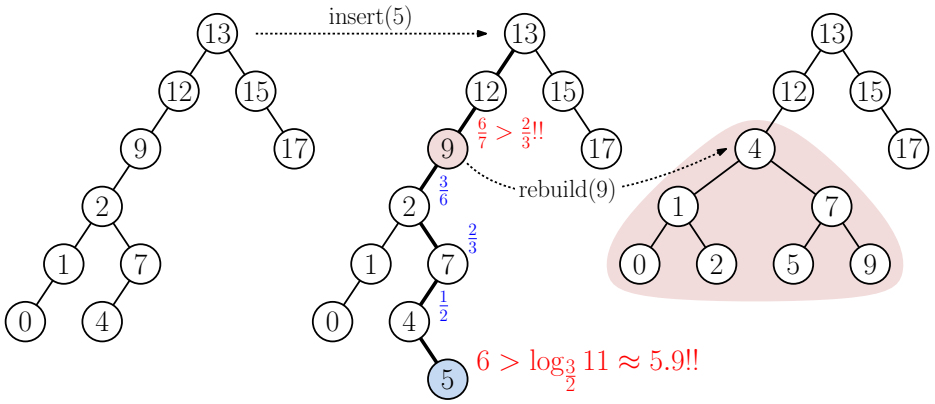



Fig. 76: Inserting a key into a scapegoat tree, which triggers a rebuilding event. The node containing 9 is the first scapegoat candidate encountered while backtracking up the search path and is rebuilt.

**Must there be a scapegoat?** The fact that a child has over 2/3 of the nodes of the entire subtree intuitively means that this subtree has (roughly) more than twice as many nodes as its sibling. We call such a node on the search path a *scapegoat candidate*. A short way of summarize the above process is "rebuild the scapegoat candidate that is closest to the insertion point.'

You might wonder whether we will necessarily encounter an scapegoat candidate when we trace back along the search path. The following lemma shows that this is always the case.

**Lemma:** Given a binary search tree of $n$ nodes, if there exists a node $p$ such that $\text{depth}(p) > \log_{3/2} n$, then $p$ has an ancestor (possibly $p$ itself) that is a scapegoat candidate.

**Proof:** The proof is by contradiction. Suppose to the contrary that no node from $p$ to the root is a scapegoat candidate. This means that for every ancestor node $u$ from $p$ to the root, we have $\text{size}(u.\text{child}) \leq \frac{2}{3} \cdot \text{size}(u)$.

We know that the root has a size of $n$. Therefore, its child on the search path has size at most $(2/3)n$, its grandchild has size at most $(2/3)((2/3)n) = (4/9)n$, and generally the node at depth $i$ along the search path as size at most $(2/3)^i n$.

Let $d$ denote the depth of $p$. We know what its subtree rooted at $p$ must have at least one node (namely $p$ itself), and therefore

$$1 \;\leq\; \text{size}(p) \;\leq\; \left(\frac{2}{3}\right)^d n.$$

Solving for $d$, we have

$$\left(\frac{3}{2}\right)^d \;\leq\; n \quad\Longrightarrow\quad d \;\leq\; \log_{3/2} n.$$

However, this violates our hypothesis that $p$'s depth exceeds $\log_{3/2} n$, yielding the desired contradiction.

Recall that $m \geq n$, and so if a rebuilding event is triggered, the insertion depth is at least $\log_{3/2} m$, which means that it is at depth at least $\log_{3/2} n$. Therefore, by the above lemma, there must be a scapegoat candidate along the search path.

**The Sizeless Size?** No, this is not the answer to some Zen koan. We mentioned earlier that the scapegoat tree does not store any information in the nodes other than the key, value, and left and right child pointers. So how can we compute `size(u)` for a node `u` during the insertion process, without this information? There is very clever trick for doing this on the fly.

Since we are doing this as we back up the search path, we may assume that we already know the value of $s' =$ `size(u.child)`, where this is the child that lies along the insertion search path. So, to compute `size(u)`, it suffices to compute the size of `u`'s other child. To do this, we perform a traversal of this child's subtree to determine its size $s''$. Given this, we have $\texttt{size}(u) = 1 + s' + s''$, where the $+1$ counts the node `u` itself.

You might wonder, how can we possibly expect to achieve $O(\log n)$ amortized time for insertion if we are using brute force (which may take as much as $O(n)$ time) to compute the sizes of the subtrees? The reason is to first recall that we do not need to compute subtree sizes unless a rebuild event has been triggered. Every node that we are visiting in the counting process will need to be visited again in the rebuilding process. Thus, the cost of this counting process can be "charged for" in the cost of the rebuilding process, and hence it essentially comes for free!

**The Lazy Way:** By the way, there is an easier way for computing subtree sizes, but this requires that we use additional space to store the size value of each node explicitly within each node. The size of a node is easy to update whenever there are changes in the tree's structure, since we have:

```
size(u) = (u == null ? 0 : 1 + size(u.left) + size(u.right))
```

**Amortized Analysis:** Because we rebuild the tree whenever its height exceeds $O(\log n)$, `find` operations run in $O(\log n)$ time, even in the worst case. The insert/delete operations can take longer, however, because they may trigger rebuilding. The next theorem shows that for a sequence of $k$ operations, the total cost for insertion and deletion is $T(k) = O(k \log k)$. It follows immediately that the amortized cost for the operations is $T(k)/k = O(\log k)$. The proof is based on a presentation from the Open Data Structures web site.

**Theorem:** Starting with an empty Scapegoat Tree any sequence of $k$ `insert` and `delete` operations (including rebuilding) can be performed in $O(k \log k)$ time.

**Proof:** As usual, we will apply our usual token-based argument. We imagine that each node stores a number of tokens. Each token can pay for some constant, $c$, units of time spent rebuilding. Our scheme will give out a total of $O(k \log k)$ tokens, and we will show that every call to `buildSubtree(u)` will be paid for with tokens stored at node `u`.

During an insertion or deletion, we give one token to each node along the path to the inserted or deleted node. Recalling that $n$ denotes the number of elements in the tree at any time, we have $n \leq k$. As shown earlier, the height of the tree is never greater than $\log_{3/2} n$ (for otherwise we rebuild). Therefore, we generate at most $\log_{3/2} m \leq \log_{3/2} k$ new tokens per operation. During a deletion we also store an additional token on the side. Thus, in total we generate at most $O(k \log k)$ tokens. All that remains is to show that these tokens are sufficient to pay for all calls to `buildSubtree(u)`.

If we call `buildSubtree(u)` during an insertion, it is because `u` is a scapegoat. Suppose, without loss of generality, that

$$\frac{\texttt{size(u.left)}}{\texttt{size(u)}} > \frac{2}{3}.$$

By definition of size, $\texttt{size(u)} = 1 + \texttt{size(u.left)} + \texttt{size(u.right)}$. It follows that

$$\frac{1}{2}\texttt{size(u.left)} > \texttt{size(u.right)},$$

and therefore

$$\texttt{size(u.left)} - \texttt{size(u.right)} > \frac{1}{2}\texttt{size(u.left)} > \frac{1}{3}\texttt{size(u)}.$$

Now, the last time a subtree containing `u` was rebuilt (or when `u` was inserted, if a subtree containing `u` was never rebuilt), the tree was perfectly balanced, and thus

$$\texttt{size(u.left)} - \texttt{size(u.right)} \leq 1.$$

Therefore, the number of `insert` or `delete` operations that have affected `u.left` or `u.right` since then is at least

$$\frac{1}{3}\texttt{size(u)} - 1.$$

It follows that at least this many tokens stored at `u` that are available to pay for the $O(\texttt{size(u)})$ time it takes to call `buildSubtree(u)`.

If we call `buildSubtree(u)` during a deletion, it is because $m > 2n$. In this case, we have $m - n > n$ tokens stored on the side, and we use these to pay for the $O(n)$ time it takes to rebuild the root. This completes the proof.

# Lecture 13: Splay Trees

**Recap:** We have discussed a number of different search structures for performing the basic ordered-dictionary operations (insert, delete, and find). Here is a brief review:

**(Standard) Binary Search Trees:** Very simple, but no effort is made to balance the tree. Height is $O(\log n)$ if keys are inserted in random order, but can be as bad as $\Omega(n)$.

**AVL Trees:** These are height-balanced trees. Height is guaranteed to be $O(\log n)$, and all dictionary operations run in $O(\log n)$ time. Each node stores height information (or the balance factor), and the tree is rebalanced by rotations.

**2-3, Red-Black, and AA Trees:** 2-3 trees were based on the notion of having a variable-width node, but the tree was perfectly balanced. Red-black and AA trees were binary tree encodings of 2-3 trees. For all three, the height is guaranteed to be $O(\log n)$, and all dictionary operations run in $O(\log n)$ time.

**Scapegoat Trees:** These are standard binary search trees (no balance information), which maintain balance by rebuilding subtrees. The find operation runs in $O(\log n)$ time unconditionally, but insert and delete may take as long as $O(n)$ time, but the amortized time is $O(\log n)$.

**Beyond Dictionary Operations:** There are many other operations, beyond the standard ordered-dictionary operations (insert, delete, find) that a user might want the data structure to support. Here are a few examples:

**Order-statistic queries:** Given an integer $k$, where $1 \leq k \leq n$, find the $k$th smallest element in the set (see Fig. 77).

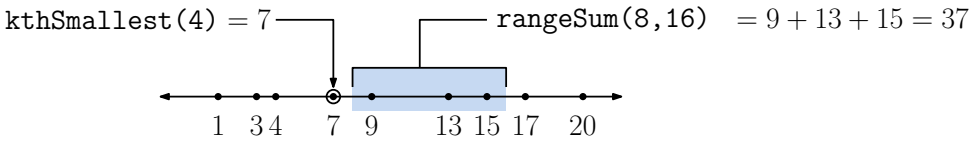



Fig. 77: Order-statistic and range queries.

**Range queries:** Given two keys $x_0$ and $x_1$, report (or generally compute some associative operation like sum or product over) all the values of the dictionary whose keys lie between $x_0$ and $x_1$ (see Fig. 77).

**Split/Merge:** Given a search tree $T$ and a key $x$, *split* $T$ into two search trees $T_1$ and $T_2$ such that all the keys of $T_1$ are $\leq x$ and all the keys of $T_2$ are $> x$ (see Fig. 78). Conversely, given two search trees $T_1$ and $T_2$ such that every key of $T_1$ is smaller than every key of $T_2$, *merge* them into a single search tree. The target is to solve both problems in $O(\log n)$ time.

**Finger-Search Queries:** We have just performed a find and located the node containing a key $x$. Now, we want to access a key $y$ that is close to $x$ in order. (E.g., you have a pointer to "Brenda" and now you want to find "Brandon.") Can this be done faster than restarting the search from the root?

**Beyond Worst-Case Complexity:** In the above data structures, our analysis was based on worst-case performance. In practice, some queries are more likely than others. If so, *expected-case performance* may be more appropriate.

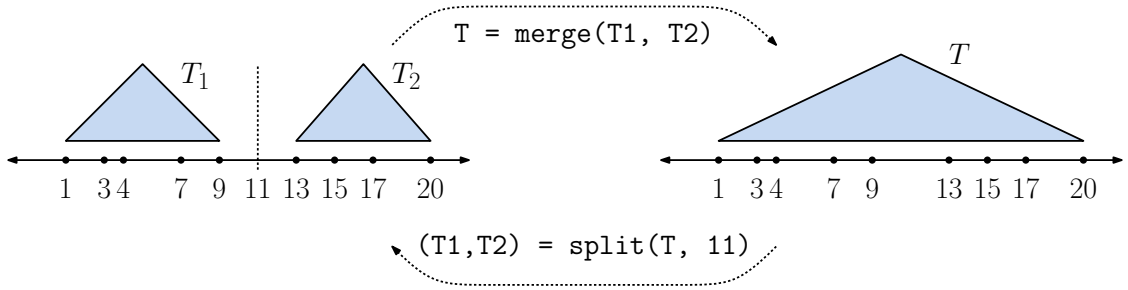



Fig. 78: Splitting and merging.

**Expected-Case Optimal Search Tree:** You are given $n$ keys, $\{x_1, \ldots, x_n\}$. Let $p_i$ denote the probability of accessing key $x_i$. Thus, $0 \leq p_i \leq 1$ and $\sum_{i=1}^{n} p_i = 1$. Suppose a binary search tree $T$ stores $x_i$ in a node at depth $d_i$ from the root. The *expected search time* for this tree is $E(T) = \sum_{i=1}^{n} p_i d_i$.

Given the $p_i$'s can we compute the binary search tree $T$ that minimizes $E(T)$?

The structures we have seen so far do not address this question. There does exist an efficient algorithm for computing the optimum binary-search tree. (The answer is yes, and it involves an interesting exercise in dynamic programming.)

However, in order to compute this data structure, we assume that we *know* what the access probabilities. What if they are not known? What if they are known at some time, but they change over time? In such a dynamic setting, a better solution would be a *self-adjusting tree*. This is a tree that dynamically adjusts its structure according to a dynamically changing set of access probabilities. Intuitively, keys that are frequently accessed will filter up near the root, and keys that are rarely accessed will slowly fall to the deeper levels of the search tree.

Achieving this goal in a manner that can be made theoretically rigorous is a challenging problem. It was solved by Danniel Sleator and Robert Tarjan in 1985 by a data structure, called a *splay tree*. (This is the same Tarjan of Fibonacci heaps and depth-first search.) The splay tree itself is an amazing idea. While the tree is provably (theoretically) optimal with respect to a number of different criteria, the efficiency comes at a cost. Individual operations may take a long time, and efficiency is in the *amortized* sense.

**Splay Trees and Amortization:** All the balanced binary tree structures we have seen so far have two things in common: (1) they use rotations to maintain structure and (2) each node stores additional information to allow the tree to maintain balance. A *splay tree* is a binary search tree, and it uses rotations to maintain its structure, but unlike the others no additional storage is needed for balance information. (Thus, each node is just a node of a standard binary search tree. It stores a key, value, left child, and right child. That is all!)

Because a splay tree has no balance information, it is possible for the tree to become very unbalanced. Splay trees are remarkable in that they are *self-adjusting*. Having nodes that are great depth in a binary tree is not a problem *per se*, until such an element is accessed. Splay trees employ a clever trick so that whenever a very deep node is accessed, the tree will restructure itself so that the tree becomes significantly more balanced. *The splay tree maintains itself in balance (on average), but it has no idea whether it is balanced or not!*

This means that, as with standard binary search trees, it is possible that a single access operation could take as long as $\Omega(n)$ time (and not the $O(\log n)$ that we would like). However,

splay trees are efficient in the amortized sense:

**Splay Tree Performance Bound:** Starting with an empty tree, the total time needed to perform any sequence of $m$ insert/delete/find operations on a splay tree is $O(m \log n)$, where $n$ is the maximum number of nodes in the tree.

Thus, although any individual operation may be quite costly (as high as $\Omega(n)$ time), the average cost of any operation is at most $O(\log n)$. As we have seen before, such an analysis (averaging over a sequence of operations) is called an *amortized analysis*. In the business world, amortization refers to the process of paying off a large payment over time in small installments. Here we are paying for the total running time of the data structure's algorithms over a sequence of operations in small installments. Even though each individual operation may be costly, the overall average is small.

As mentioned above, splay trees tend to bring frequently accessed keys up near the root. Indeed, it can be shown that if the access probabilities are stable, the cost of splay-tree operations asymptotically matches the performance of an optimal binary search tree.

No balance information, optimal expected search times, self-adjusting behavior? Splay trees sound amazing—and they are. However, they are not used that often in practice. In spite of their cool properties, they tend to be slower in practice than the other tree-based dictionary structures we have covered. So, these other structures tend to be used more often than splay trees.

**Splaying:** The key to any self-adjusting data structure is the operation that incrementally modifies the organization of objects in the structure. In the case of a splay tree, this operation is called *splaying*. Given a key value $x$ and a splay tree $T$ the operation `T.splay(x)` searches for the key $x$ within $T$, and reorganizes $T$ while rotating the node with key $x$ up to the root of the tree. If $x$ is not in the tree, either the inorder predecessor or inorder successor of $x$ will be brought to the root instead.

Here is how `T.splay(x)` works. We start with the normal binary search descent from the root of $T$ to find the node $p$ containing key $x$, or the last node visited before we fall out of the tree. (Observe that in the latter case, $p$ is either the inorder successor or predecessor of $x$, depending on whether we fell out along a `null` left child link or a `null` right child link. Our objective is to bring the node $p$ up to the root.

**An idea that doesn't work:** At this point you may see an obvious strategy to bring $p$ to the root. We walk up the tree to $p$'s ancestors, applying a rotation at each. While this will satisfy one of our requirements of moving $p$ to the root, it will not do a good job of reorganizing the tree. To see why, suppose that we attempt to apply rotations to node $a$ in Fig. 79. Observe that while $a$ is brought up to the root, the tree is still very skewed and unbalanced.

**A better idea:** The poor performance of the single-rotation method suggests that we try something that "stirs things up" a bit more. Our next idea is to go two nodes at a time and apply rotations at each of these nodes. For example, in Fig. 80, we see that by applying two rotations at a time, first at the grandparent and then at the parent has a dramatically better result on the tree height, cutting it roughly in half. (But will it work general?)

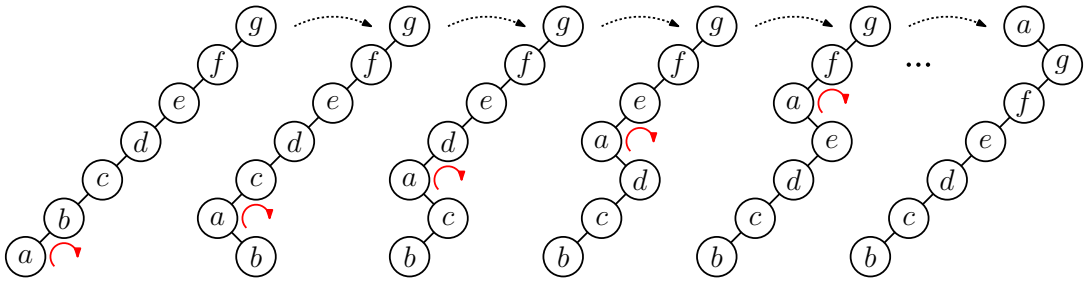


Fig. 79: Single rotations up to the root—the tree is still poorly balanced.

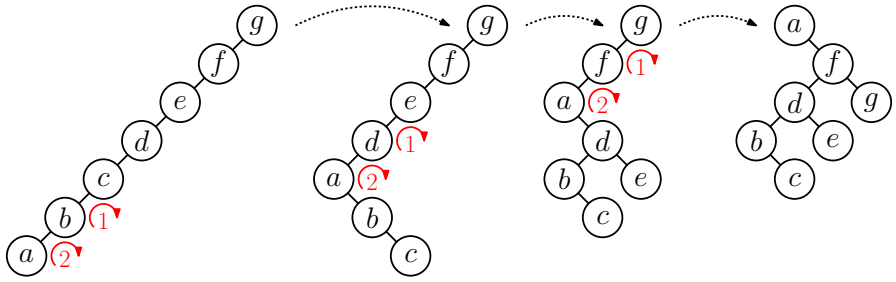


Fig. 80: Two-at-a-time rotations up to the root—the tree height is cut almost in half.

The above exercise suggests that we work two levels at a time. Here is a more formal description of the splay operation at a single node $p$ of the tree:

- If $p$ has both a parent and grandparent, $q$ and $r$ be the parent and grandparent, respectively. We consider two cases:
  - **Zig-zig:** If $p$ and $q$ are both right children or both left children, we apply a rotation at $r$ followed by a rotation at $q$, to bring $p$ to the top of this 3-node ensemble (see Fig. 81(a)), and continue up the tree.

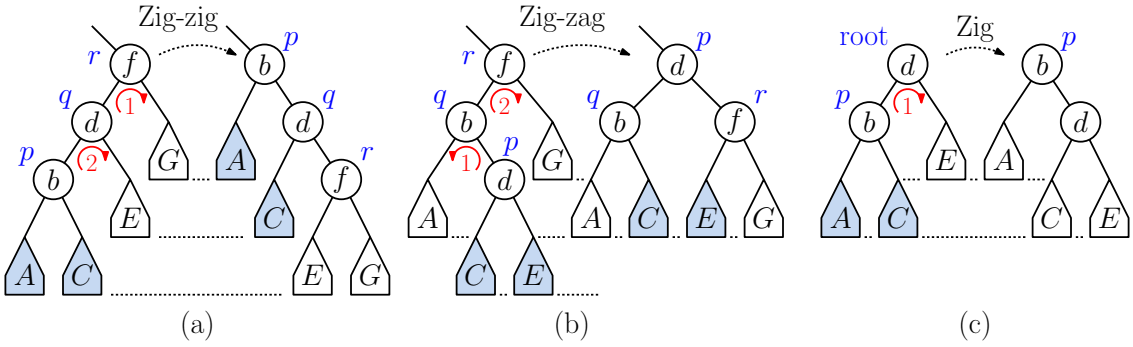



Fig. 81: Splaying cases: (a) Zig-Zig, (b) Zig-Zag and (c) Zig.

  - **Zig-zag:** If $p$ and $q$ are left-right or right-left children, we apply a rotation at $q$ followed by a rotation at $r$, to bring $p$ to the top of this 3-node ensemble (see Fig. 81(b)), and continue up the tree.
- **Zig:** If $p$ is the child of the root, we do a single rotation at the root of $T$, making $p$ the new root (see Fig. 81(a)), and are now done.

- If $p$ is the root of $T$, we are done.

A full example is shown in Fig. 82. Note that the tree's inorder structure is preserved. Also observe that nodes lying on or near the search path to $p$ (such as 1) tend to be lifted much closer to the root through this operation.

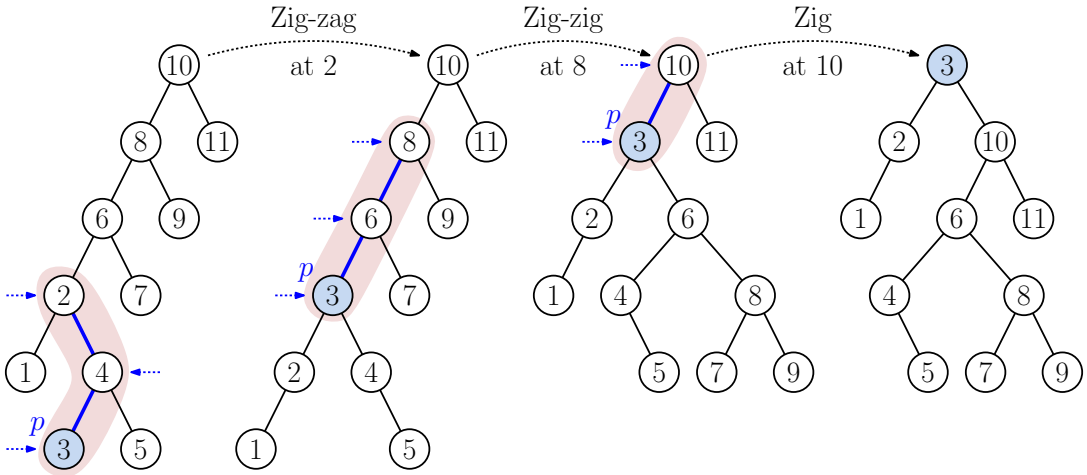



Fig. 82: The full splay operation on $p$.

**Why these rotations?** You might wonder why we performed these particular rotations in this particular order. The situation that we are most concerned about is where the tree is highly imbalanced and we repeatedly attempt to access elements that are unusually deep in the tree, say much deeper than $O(\log n)$ depth. Such an operation will be expensive. We want to be sure that we cannot repeat it many times.

Because we rotate as we are backing up the tree, we may assume that the key we sought was in one of $p$'s two children (shaded in blue in the figure). Observe in Fig. 81(a) and (b) that both of these subtrees are lifted up at least one level in the tree following the zig-zig or zig-zag rotation. Thus, if this were a long search path, then after repeating this operation all the way to the root, the nodes along this long path would be lifted up to roughly half of their original level. (The reason we say "halved" is that for every two levels we perform an operation that lifts each subtree up by at least one level.) *Thus, splaying has the desirable effect that it tends to significantly reduce the length of long search paths, whenever we attempt to access something within one of these long paths.*

You may protest at this point. What about the negative impact splaying has on the levels of other subtrees (like $G$ in Fig. 81(a))? This is true. Clearly, there must be winners and losers. But we the accessed node was in $p$'s subtree, and our principal concern is repeat visits deep in the tree cannot be allowed to repeat excessively. A certain amount of damage to the rest of the tree's structure is the price that we pay for this. But to make this convincing, you should read the full amortized proof. We will not cover it since it is quite mathematically involved.

**Implementation:** As mentioned above. The splay tree requires no balance information, levels, or colors. We assume each node has left, right, and parent links. A high-level description is provided in the code block below. The function is invoked as `root = splay(x)`.

**Splay Tree Operations:** Now that we know how to perform a single splay operation, how to we use this to perform the basic dictionary operations, insert, delete, and find? A key idea is to

Splay Operation

```
Node splay(Key x) {
    Node p = find x using standard binary tree search
    // if x is not present, then p is the inorder successor or predecessor
    while (p != root) {
         if (p is a child of the rooot)
           zig(p)
        else if (p is left-left or right-right grandchild)
           zig-zig(p)
        else /* p is left-right or right-left grandchild */
           zig-zag(p)
    }
    return p                                  // p is the new root
}
```

Rotation Utilities

```
void zig(Node p) {
    if (p == p.parent.left)                   // p is left child of root
        rotateRight(p.parent)                 // rotate p to be new root
    else                                      // p is right child of root
        rotateLeft(p.parent)
}

void zig-zig(Node p) {
    if (p == p.parent.parent.left.left) {     // left-left grandchild
        rotateRight(p.parent.parent)          // rotate grandparent first
        rotateRight(p.parent)                 // ... then parent
    } else {                                  // right-right grandchild
        rotateLeft(p.parent.parent)           // symmetrical
        rotateLeft(p.parent)

    }
}

void zig-zag(Node p) {
    if (p == p.parent.parent.left.right) {    // left-right grandchild
        rotateLeft(p.parent)                  // rotate parent first
        rotateRight(p.parent.parent)          // ... then grandparent
    } else {                                  // right-left grandchild
        rotateRight(p.parent)                 // symmetrical
        rotateLeft(p.parent.parent)
    }
}
```

use the splay operation to do the "heavy lifting" after which we perform a few constant-time operations.

`find(x)`: To find key $x$ in tree $T$, we simply invoke `T.splay(x)`. If $x$ is in the tree it will be transported to the root. If after the operation, the root key is not $x$, we know that $x$ is not in the dictionary.

This is a bit weird. In all our previous data structures, the `find` operation does not alter the tree structure. Why do it here? Recall that in the case of expected-case optimal trees, a small number of nodes may have very high access probabilities. The splaying operation has the tendency to keep these frequently accessed nodes nearest to the root.

Find in a Splay Tree

```
Value find(Key x) {
    root = splay(x)
    if (root.key == x) return x.value       // found it
    else return null                        // not found
}
```

`insert(x, v)`: To insert the key-value pair $(x, v)$, we first invoke `T.splay(x)`. If $x$ is already in the tree, it will be transported to the root, and we can take appropriate action (e.g., throw an exception). Otherwise, the root will consist of some key $y$ that is either the key immediately before $x$ or immediately after $x$ in $T$. Let us consider the former case ($y < x$), since the other case is symmetrical. Let $R$ denote the right subtree of the root. We know that all the keys in $R$ are greater than $x$ so we create a new root node containing $x$ and $v$, and we make $R$ its right subtree (see Fig. 83). The remaining nodes are hung off as the left subtree $L$ of this node.

Insert in a Splay Tree

```
void insert(Key x, Value v) {
    Node p = splay(x)                       // pull x's pred/succ to root
    if (p.key == x) Error! "Duplicate key"  // Oops! x is already here
    q = new Node(x, v)                      // new node for x
    if (p.key < x) {                        // p is predecessor?
        q.left  = p                         // put p to our left
        q.right = p.right                   // ... and p's right to our right
        p.right = null
    } else /* p.key > x */ { ... }          // symmetrical
    root = q                                // new root holds x
}
```

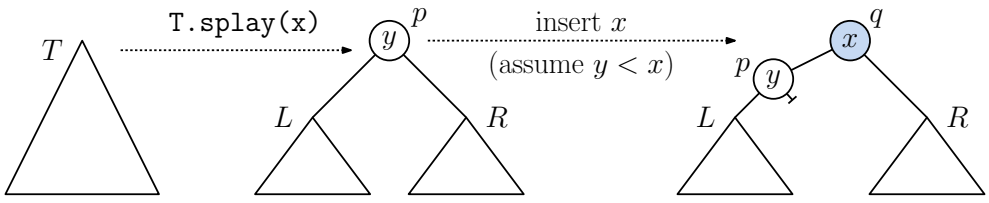



Fig. 83: Splay-tree insertion of $x$.

`delete(x)`: To delete $x$, we first invoke `T.splay(x)` to bring the deleted node to the root. If the root's key is not $x$, then $x$ is not in the tree, and we can take appropriate error action. Otherwise, let $L$ and $R$ be the left and right subtrees of the resulting tree (see Fig. 84). If $L$ is empty, then $x$ is the new smallest key in the tree. We remove $x$ and $R$ becomes the new tree. We can do the symmetrical thing if $R$ is empty.

Delete in a Splay Tree

```
void delete(Key x) {
    Node p = splay(x)                        // pull x to root
    if (p.key != x) Error! "Non-existant"    // Oops! x is not here
    p.right.splay(x)                         // splay x in p's right subtree
    Node q = p.right                         // q is p's inorder successor
    q.left = p.left                          // transfer left child to q
    root = q                                 // make q the new root
}
```

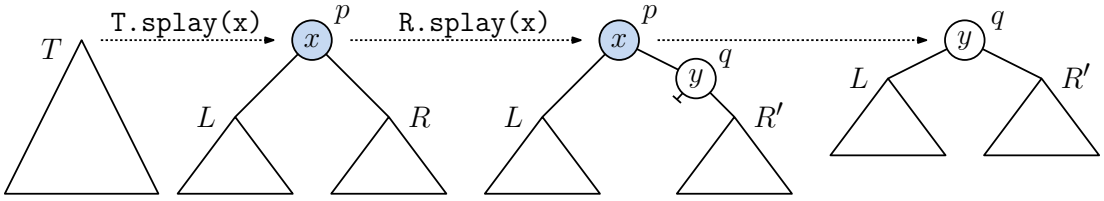



Fig. 84: Splay-tree deletion of $x$.

Otherwise, both subtrees are nonempty, and we next find an appropriate replacement node. To do this, we perform `R.splay(x)`. (`L.splay(x)` would work equally well). Huh? We know $x$ is not there. The reason is that we want to pull $x$'s preorder successor, call it $y$, up and make it a child of $x$. Since $y$ immediately follows $x$ in the node ordering, $y$'s left child must be null. To delete $x$, we make $y$ the new root of the tree and make $x$'s left child $L$ to be $y$'s left child. (Take time to convince yourself that this is a valid search tree!)

**Caveat:** The code given above intended to provide an intuitive understanding. We have shown how to update the left and right children, but we have neglected updating the parent links. We will leave this as an exercise.

**Why are Splay Trees Great?** Splay trees are amazing in the sense that they satisfy (at least theoretically) many optimality properties, which none of the other search trees that we have seen. Here is a partial list.

**Balance Theorem:** The cost of applying any sequence of $m$ accesses (insert, delete, find) on a splay tree with $n$ elements is $O(m \log n + n \log n)$

**Static Optimality Theorem:** Let $q_x$ denote the number of times that an element $x$ is accessed in a sequence of $m$ accesses to a splay tree. (Think of $q_x/m = p_x$ as an empirical measure of the probability of accessing $x$.) Then the cost of performing these accesses is

$$O\left(m + \sum_x q_x \log \frac{m}{q_x}\right).$$

The expression in the summation is the *entropy* of the access probability distribution, and this is a theoretical lower bound on the performance of *any* decision-based data structure.

**Static Finger Theorem:** Assume that the items are numbered 1 through $n$ in ascending order. Let $f$ be any fixed element (the "finger"). Then the cost of performing any sequence of operations is

$$O\left(m + n\log n + \sum_x \log(|x - f| + 1)\right).$$

Intuitively, this says that the cost of any operation is the logarithm of the number of elements between the accessed element and the finger.

**Dynamic Finger Theorem:** Assume that finger for each step in accessing an element $y$ is the location of element accessed in the previous step $x$. Then the cost of performing any sequence of operations is

$$O\left(m + n\log n + \sum_{x,y} \log(|y - x| + 1)\right).$$

Intuitively, this that if you start a search from some element $y$ to another element $x$, the cost of the operation is the log of the number of elements between them.

**Working-Set Theorem:** Each time an element $x$ is accessed, let $t(x)$ be the number of elements that were accessed since $x$'s last access. Then the cost of performing the sequence is

$$O\left(m + n\log n + \sum_x \log(t(x) + 1)\right).$$

Intuitively, this says that if we accessed an element $t$ steps ago, then the time to access it now is rougly $\log t$.

**Scanning Theorem:** If each element of a splay tree is accessed in ascending (or descending) order, the total time for all these accesses is $O(n)$. (A naive bound would be $O(n \log n)$.)

**Why Splay Trees Work:** (Optional) Sleator and Tarjan proved that, if you start with an empty tree and perform any sequence of $m$ splay tree operations (insert, delete, find), then the total running time will be $O(m \log n)$, where $n$ is the maximum number of elements in the tree at any time. Thus the average time per operation is $O(\log n)$, as we would like. Their amortized analysis involves a *potential-based argument*. We will sketch the idea without getting into the details.

The intuition is to associate a *potential* with any tree. Intuitively, the potential is a value that informs you how badly unbalanced the tree is. In financial terms, we think of potential as money in a bank account. As it accrues, we can use it to pay for the cost of rebalancing the tree. The argument relies on showing that no matter what sequence of operations occurs, there is always a nonnegative potential ("money in the bank"). The *amortized cost* of any operation is defined to be the sum of the actual cost of the operation (e.g., the number of rotations performed) and the change in potential. The objective is to show that the amortized cost of every operation is $O(\log n)$. Some operations may be very costly (e.g., splaying along a path of length $n$), but the resulting decrease in the tree's potential will be large enough to compensate for this.

So, what is the potential function used for proving splay trees have good amortized performance? First, for each node $p$ of the tree, define size($p$) to be the number of nodes in the subtree rooted at $p$. Define $\text{rank}(p) = \lg \text{size}(p)$. Intuitively, the rank of a node is the "ideal height" of this subtree in a perfectly balanced tree. The potential function of a tree is defined to be

$$\Phi(T) \;=\; \sum_{p \in T} \text{rank}(p) \;=\; \sum_{p \in T} \lg \text{size}(p).$$

The following is the key to the analysis. It bounds the amortized cost of each rotation operation.

**Rotation Lemma:** Given any node $p$, let $\text{rank}(p)$ and $\text{rank}'(p)$ denote its rank before and after applying a rotation step. The amortized cost of a zig rotation at $p$ is at most $1 + 3(\text{rank}'(p) - \text{rank}(p))$, and the amortized cost of a zig-zig or zig-zag rotation at any node $p$ is at most $3(\text{rank}'(p) - \text{rank}(p))$.

**(Partial) Proof:** We will only prove the case of the zig-zig rotation. The other cases follow by a similar sort of derivation, which we leave as an exercise.

To simplify notation, let's write $\text{rank}(x)$ as $r(x)$ and $\text{size}(x)$ and $s(x)$. Suppose that we perform a zig-zig rotation involving three nodes $x$, $y$, and $z$, as shown in Fig. 81(a) (where $x$, $y$, and $z$ play the roles of $p$, $q$, and $r$, respectively). Let $r(x)$, $r(y)$, and $r(z)$ denote the ranks of these items before the rotation, and let $r'(x)$, $r'(y)$, and $r'(z)$ denote these ranks after the rotation. The actual cost for the zig-zig is 2 rotations, and since these are the only changes made in the tree, the change in potential is $\Delta\Phi = (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z))$. Thus, the amortized cost of this operation is

$$A \;=\; 2 + \Delta\Phi \;=\; 2 + (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z)).$$

Rearranging terms we have

$$A \;=\; 2 + (r'(x) - r(z)) + r'(y) + r'(z) - r(x) - r(y).$$

Observe that after the rotation $x$'s subtree is the same as $z$'s subtree before the rotation, so $r'(x) = r(z)$. Also, observe that before the rotation $y$'s subtree contains $x$'s subtree, so $r(y) \geq r(x)$ or equivalently $-r(y) \leq -r(x)$. After the rotation $y$'s subtree is contained in $x$'s subtree, so $r'(y) \leq r'(x)$. Thus, we have

$$A \;\leq\; 2 + 0 + r'(x) + r'(z) - r(x) - r(x) \;=\; 2 + r'(x) + r'(z) - 2r(x).$$

Next, we will employ an observation about the logarithm function. It is a concave function, which implies that for any $a$ and $b$, $(\lg a + \lg b)/2 \leq \lg((a+b)/2)$. Also, observe that $s(x) + s'(z) \leq s'(x)$. Using these and the fact that $r(x) = \lg s(x)$, we have

$$\begin{aligned} \frac{r(x) + r'(z)}{2} \;&=\; \frac{\lg s(x) + \lg s'(z)}{2} \;\leq\; \lg \frac{s(x) + s'(z)}{2} \\ &\leq\; \lg \frac{s'(x)}{2} \;=\; (\lg s'(x)) - 1 \;=\; r'(x) - 1. \end{aligned}$$

This implies that $r'(z) \leq 2r'(x) - r(x) - 2$. Plugging this in to our expression for $A$, we have

$$A \;\leq\; 2 + r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \;\leq\; 3r'(x) - 3r(x) \;\leq\; 3(r'(x) - r(x)),$$

which is what we set out to prove. Whew!

It is rather difficult to see what the benefit of this symbol manipulation is, but the key is that we have completely eliminated the +2 term in the zig-zig and zig-zag cases. This means that we can perform any number of these and the only terms that accumulate will be of the form $r'(x) - r(x)$, which is just the potential change. By applying this all the way up the rotation path, we obtain a telescoping series (and a final +1 for the last zig rotation). This implies the following:

**Splay Lemma:** The amortized cost of a `T.splay(p)` is at most $1 + 3(\text{rank}(\text{root}) - \text{rank}(p))$.

Since the rank of the root cannot exceed $\lg n$ and the rank of $p$ is nonnegative, we immediately have:

**Corollary:** The amortized cost of `T.splay(p)` is $O(\log n)$.

Finally, since each dictionary operation involves a constant number of splaying operations, we obtain the final result.

**Theorem:** The amortized cost of each dictionary operation in a splay tree is $O(\log n)$.

Wow! That is one impressive bit of data structure analysis. You are not responsible for knowing it, but this is great example of how amortized analyses are performed.

## Lecture 14: Randomized Search Structures: Skip Lists

**More Randomized Search Structures:** In this lecture we discuss a randomized approach to designing efficient data structures. A *randomized data structure* uses a random number generator to guide its operations and structure. The running time for any given input is expressed in terms of the *expected* running time, which is averaged over all possible outcomes of the algorithm's random choices.

Note that unlike the expected-case performance of unbalanced binary search trees, which depended on the order of operations, the expectation here is independent of what the user does, only on the random number generator. Since the random number generator is not under the control of the data structure's user, even a malicious user cannot engineer a sequence of operations that will guarantee a higher than expected running time. The only thing that could go wrong is having bad luck with respect to the random number generator.

In this lecture we will discuss a data structure that is reputed to be among the fastest data structures for ordered dictionaries, called the skip list. (It also has the feature that it was designed by Bill Pugh, a former professor at the University of Maryland!)

**Skip Lists:** Skip lists began with the idea, "how can we make sorted linked lists better?" It is easy to do operations like insertion and deletion into linked lists, but it is costly to locate items efficiently because we have to walk through the list one item at a time. If we could "skip" over multiple of items at a time, however, then we could perform searches efficiently. Intuitively, a skip list is a data structure that encodes a collection of sorted linked lists, where links skip over 2, then 4, then 8, and so on, elements with each link.

To make this more concrete, imagine a linked list, sorted by keys. There are two nodes at either end of the list, called `head` and `tail`. Take every other entry of this linked list (say the even numbered entries) and extend it up to a new linked list with 1/2 as many entries. Now take every other entry of this linked list and extend it up to another linked with 1/4 as

many entries as the original list, and continue this until no elements remain. The `head` and `tail` nodes are always lifted (see Fig. 85). Clearly, we can repeat this process $\lceil \lg n \rceil$ times. (Recall that "lg" denotes log base 2.)

The result is something that we will call the *ideal skip list*. Unlike the standard linked list, where you may need to traverse $O(n)$ links to reach a given node, in this list *any* node can be reached with $O(\log n)$ links from the `head`.

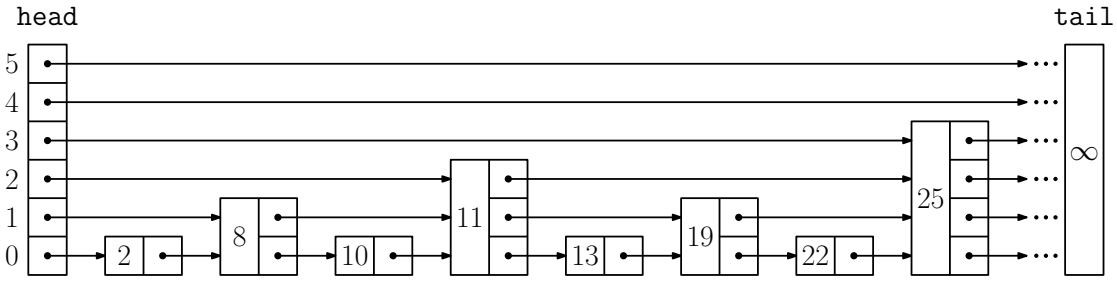



Fig. 85: The "ideal" skip list.

The philosophy in searching/processing is to operate at the highest level possible and drop down only when needed. To perform the operation `find(x)`, we start at the highest level of `head`. We scan linearly along the list at the current level $i$, until we are about to jump to an node whose key value is strictly greater than to $x$. Since `tail` is associated with $\infty$, we will always succeed in finding such a node. Let $p$ point to the node just before this step. If $p$'s data value is equal to $x$ then we stop. Otherwise, if we are not yet at the lowest level, we descend to the next lower level $i - 1$ and continue the search there. Finally, if we are at the lowest level and have not found $x$, we announce that the $x$ is not in the list (see Fig. 86).

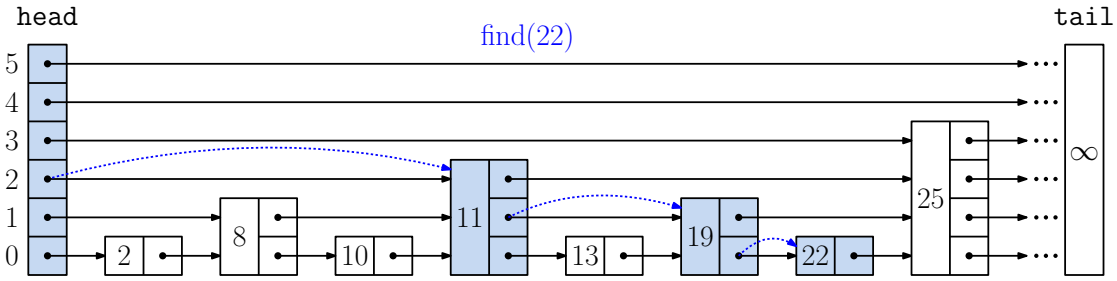



Fig. 86: Searching the ideal skip list.

How long would this search require in the worst case? Observe that we need never traverse more than one link at any given level in the path to the desired node. (Convince yourself that this is true.) We will generally need to access two nodes at each level, however, because the need to determine the node whose key is greater than $x$'s. As mentioned earlier, the number of levels is $\lceil \lg n \rceil$. Therefore, the total number of nodes accessed is at most $2 \lceil \lg n \rceil = O(\log n)$.

**Randomized Skip Lists:** Unfortunately, like a perfectly balanced binary tree, the ideal skip list is too pure for use as a dynamic data structure. As soon as a single node was added to the middle of the lists, all the heights following it would need to be modified. But we can relax this requirement to achieve an efficient data structure. In the ideal skip list, every other node from level $i$ is extended up to level $i + 1$. Instead, how about if we did this *randomly*?

Following the "ideal" skip list, we would like roughly *half* of the entries at any given level to survive and contribute to the next higher one. We can control this through a random process.

Suppose that we have built the skip list up to some level $i$, and we want to extend this to level $i+1$. Imagine every node at level $i$ tossing a coin. If the coin comes up heads (with probability 1/2) this node promotes itself to level $i+1$, and otherwise it stops here. By the nature of randomization, the expected number of nodes at level $i+1$ will be half the number of nodes at level $i$. Thus, the expected number of nodes at any level $i$ will be $n/2^i$, which means that the expected number of nodes at level $\lceil \lg n \rceil$ is a constant. Fig. 87 shows what such a *randomized skip list*, or simply *skip list*, might look like.

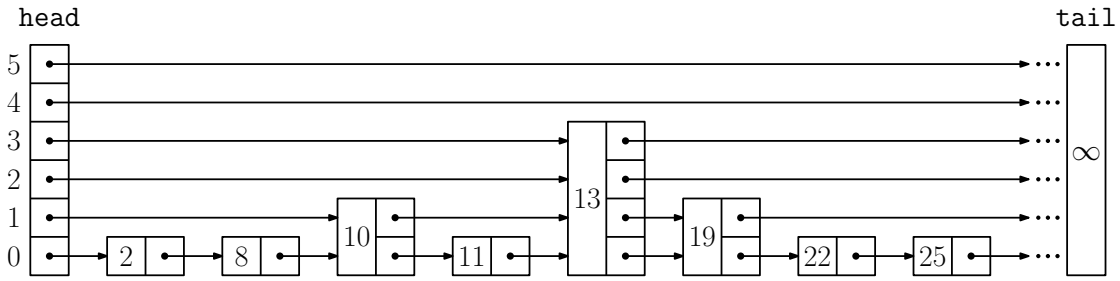



Fig. 87: A (randomized) skip list.

**Implementation:** As we saw with 2-3 and 2-3-4 trees, skip lists have variable sized nodes. In the skip list, there is no a priori limit on a node's size. This is not an issue in languages like Java, where it is easy to dynamically allocate arrays of arbitrary length, however. For example, our skip list node might have the following structure

Node in a Skip List

```
class SkipNode {
    Key key             // key (for searching)
    Value value         // value (used by the application)
    SkipNode[] next     // array of next pointers (depending on level)

    SkipNode(Key x, Value v, int size) {  // constructor (given node size)
        key = x; value = v; next = new SkipNode[size];
    }
}
```

Here is sample code to find a key in a Skip list. Notice it is very simple (and does not involve recursion!) Intuitively, we start at the topmost nonempty level of the skip list, and keep moving along a given level as long as the key in the next node is less than or equal to the key we are looking form. Whenever we get stuck, we drop down a level. When we drop out of the skip list (level is less than zero), we know that the current node's key is at most $x$ and the next node's key is strictly greater than $x$. So, we check this node's key matches, and if so, we have found it. If so, we drop down a level. We stop when we fall below level zero. If the key is in the skip list, the last node visited will match. Otherwise, the find fails.

**Height and Space Analysis:** Unlike binary search trees whose nodes are all of the same size, the nodes of a skip list have variable sizes. How many levels will the skip list have? The process we described earlier could create an arbitrarily large number of levels (if your coin

Find in a Skip List

```
Value find(Key x) {
    int i = topmostLevel                          // start at topmost nonempty level
    SkipNode p = head                             // start at head node
    while (i >= 0) {                              // while levels remain
        if (p.next[i].key <= x) p = p.next[i]     // advance along same level
        else i--                                  // drop down a level
    }
    return (p.key == x ? p.value : null)          // return value if found
}
```

keeps coming up heads). We can show that the maximum level in the skip list will be $O(\log n)$ with high probability.

**Theorem:** The expected number of levels in a skip list with $n$ entries is $O(\log n)$. Moreover, for any $c \geq 1$, the probability that the number of levels exceeds $c \lg n$ is at most $1/n^{c-1}$.

**Proof:** The intuitive explanation for the expected bound is easy: With every new level, we cut the number of remaining entries (in expectation) by half. After doing this $\lg n$ times, we would expect there to be only one item left.

Let's consider the probability of exceeding the "ideal" value of $\lg n$ by a factor of $c \geq 1$. Suppose that we have $n$ entries and we observe that our skip list contains at least one node at level $\ell$. In order for an arbitrary entry to make it to level $\ell$, its coin toss came up heads at least $\ell$ times consecutively. The probability of this happening is clearly at most $1/2^\ell$. Since this holds independently for all of the $n$ entries, the probability that any of the entries survives to level $\ell$ is at most the $n$-fold sum $\frac{1}{2^\ell} + \frac{1}{2^\ell} + \frac{1}{2^\ell} + \cdots = \frac{n}{2^\ell}$.

Setting $\ell = c \cdot \lg n$, it follows that the probability that the maximum level exceeds $\ell$ is at most

$$\frac{n}{2^\ell} = \frac{n}{2^{c \lg n}} = \frac{n}{(2^{\lg n})^c} = \frac{n}{n^c} = \frac{1}{n^{c-1}}.$$

Observe that if we take even moderate values of $c$ and $n$, say $c = 3$ and $n = 1000$, the probability that the maximum height exceeds $3 \lg n$ is at most $1/n^2 = 1/1,000,000$, that is, one chance in a million. The chances of exceeding it by a factor of 5 is less than one in a trillion!

This implies that, if you have an upper bound on the value of $n$, you are safe (with high probability) to allocate head and tail arrays of size $\lceil 3 \lg n \rceil$. Of course, even if a node were to attempt to exceed this level, you could just arbitrarily limit it. The asymptotic analysis that follows would not change as a result.

Next, let's consider the storage space. Under the assumption that the maximum number of levels is $O(\log n)$, then in the worst case every node contributes to every level, and the skip list would have total storage of $O(n \log n)$. This is too high! The following lemma indicates that the expected space used is just linear in $n$.

**Theorem:** The expected storage for a skip list with $n$ entries is $O(n)$.

**Proof:** For $i \geq 0$, let $n_i$ denote the number of nodes that contribute to level $i$ of the skip list. This is a *random variable*, meaning that its value varies with our random choices. The quantity that we are interested in is its expected value $E(n_i)$. The probability that

an arbitrary entry contributes to level $i$, is just the probability of tossing $i$ consecutive heads using a fair coin, which is clearly $1/2^i$. Thus, the expected number of entries that survive to this level is just the total population size ($n$) times the probability of contributing ($1/2^i$). Thus, $E(n_i) = n/2^i$.

Let $L$ denote the maximum level in our skip list. By basic facts on the geometric series, we know that $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2$, and hence

$$\sum_{i=0}^{L} E(n_i) \;\leq\; \sum_{i=0}^{\infty} E(n_i) \;=\; \sum_{i=0}^{\infty} \frac{n}{2^i} \;=\; n \sum_{i=0}^{\infty} \frac{1}{2^i} \;=\; 2n.$$

Finally, by the linearity of expectation (the expected value of a sum of random variables is the sum of the expected values of the random variables), it follows that the total expected storage needed is the sum of $E(n_i)$ over all possible levels

$$E\left(\sum_{i=0}^{L} n_i\right) \;\leq\; E\left(\sum_{i=0}^{\infty} n_i\right) \;=\; \sum_{i=0}^{\infty} E(n_i) \;=\; 2n \;=\; O(n),$$

and this completes the proof.

The above result may be a bit surprising. (After all, if the maximum level is $O(\log n)$, then shouldn't the space be $O(n \log n)$?) The key is that the vast majority of entries never make it beyond the first few levels. For example, less than 10% make it beyond level 4, and less than 1% make it beyond level 7.

**Search-Time Analysis:** Earlier, we argued that the worst-case search time in an ideal skip list is $O(\log n)$. Now, we will show that the *expected case* search time in the randomized skip list will be $O(\log n)$. It is important to note that the analysis to follow will *not* depend on the choice of keys in the data structure nor the order in which they were inserted. Rather, it will depend solely on the randomized (coin-flipping) process used to build the data structure.

The analysis of skip lists is an example of a *probabilistic analysis*. As observed earlier, the expected number of levels in a skip list is $O(\log n)$. We will show that for any fixed node, the length of the search path leading here is $O(\log n)$ in expectation. Our analysis will be based on walking backwards along the search path. (This is sometimes called a *backwards analysis*.)

Observe that the forward search path drops down a level whenever the next link would have taken us "beyond" the node we are searching for. Thus, when we consider the reversed search path, it will always take a step up if it can (i.e., if the node it is visiting contributes to the next higher level), otherwise it will take a step to the left.

**Theorem:** The expected number of nodes visited in a search in a skip list of $n$ keys is $O(\log n)$.

**Proof:** We will prove this for the more general case, where the probability that a node is promoted to the next higher level is $p$, for some constant $0 < p < 1$. The analysis for our coin-flipping version of the skip follows by setting $p = 1/2$.

For $0 \leq i \leq O(\log n)$, let $E(i)$ denote the expected number of nodes visited in the skip list at the *top* $i$ levels of the skip list. (For example, in Fig. 88, the skip list's top level is 5. In this case $E(2)$ would be the expected number of steps taken at levels 4 and 5, $E(3)$ would be the expected time spent in levels 3–5, and $E(6)$ would be the expected number of steps at all the levels.)

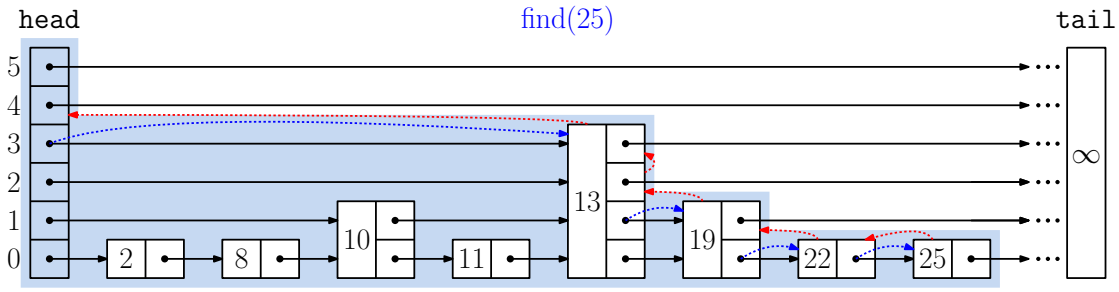



Fig. 88: The search path (blue) to $x = 25$ and the reverse search path (red).

As mentioned above, our analysis is based on walking backwards along the search path. Whenever we arrive at some node of level $i$, the probability that it contributes to the next higher level is exactly $p$, and if so, the search path came from above. On the other hand, with the remaining probability $1 - p$, this node stopped at this level, and the backwards search stays at the same level. Counting the current node we are visiting $(+1)$, we can express $E(i)$ by the recurrence:

$$E(i) \;=\; 1 + p \cdot E(i-1) + (1-p)E(i).$$

This is a weird recurrence, because we are defining $E(i)$ in terms of itself! But this is not circular, since each subsequent invocation occurs with a lower probability value, and so this converges. In particular, with a bit of algebra, we have:

$$E(i) \;=\; \frac{1}{p} + E(i-1).$$

By expansion, it is easy to verify that $E(i) = \frac{i}{p}$. Thus, if our skip list has $\ell$ levels, the expected search time is $E(\ell) = \ell/p$. By our assumption that $p$ is a constant, we have $E(\ell) = O(\ell)$. By our earlier theorem on the maximum number of levels, we know that $\ell = O(\log n)$. Therefore, the expected search time for skip lists is $O(\log n)$.

Clearly, there is more math involved in establishing expected-case bounds for randomized data structures. (This is the bane of students of data structures!) The payoff is that these data structures are usually faster than traditional structures because they are so simple. There is no need for rotating nodes, merging or splitting nodes, skewing and splitting, as we have seen with tree-based data structures.

**Insertion and Deletion:** Insertion into a skip list is almost as easy as insertion into a standard linked list. Given a key $x$ to insert, we first do a search on key $x$ to find its immediate predecessors in the skip list at each level of the structure. Next, we create a new node $x$. To determine the height of this node, we toss a coin repeatedly until it comes up tails. (More practically, we generate a random number until its parity is odd.) Letting $k$ denote the number of tosses needed, we create a node who height is the minimum of $k + 1$ and the maximum height of the skip list. We then link this node in to its $k + 1$ lowest predecessors in the list (see Fig. 89).

Deletion is quite similar. Again, we search for the node containing the key to delete, and we keep track of all its predecessors at various levels in the skip list (see Fig. 90). On finding it we unlink the node from each level, exactly as we would do in a standard linked-list deletion. Both operations take $O(\log n)$ time in expectation.

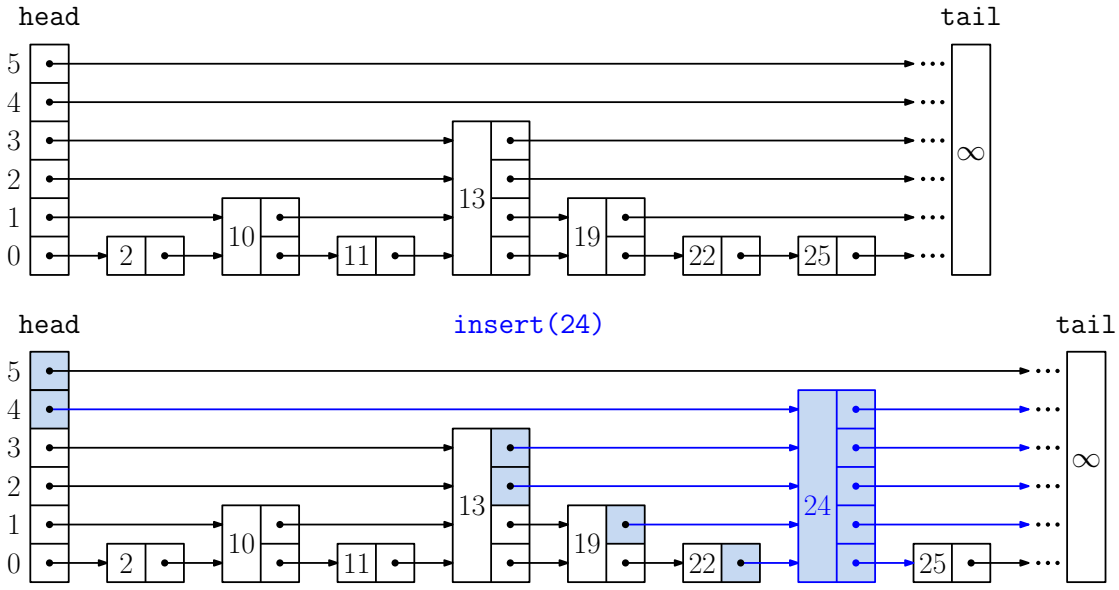



Fig. 89: Inserting a new key 24.

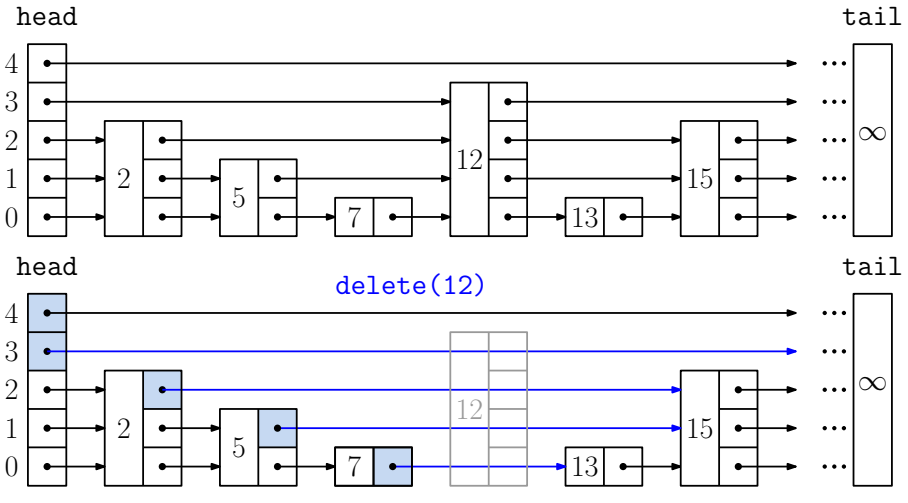



Fig. 90: Deleting key 12.

**Implementation Notes:** One of the appeals of skip lists is their ease of implementation. Most of procedures that operate on skip lists make use of the same simple code that is used for linked-list operations. One additional element is that you need to keep track of the level that you are currently on when performing searching.

Skip-list nodes have variable size, which is a bit unusual. This is not a problem in programming languages like Java that allow us to dynamically allocated arrays of variable size. Thus, each node of the skip list will generally contain the key-value pair associated with this entry, a variable-sized array of `next` pointers (so that `p.next[i]` points to the next node in the skip list from node `p` at level $i$). Finally, the structure has two special "sentinel nodes, " `head` and `tail`. We assume that `tail.key` is set to some incredibly large value so that searches always stop here.

**Overall Performance:** From a practical perspective, skip lists can do pretty much everything that standard binary trees structures can do. In expectation, they require $O(n)$ storage space, and all dictionary operations can be performed in time $O(\log n)$ in expectation. Given their simple linear structure, they are arguably easier to visualize and program. Experimental studies show that skip lists are among the fastest data structures for sorted dictionaries (with treaps). This is largely because the power of randomization keeps us from having to maintain more complex balance information, and thus simplifies the code and processing.

## Lecture 15: Hashing

**Hashing:** We have seen various data structures (e.g., binary trees, AVL trees, splay trees, skip lists) that can perform the dictionary operations `insert()`, `delete()` and `find()`. We know that these data structures provide $O(\log n)$ time access. It is unreasonable to expect any type of comparison-based structure to do better than this in the worst case. Using binary decisions, there is a lower bound of $\Omega(\log n)$ (and more precisely, $1 + \lfloor \lg n \rfloor$) on the worst case search time.

Remarkably, there is a better method, assuming that we are willing to give up on the idea of using comparisons to locate keys. The best known method is called *hashing*. Hashing and its variants support all the dictionary operations in $O(1)$ (i.e. constant) expected time. Hashing is considered so good, that in contexts where just these operations are being performed, hashing is the *method of choice*.

**Unordered vs. Ordered Dictionary:** We pay a price for this speed advantage. Operations based on the totally ordering of keys *cannot* be answered efficiently. (E.g., find the $k$th smallest key, or find the next key larger than $x$, or count the number keys that lie in the interval $[a, b]$.) All of the dictionary data structures we have seen so far are examples of the *ordered dictionary ADT*. This is generally true for data structures that find keys based on comparisons. In contrast, hashing implements the more limited (unordered) *dictionary ADT*. This means we can only perform *exact* look-ups in the dictionary.

**Overview:** The idea behind hashing is very simple. We have a table of given size $m$, called the *table size*. We will assume that $m$ is at least a small constant factor larger $n$. (As we shall see, when $m$ gets close to $n$, the running time slows down considerably. Making $m$ much larger than $n$ does not improve the running time by much and wastes space.) We select a *hash function* $h(x)$, which is an easily computable function that maps a key $x$ to a "random-like"

index in the range `[0..m-1]`. We then attempt to store $x$ (and its associated value) in index $h(x)$ in the table.

Of course, it may be that different keys are mapped to the same location. Such events are called *collisions*, and a key element in the design of a good hashing system how collisions are to be handled. If the table size is large (relative to the total number of entries) and the hashing function has been well designed, collisions should be relatively rare.

**Content-Addressable Storage:** Hashing is quite a versatile technique. One way to think about hashing is as a means of implementing a *content-addressable array*. We know that arrays can be addressed by an integer index. But it is often convenient to have a look-up table in which the elements are addressed by a key value which may be of any discrete type, strings for example or integers that are over such a large range of values that devising an array of this size would be impractical.

For example, suppose that in a mapping application, I want to compute the travel distance between two cities. Rather than running Dijkstra's algorithm, I could just precompute and store the distances between the largest 10,000 cities in the country. While this would take a lot of space, compute times would be super fast! Note that hashing is not usually appropriate for search based on real-valued data. Very similar keys, like 3.14159 and 3.14158, may be mapped to entirely different locations by the hash function.

There are two important issues that need to be addressed in the design of any hashing system, the *hash function* and the method of *collision resolution*. Let's discuss each of these in turn.

**Hash Functions:** A good hashing function should have the following properties:

- **Efficiently computable**: Ideally in constant time using simple arithmetic operations.
- **Avoids collisions**: Two different keys (even if very similar) should have a very low probability of colliding. To achieve this, the hash function should:
  - Involve *every bit* of the key (otherwise keys that differ only in these bits will collide)
  - Scatter naturally occurring *clusters* of key values.

As an example of the last rule, observe that in writing programs it is not uncommon to use very similar variables names, "`temp1`", "`temp2`", and "`temp3`". It is important such similar names be mapped to very different locations in the *hash output space*. By the way, the origin of the name "hashing" is from this mixing aspect of hash functions (thinking of "hash" in food preparation as a mixture of things).

We will think of hash functions as being applied to *nonnegative integer keys*. Keys that are not integers will generally need to be converted into this form (e.g., by converting the key into a bit string, such as an ASCII or Unicode representation of a string) and then interpreting the bit string as an integer. Since the hash function's output is the range $[0..m-1]$, an obvious (but not very good) choice for a hash function is:

$$h(x) \;=\; x \bmod m.$$

This is called *division hashing*. It satisfies our first criteria of efficiency, but consecutive keys are mapped to consecutive entries, and this is does not do a good job of breaking up clusters.

**Some Common Hash Functions:** Many different hash functions have been proposed. The topic is quite deep, and we will not claim to have a definitive answer for the best hash function. Here are three simple, commonly used hash functions:

**Multiplicative Hash Function:** Uses the hash function

$$h(x) \;=\; (ax) \bmod m,$$

where $a$ is a *large prime number* (or at least, sharing no common factors with $m$).

**Linear Hash Function:** Enhances the multiplicative hash function with an added constant term

$$h(x) \;=\; (ax + b) \bmod m.$$

**Polynomial Hash Function:** We can further extend the linear hash function to a polynomial. This is often handy with keys that consist of a sequence of objects, such as strings or the coordinates of points in a multi-dimensional space.

Suppose that the key being hashed involves a sequence of numbers $x = (c_0, c_1, \ldots, c_{k-1})$. We map them to a single number by computing a polynomial function whose coefficients are these values.

$$h(x) \;=\; h(c_0, \ldots, c_{k-1}) \;=\; \left(\sum_{i=0}^{k-1} c_i p^i\right) \bmod m$$

For example, if $k = 4$ and $p = 37$, and $x$ is the sequence $x = (c_0, \ldots, c_3)$, the associated polynomial would be $c_0 + c_1 37 + c_2 37^2 + c_3 37^3$.

**Collision Resolution:** We have discussed how to design a hash function in order to achieve good scattering properties. But, given even the best hash function, it is possible that distinct keys can map to the same location, that is, $h(x) = h(y)$, even though $x \neq y$. Such events are called *collisions*, and a fundamental aspect in the design of a good hashing system how collisions are handled. We focus on this aspect of hashing in this lecture, called *collision resolution*.

**Separate Chaining:** If we have additional memory at our disposal, a simple approach to collision resolution, called *separate chaining*, is to store the colliding entries in a separate linked list, one for each table entry. More formally, each table entry stores a reference to a list data structure that contains all the dictionary entries that hash to this location.

To make this more concrete, let $h$ be the hash function, and let `table[]` be an $m$-element array, such that each element `table[i]` is a linked list containing the key-value pairs $(x, v)$, such that $h(x) = i$. We will set the value of $m$ so that each linked list is expected to contain just a constant number of entries, so there is no need to be clever by trying to sort the elements of the list. The dictionary operations reduce to applying the associated linked-list operation on the appropriate entry of the hash table.

- `insert(x,v)`: Compute `i = h(x)`, and then invoke `table[i].insert(x,v)` to insert $(x, v)$ into the associated linked list. If $x$ is already in the list, signal a duplicate-key error (see Fig. 91).
- `delete(x)`: Compute `i = h(x)`, and then invoke `table[i].delete(x)` to remove $x$'s entry from the associated linked list. If $x$ is not in the list, signal a missing-key error.
- `find(x)`: Compute `i = h(x)`, and then invoke `table[i].find(x)` to determine (by simple brute-force search) whether $x$ is in the list.

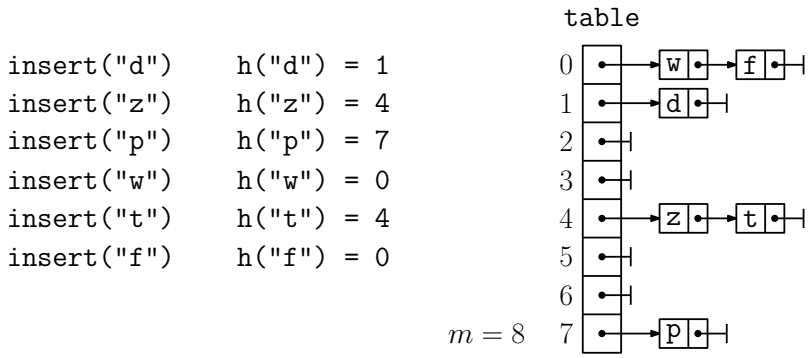



Fig. 91: Collision resolution by separate chaining.

Clearly, the running time of this procedure depends on the number of entries that are stored in the given table entry. To get a handle on this, consider a hash table of size $m$ containing $n$ keys. Define its *load factor* to be $\lambda = n/m$. If we assume that our hash function has done a good job of scattering keys uniformly about the table entries, it follows that the expected number of entries in each list is $\lambda$.

**Performance:** We say that a search `find(x)` is *successful* if $x$ is in the table, and otherwise it is *unsuccessful*. Assuming that the entries appear in each linked list in random order, we would expect that we need to search roughly half the list before finding the item being sought after. It follows that the expected running time of a successful search with separate chaining is roughly $1 + \lambda/2$. (The initial "+1" accounts for the fact that we need to check one more entry than the list contains, if just to check the `null` pointer at the end of the list.) On the other hand, if the search is unsuccessful, we need to enumerate the entire list, and so the expected running time of an unsuccessful search with separate chaining is roughly $1 + \lambda$. In summary, the successful and unsuccessful search times for separate chaining are:

$$S_{\text{SC}} \;=\; 1 + \frac{\lambda}{2} \qquad U_{\text{SC}} \;=\; 1 + \lambda,$$

Observe that both are $O(1)$ under our assumption that $\lambda$ is $O(1)$. Since we can insert and delete into a linked list in constant time, it follows that the expected time for all dictionary operations is $O(1 + \lambda)$.

Note the "in expectation" condition is not based on any assumptions about the insertion or deletion order. It depends simply on the assumption that the hash function uniformly scatters the keys. (There are methods, such as universal hashing, which can guarantee this with high probability.) It has been borne out through many empirical studies that hashing is indeed very efficient.

The principal drawback of separate chaining is that additional storage is required for linked-list pointers. It would be nice to avoid this additional wasted space. The remaining methods that we will discuss have this property. Later in the lecture we will discuss *rehashing* as a mechanism for controlling the load factor.

**Open Addressing:** Let us return to the question of collision-resolution methods that do not require additional storage. Our objective is to store all the keys within the hash table. (Therefore, we will need to assume that the load factor is never greater than 1.) To know

which table entries store a value and which do not, we will store a special value, called `empty`, in the empty table entries. The value of `empty` must be such that it matches no valid key.

Whenever we attempt to insert a new entry and find that its position is already occupied, we will begin probing other table entries until we discover an empty location where we can place the new key. In it most general form, an open addressing system involves a secondary search function, $f$. If we discover that location $h(x)$ is occupied, we next try locations

$$(h(x) + f(1)) \bmod m,\ (h(x) + f(2)) \bmod m,\ (h(x) + f(3)) \bmod m, \ldots.$$

until finding an open location. (To make this a bit more elegant, let us assume that $f(0) = 0$, so even the first probe fits within the general pattern.) This is called a *probe sequence*, and ideally it should be capable of searching the entire list. How is this function $f$ chosen? There are a number of alternatives, which we consider below.

**Linear Probing:** The simplest idea is to simply search sequential locations until finding one that is open. In other words, the probe function is $f(i) = i$. Although this approach is very simple, it only works well for fairly small load factor. As the table starts to get full, and the load factor approaches 1, the performance of linear probing becomes very bad.

To see what is happening consider the example shown in Fig 92. Suppose that we insert four keys, two that hash to `table[0]` and two that hash to `table[2]`. Because of the collisions, we will fill the table entries `table[1]` and `table[3]` as well. Now, suppose that the fifth key ("`t`") hashes to location `table[1]`. This is the first key to arrive at this entry, and so it is not involved any collisions. However, because of the previous collisions, it needs to slide down three positions to be entered into `table[4]`.

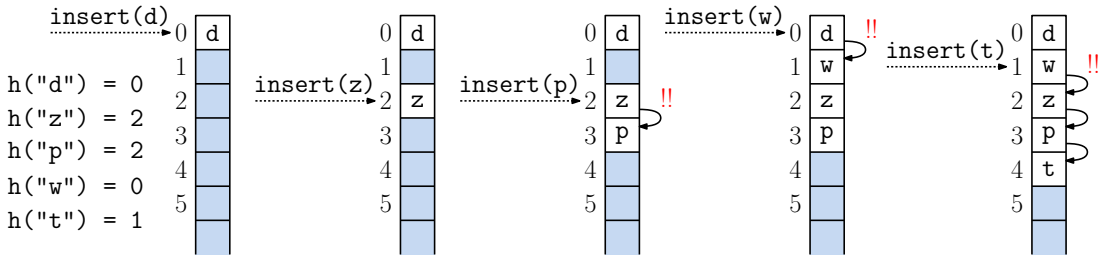



Fig. 92: Linear probing.

This phenomenon is called *primary clustering*. Primary clustering occurs when the collision resolution algorithm causes keys that hash to nearby locations to form into clumps. Linear probing is especially susceptible to primary clustering. As the load factor approaches 1, primary clustering becomes more and more pronounced, and probe sequences may become unacceptably long.

While we will not present it, a careful analysis shows that the expected costs for successful and unsuccessful searches using linear probing are, respectively:

$$S_{\text{LP}} = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \qquad U_{\text{LP}} = \frac{1}{2}\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right).$$

The proof is quite sophisticated, and we will skip it. Observe, however, that in the limit as $\lambda \to 1$ (a full table) the running times (especially for unsuccessful searches) rapidly grows to

infinity. A rule of thumb is that as long as the table remains less than 75% full, linear probing performs fairly well. Nonetheless, the issue of primary clustering is a major shortcoming, and the methods given below do significantly better in this regard.

The find procedure simply accesses sequential elements and wraps around when it reaches the end of the hash table (see the code block below).

Find Operation with Linear Probing

```
Value findLinear(Key x) {                  // find x using linear probing
    int c = h(x)                           // initial probe location
    int i = 0                              // probe offset
    while (table[c].key != empty) && (table[c].key != x) {
        c = (c+1) % m                      // increment and wrap around
    }
    return table[c].value                  // return associated value (or null if empty)
}
```

**Quadratic Probing:** To avoid primary clustering, one idea is to use a nonlinear probing function which scatters subsequent probes around more effectively. One such method is called *quadratic probing*, which works as follows. If the index hashed to $h(x)$ is full, then we consider next $h(x)+1, h(x)+4, h(x)+9, \ldots$ (again taking indices mod $m$). Thus, the probing function is $f(i) = i^2$. **Alert:** Note that the quadratic terms are added to the *initial* hash location, not to the previously probed location.

The `find` function is shown in the following code block. Rather than computing $h(x)+i^2$, we use a cute trick to update the probe location. Observe that $i^2 = (i-1)^2 + 2i - 1$. Thus, we can advance to the next position in the probe sequence ($i^2$) by incrementing the old position ($(i-1)^2$) by the value $2i-1$. We assume that each table entry `table[i]` contains two elements, `table[i].key` and `table[i].value`. If found, the function returns the associated value, and otherwise it returns `null`.

Find Operation with Quadratic Probing

```
Value findQuadratic(Key x) {               // find x using quadratic probing
    int c = h(x)                           // initial probe location
    int i = 0                              // probe offset
    while (table[c].key != empty) && (table[c].key != x) {
        c += 2*(++i) - 1                   // equivalent to c = h(x) + i*i
        c = c % m                          // wrap around if needed
    }
    return table[c].value                  // return associated value (or null if empty)
}
```

Experience shows that this succeeds in breaking up the clusters that arise from linear probing, but it is far from perfect. Quadractic probing is also clumping, called *secondary clustering*. This refers to clumping that occurs far away from the initial hash location. The problem with quadratic probing is that all keys use the same probe sequence. So, keys that hash to nearby locations have probe sequences that generate nearby locations as well.

In addition, quadratic probing suffers from a rather knotty problem. Unlike linear probing, which is guaranteed to try every entry in your table, quadratic probing bounces around less predictably. Might it miss some entries? The answer, unfortunately, is yes! To see why,

consider the rather trivial case where $m = 4$. Suppose that $h(x) = 0$ and your table has empty slots at `table[1]` and `table[3]`. The quadratic probe sequence will inspect the following indices:

$$1^2 \bmod 4 = 1 \qquad 2^2 \bmod 4 = 0 \qquad 3^2 \bmod 4 = 1 \qquad 4^2 \bmod 4 = 0 \ldots$$

It can be shown that it will only check table entries 0 and 1. This means that you cannot find a slot to insert this key, even though your table is only half full!

The following lemma shows that, if you choose your table size $m$ to be a prime number, then quadratic probing is guaranteed to visit at least half of the table entries before repeating. This means that it will succeed in finding an empty slot, provided that $m$ is prime and your load factor is smaller than 1/2.

**Theorem:** If quadratic probing is used, and the table size $m$ is a prime number, the first $\lfloor m/2 \rfloor$ probe sequences are distinct.

**Proof:** Suppose by way of contradiction that for $0 \leq i < j \leq \lfloor m/2 \rfloor$, both $h(x) + i^2$ and $h(x) + j^2$ are equivalent modulo $m$. Then the following equivalencies hold modulo $m$:

$$i^2 \equiv j^2 \iff i^2 - j^2 \equiv 0 \iff (i-j)(i+j) \equiv 0 \pmod{m}$$

This means that the quantity $(i-j)(i+j)$ is a multiple of $m$. But this cannot be, since $m$ is prime and both $i - j$ and $i + j$ are nonzero and strictly smaller than $m$. (The fact that $i < j \leq \lfloor m/2 \rfloor$ implies that their sum is strictly smaller than $m$.) Thus, we have the desired contradiction.

This is a rather weak result, however, since people usually want their hash tables to be more than half full. You can do better by being more careful in the choice of the table size and/or the quadratic increment. Here are two examples, which I will present without proof.

- If the table size $m$ is a prime number of the form $4k + 3$, then *alternating quadratic* probe sequence $(-1)^i i^2$ (that is, $h(x) + \langle 0, -1, +4, -9, +16, \ldots \rangle$) succeeds in probing all entries.
- If the table size $m$ is a power of two, and the increment is chosen to be $\frac{1}{2}(i^2 + i)$ (that is, $h(x) + \langle 0, +1, +3, +6, +10, \ldots \rangle$) succeeds in probing all entries.

The question of performance of quadratic probing is quite mathematically deep, and is closely related to the topic of *quadratic residues*[8] from number theory.

Examples of hash insertion with linear and quadratic probing are shown in Fig. 93 below.

**Double Hashing:** Both linear probing and quadratic probing have shortcomings. Our final method overcomes both of these limitations. Recall that in any open-addressing scheme, we are accessing the probe sequence $h(x) + f(1)$, $h(x) + f(2)$, and so on. How about if we make the increment function $f(i)$ a function of the search key? Indeed, to make it as random as possible, let's use another hash function! This leads to the concept of *double hashing*.

More formally, we define two hash functions $h(x)$ and $g(x)$. We use $h(x)$ to determine the first probe location. If this entry is occupied, we then try:

$$h(x) + g(x), \quad h(x) + 2g(x), \quad h(x) + 3g(x), \quad \ldots$$

[8] The *quadratic residues* of a number $m$ are the numbers of the form $(i^2 \bmod m)$, where $i$ is a nonnegative integer.

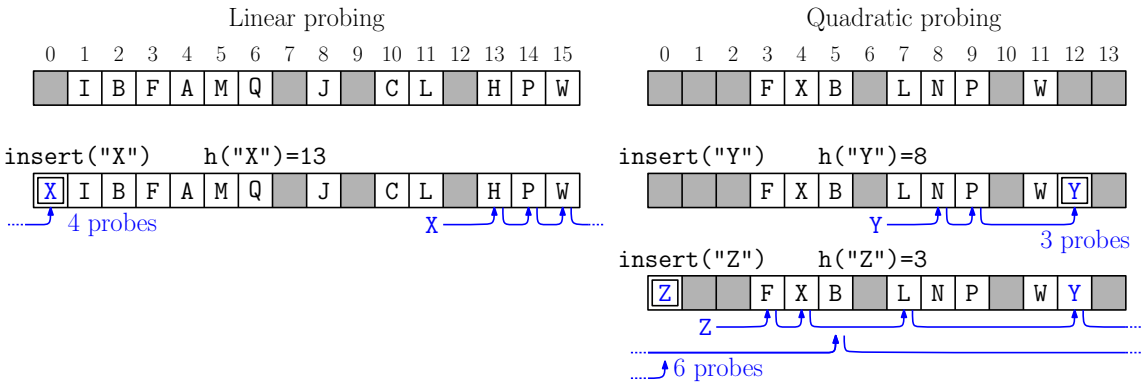



Fig. 93: Examples of open-addressing hashing using linear probing and quadratic probing.

More formally, the probe sequence is defined by the function $f(i) = i \cdot g(x)$. In order to be sure that we do not cycle, it should be the case that $m$ and $g(x)$ are *relatively prime*, that is, they share no common factors. There are lots of ways to achieve this. For example, select $g(x)$ to be a prime that is strictly larger than $m$ or the product of primes that are larger than $m$. Another approach would be to set $m$ to be a power of 2, and then to generate $g(x)$ as the product of prime numbers other than 2. In short, we should be careful in the design of a double-hashing scheme, but there is a lot of room for adjustment.

A couple examples of hash insertion with double hashing are shown in Fig. 94 below. Observe that in the second case, we go into an infinite loop and fail to insert the key, even though the table is not full. This demonstrates the danger that occurs if $g = 3$ and $m = 15$ are not relatively prime (in this case, they share the common factor 3)s.

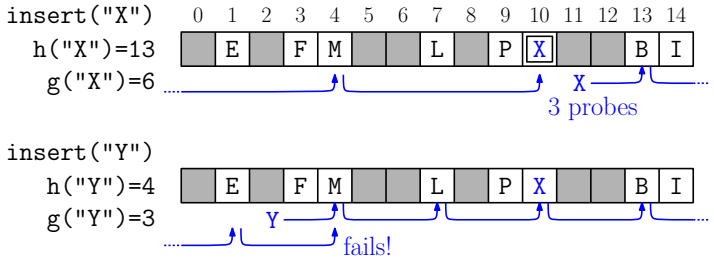



Fig. 94: Examples of open-addressing hashing using double hashing.

**At a Glance:** Fig. 95 provides an illustration of how the various open-addressing probing methods work.

Theoretical running-time analysis shows that double hashing is the most efficient among the open-addressing methods of hashing, and it is competitive with separate chaining. The running times of successful and unsuccessful searches for open addressing using double hashing are

$$S_{\text{DH}} = \frac{1}{\lambda} \ln \frac{1}{1-\lambda} \qquad U_{\text{DH}} = \frac{1}{1-\lambda}.$$

To get some feeling for what these quantities mean, consider the following table:

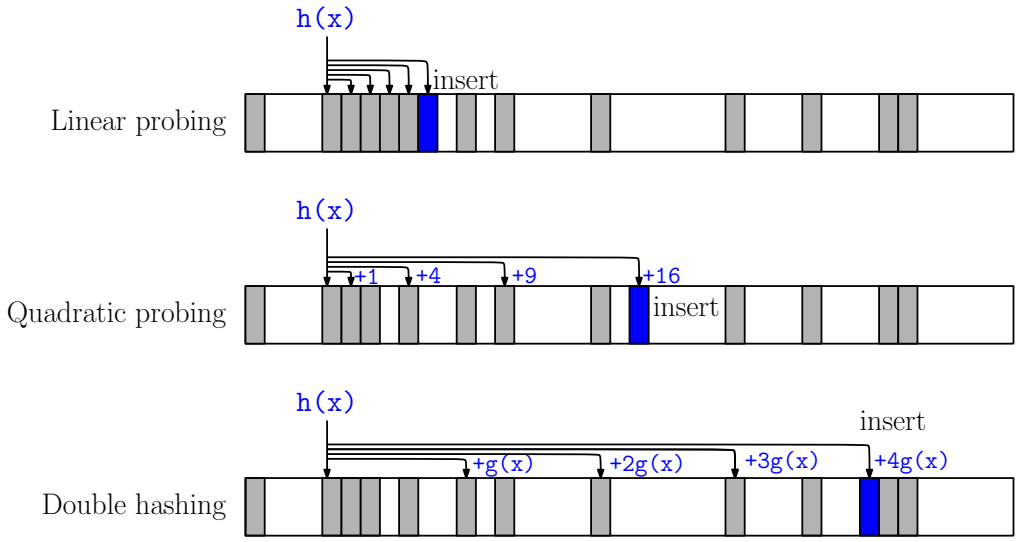



Fig. 95: Various open-addressing systems. (Shaded squares are occupied and the black square indicates where the key is inserted.)

| $\lambda$ | 0.50 | 0.75 | 0.90 | 0.95 | 0.99 |
|---|---|---|---|---|---|
| $U(\lambda)$ | 2.00 | 4.00 | 10.0 | 20.0 | 100. |
| $S(\lambda)$ | 1.39 | 1.89 | 2.56 | 3.15 | 4.65 |

Note that, unlike tree-based search structures where the search time grows with $n$, these search times depend only on the load factor. For example, if you were storing 100,000 items in your data structure, the above search times (except for the very highest load factors) are superior to a running time of $O(\log n)$.

**Deletions:** Deletions are a bit tricky with open-addressing schemes. Can you see why?

The issue is illustrated Fig. 96. When we insert "`a`", an existing key "`f`" was on the probe path, and we inserted "`a`" beyond "`f`". Then we delete "`f`" and then search for "`a`". The problem is that with "`f`" no longer on the probe path, we arrive at the empty slot and take this to mean that "`a`" is not in the dictionary, which is not correct.

To handle this we create a new special value (in addition to `empty`) for cells whose keys have been deleted, called, say "`deleted`". If the entry is marked `deleted` this means that the slot is available for future insertions, but if the `find` function comes across such an entry, it should keep searching. The searching stops when it either finds the key or arrives at an cell marked "`empty`" (key not found).

Using the "`deleted`" entry is a rather quick-and-dirty fix. It suffers from the shortcoming that as keys are deleted, the search paths are unnaturally long. (The load factor has come down, but the search paths are just as long as before.) A more clever solution would involve moving keys that that were pushed down in the probe sequence up to fill the vacated entries. Doing this, however make deletion times longer.

**Further refinements:** Hashing is a very well studied topic. We have hit the major points, but there are a number of interesting refinements that can be applied. One example is a technique called *Brent's method*. This approach is used to reduce the search times when double hashing is used. It exploits the fact that any given cell of the table may lie at the intersection of

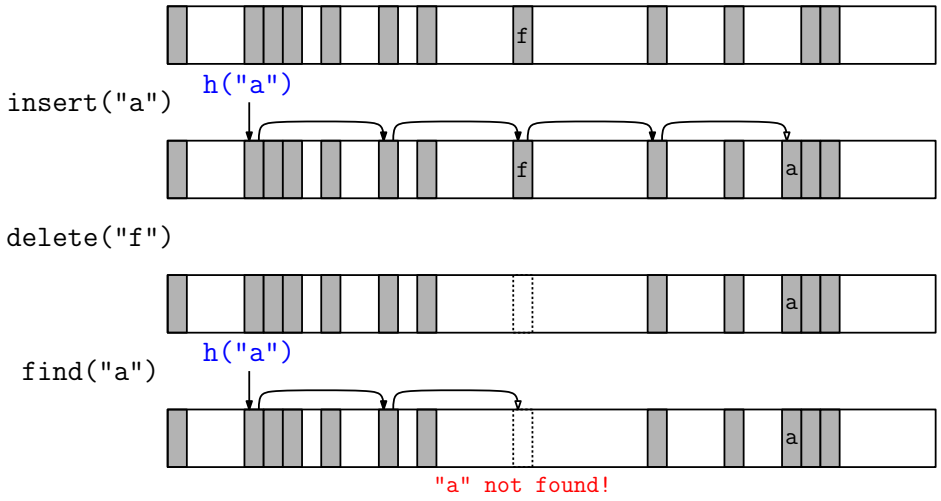



Fig. 96: The problem with deletion in open addressing systems.

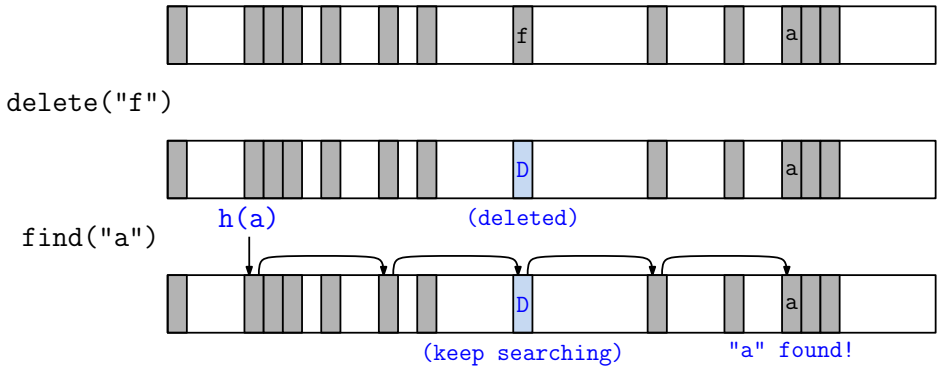



Fig. 97: Deleting in open-addressing by using special *empty* entry.

two or more probe sequences. If one of these probe sequences is significantly longer than the other, we can reduce the average search time by changing which key is placed at this point of overlap. Brent's algorithm optimizes this selection of which keys occupy these locations in the hash table.

**Controlling the Load Factor and Rehashing:** Recall that the load factor of a hashing scheme is $\lambda = n/m$. The collision-resolution methods we have seen have running times that grow as $O(\lambda/(1-\lambda))$. Clearly, we would like $\lambda$ to be small and in fact strictly smaller than 1. Making $\lambda$ too small is wasteful, however, since it means that our table size is significantly larger than the number of keys. This suggests that we define two constants $0 < \lambda_{\min} < \lambda_{\max} < 1$, and maintain the invariant that $\lambda_{\min} \leq \lambda \leq \lambda_{\max}$. This is equivalent to saying that $n \leq \lambda_{\max} m$ (that is, the table is never too close to being full) and $m \leq n/\lambda_{\min}$ (that is, the table size is not significantly larger than the number of entries). Define the *ideal load factor* to be the mean of these two, $\lambda_0 = (\lambda_{\min} + \lambda_{\max})/2$.

Now, as we insert new entries, if the load factor ever exceeds $\lambda_{\max}$ (that is, $n > \lambda_{\max} m$), we replace the hash table with a larger one, devise a new hash function (suited to the larger size), and then insert the elements from the old table into the new one, using the new hash function. This is called *rehashing* (see Fig. 98). More formally:

- Allocate a new hash table of size $m' = \lceil n/\lambda_0 \rceil$
- Generate a new hash function $h'$ based on the new table size
- For each entry $(x, v)$ in the old hash table, insert it into the new table using $h'$
- Remove the old table

Observe that after rehashing the new load factor is roughly $n/m' \approx \lambda_0$, thus we have restored the table to the ideal load factor. (The ceiling is a bit of an algebraic inconvenience. Throughout, we will assume that $n$ is sufficiently large that floors and ceilings are not significant.)

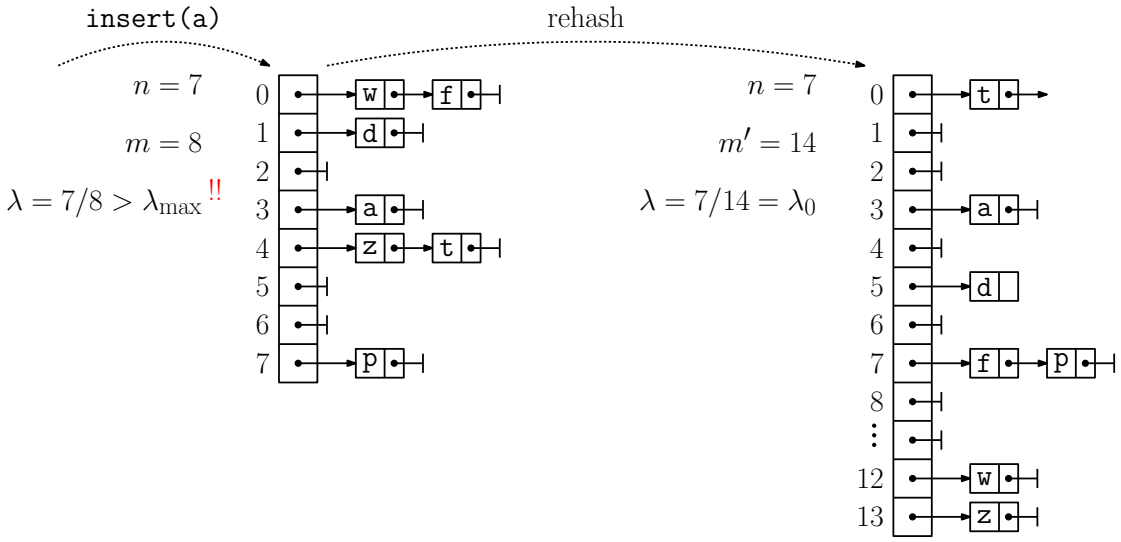



Fig. 98: Controlling the load factor by rehashing, where $\lambda_{\min} = 0.25$, $\lambda_{\max} = 0.75$, and $\lambda_0 = 0.5$. When the table size changes, we create an entirely different hash function.

Symmetrically, as we delete entries, if the load factor ever falls below $\lambda_{\min}$ (that is, $n < \lambda_{\min} m$), we replace the hash table with a smaller one of size $\lceil n/\lambda_0 \rceil$, generate a new hash

function for this table, and we rehash entries into this new table. Note that in both cases (expanding and contracting) the hash table changes by a constant fraction of its current size. This is significant in the analysis.

**Rehashing:** The advantage of open addressing is that we do not have to worry about pointers and storage allocation. However, if the table becomes full, or just too close to full so that performance starts degrading (e.g. the load factor exceeds some threshold in the range from 80% to 90%). The simplest scheme, is to allocate a new array of size a constant factor larger (e.g. twice) the previous array. Then create a new hash function for this array. Finally go through the old array, and hash all the old keys into the new table, and then delete the old array.

You may think that in the worst case this could lead to *lots* of recopying of elements, but notice that if the last time you rehashed you had 1000 items in the array, the next time you may have 2000 elements, which means that you performed at least 1000 insertions in the mean time. Thus the time to copy the 2000 elements is *amortized* by the previous 1000 insertions leading up to this situation.

**Amortized Analysis of Rehashing:** (Optional.) Here we show that if a hash table is maintained using rehashing when it gets too full (as described earlier), the amortized cost per operation is a constant. The proof is quite similar to the proof given earlier in the semester for the dynamically expanding stack.

Observe that whenever we rehash, the running time is proportional to the number of keys $n$. If $n$ is large, rehashing clearly takes a lot of time. But once we have rehashed, we will need to do a significant number of insertions or deletions before we need to rehash again.

To make this concrete, let's consider a specific example. Suppose that $\lambda_{\min} = 1/4$ and $\lambda_{\max} = 3/4$, and hence $\lambda_0 = 1/2$. Also suppose that the current table size is $m = 1000$. Suppose the most recent insertion caused the load factor to exceed our upper bound, that is $n > \lambda_{\max} m = 750$. We allocate a new table of size $m' = n/\lambda_0 = 2n = 1500$, and rehash all the old elements into this new table. In order to overflow this new table, we will need for $n$ to increase to some higher value $n'$ such that $n'/m' > \lambda_{\max}$, that is $n' > (3/4)1500 = 1125$. In order to grow from the current 750 keys to 1125 keys, we needed to have at least 375 more insertions (and perhaps many more operations if finds and deletions were included as well). This means that we can *amortize* the (expensive) cost of rehashing 1125 keys against the 375 (cheap) insertions. (If you do the calculations, with 4 tokens per insertion per the example, you get $(1 + 3)375 = 375 + 1125$ so you've accumulated the desired 1125 tokens to pay for the cost of rehashing.)

Recall that the *amortized cost* of a series of operations is the total cost divided by the number of operations.

**Theorem:** Assuming that individual hashing operations take $O(1)$ time each, if we start with an empty hash table, the amortized complexity of hashing using the above rehashing method with minimum and maximum load factors of $\lambda_{\min}$ and $\lambda_{\max}$, respectively, is at most $1 + 2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min})$.

**Proof:** Our proof is based on the same *token-based argument* that we used in the earlier lecture. Let us assume that each standard hashing operation takes exactly 1 unit of time, and rehashing takes time $n$, where $n$ is the number of entries currently in the table. Whenever we perform a hashing operation, we assess 1 unit to the actual operation, and

save $2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min})$ *tokens* to pay for future rehashings. (Where did this "magic" number of tokens come from? The answer is to work through the analysis below treating the number of tokens as an unknown quantity $x$. Then figure out what value $x$ needs to be to make the inequalities work out.)

There are two ways to trigger rehashing: expansion due to insertion, and contraction due to deletion. Let us consider insertion first. Suppose that our most recent insertion has triggered rehashing. This implies that the current table contains roughly $n \approx \lambda_{\max} m$ entries. (Again, to avoid worrying about floors and ceilings, let's assume that $n$ is quite large.) The last time the table was rehashed, the table contained $n' = \lambda_0 m$ entries immediately after the rehashing finished. This implies that we inserted at least $n - n' = (\lambda_{\max} - \lambda_0)m$ entries. Therefore, the number of tokens we have accumulated since then is at least

$$\begin{aligned}
(\lambda_{\max} - \lambda_0)m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} &= \left(\lambda_{\max} - \frac{\lambda_{\max} + \lambda_{\min}}{2}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \left(\frac{\lambda_{\max} - \lambda_{\min}}{2}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \lambda_{\max} m \;\approx\; n,
\end{aligned}$$

which implies that we have accumulated enough tokens to pay the cost of $n$ to rehash.

Next, suppose that our most recent deletion has triggered rehashing. This implies that the current table contains roughly $n \approx \lambda_{\min} m$ entries. (Again, to avoid worrying about floors and ceilings, let's assume that $n$ is quite large.) The last time the table was rehashed, the table contained $n' = \lambda_0 m$ entries immediately after the rehashing finished. This implies that we deleted at least $n' - n = (\lambda_0 - \lambda_{\min})m$ entries. Therefore, the number of tokens we have accumulated since then is at least

$$\begin{aligned}
(\lambda_0 - \lambda_{\min})m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} &= \left(\frac{\lambda_{\max} + \lambda_{\min}}{2} - \lambda_{\min}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \left(\frac{\lambda_{\max} - \lambda_{\min}}{2}\right) m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\
&= \lambda_{\max} m \;\geq\; \lambda_{\min} m \;\approx\; n,
\end{aligned}$$

again implying that we have accumulated enough tokens to pay the cost of $n$ to rehash.

To make this a bit more concrete, suppose that we set $\lambda_{\min} = 1/4$ and $\lambda_{\max} = 3/4$, so that $\lambda_0 = 1/2$ (see Fig. 98). Then the amortized cost of each hashing operation is at most $1 + 2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min}) = 1 + 2(3/4)/(1/2) = 4$. Thus, we pay just additional factor of four due to rehashing. Of course, this is a worst case bound. When the number of insertions and deletions is relatively well balanced, we do not need rehash very often, and the amortized cost is even smaller.

## Lecture 16: B-Trees

**Data Structures for External Memory:** While binary trees are great data structures for ordered dictionaries stored in main memory, these data structures are really not appropriate for data stored on external memory systems (i.e., disks). When accessing data on a disk, latency (that is, the delay waiting for the transfer to begin) is significant, but an entire *block*

(or "*page*") of memory input at once. So it makes sense to design the tree so that each node of the tree essentially occupies one entire page.

This idea applies more generally to modern memory systems, which are organized hierarchically. Memory is partitioned into various levels of caches, with each successive level having higher latency and larger page size. The data structure we will discuss today is appropriate whenever memory can be accessed efficiently in blocks.

This suggests the generalization of *multiway search trees*, where each node is allocated so that it coincides with a single page. Each node of a standard binary search tree store a single key value and two children left and right storing keys that are smaller than and greater than this key, respectively. In a *j-ary* multiway search tree node, a node stores references to $j$ different subtrees, $T_1, \ldots, T_j$ and contains $j-1$ key values, $a_1 < \ldots < a_{j-1}$, such that subtree $T_i$ stores nodes whose key values $x$ such that $a_{i-1} < x < a_i$. (To handle the boundary cases, let's make the convention that $a_0 = -\infty$ and $a_j = +\infty$, but these are not stored in the node.)

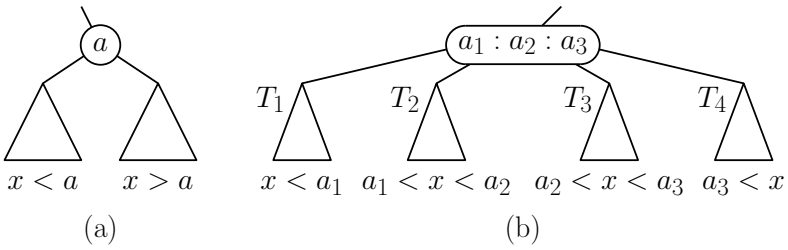



Fig. 99: Binary and 4-ary search tree nodes.

**B-trees:** B-trees are multiway search trees, in which we achieve balance by constraining the "width" of each node. B-trees were first introduced way back in 1970 by Rudolf Bayer and Edward McCreight. They have proven to be very popular. The 2-3 tree that we studied in an earlier lecture is a special case (indeed, the smallest special case) of a B-tree. Numerous modifications and adaptations of B-trees have been developed over the years. We will present one, fairly simple, formulation. (Later in the lecture we will discuss a particularly popular variant, called B+ trees.)

For any integer $m \geq 3$, a *B-tree of order m* is a multiway search tree has the following properties (see Fig. 100):

- The root is either a leaf or has between two and $m$ children.
- Each node except the root has between $\lceil m/2 \rceil$ and $m$ children (which may be empty, that is `null`). A node with $j$ children contains $j-1$ key.
- All leaves are at the same level of the tree.

The 2-3 tree that we presented earlier is an example of a B-tree of order 3. The typical fan-out values for B-trees are quite large. For example, B-trees of order of around 100 are common in practice. A node in such a tree has between 50 and 100 children and holds between 49 and 99 keys. Of course, with such high fan-outs, the heights of the tree are quite small.

**Height Analysis:** The following theorem show that as fan-out of a B-tree grows, the height of the tree decreases.

**Theorem:** A B-tree of order $m$ containing $n$ keys has height at most $\frac{\lg n}{\gamma}$, where $\gamma = \lg \frac{m}{2}$.

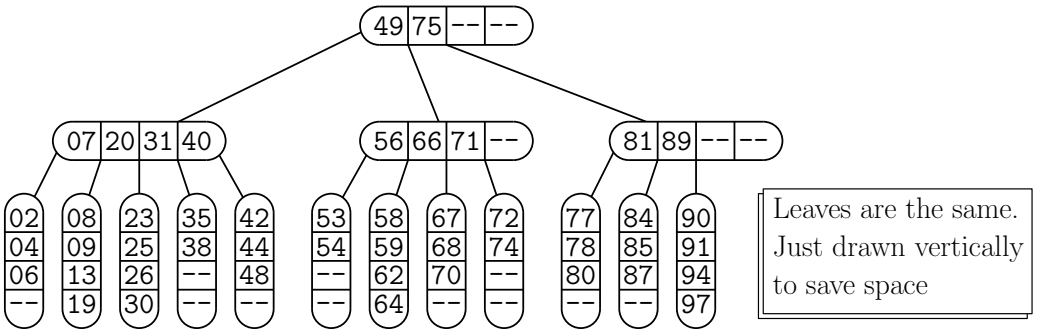



Fig. 100: B-tree of order 5. (Each node, except possibly the root has 3–5 children.)

**Proof:** The formal proof is a bit messy, so we will instead prove a simpler result. Let's let $n$ denote the number of keys just at the leaf level. (This is reasonable, because in any multiway tree, a constant fraction of the nodes are leaves, and as the degree increases, the fraction increases.) Let $h \geq 1$ denote the height of our tree, and let $N(h)$ denote the smallest possible number of keys in the leaves of this tree. Since $N(h) \leq n$, it suffices to show that $h \leq \frac{\lg N(h)}{\gamma}$.

Recall that the degree of a node is its number of children. The root node has degree at least two, and all other nodes have degree at least $d = m/2$. Thus, there are at least two nodes at depth 1, and with each additional level the number increases by a factor of $d$. Thus, there are at least $2d$ nodes at depth 2, $2d^2$ at depth 3, and generally $2d^{h-1}$ at depth $h$. Each of these leaf nodes contains at least $d$ keys, implying the total number of keys at the leaf level is at least $2d^h$. Thus, we have

$$\begin{aligned} & N(h) \;\geq\; 2d^h \;\geq\; d^h \\ \Longrightarrow\quad & \lg N(h) \;\geq\; \lg(d^h) \;=\; h \lg d \qquad \text{(taking lg on both sides)} \\ \Longrightarrow\quad & \frac{\lg N(h)}{\lg d} \;\geq\; h. \end{aligned}$$

Recalling that $\gamma = \lg \frac{m}{2} = \lg d$, a tree with $n$ keys has height at least $h \leq \frac{\lg N(h)}{\gamma}$, as desired.

For example, when $m = 100$, this implies that the height of the B-tree is not greater than $(\lg n)/5.6$, that is, it is 5.6 times smaller than a binary search tree. For example, this means that you can store over a 100 million keys in a search structure of height roughly five!

**Node structure:** Although B-tree nodes can hold a variable number of items, this number generally changes dynamically as keys are inserted and deleted. Therefore, every node is allocated with the maximum possible size, but most nodes will not be fully utilized. (Experimental studies show that B-tree nodes are on average about 2/3 utilized.) The code block below shows a possible Java implementation of a B-tree node implementation.

Note that 2-3 trees and 2-3-4 trees discussed in earlier lectures are special cases (when $M = 3$ and $M = 4$, respectively.)

**Search:** Searching a B-tree for a key $x$ is a straightforward generalization of binary tree searching. When you arrive at an internal node with keys $a_1 < a_2 < \ldots < a_{j-1}$ search (either linearly or by binary search) for $x$ in this list. If you find $x$ in the list, then we have found $x$. Otherwise,

B-Tree Node

```
final int M = ...                   // order of the B-tree

class BTreeNode {
    int         nChildren;          // current number of children (from M/2 to M)
    BTreeNode   child[M];           // children pointers
    Key         key[M-1];           // keys
    Value       value[M-1];         // values
}
```

determine the index $i$ such that $a_{i-1} < x < a_i$. (Recall that $a_0 = -\infty$ and $a_j = +\infty$.) Then recursively search the subtree $T_i$. When you arrive at a leaf, search all the keys in this node. If it is not here, then $x$ is not in the B-tree.

Since nodes may be quite wide, should we employ a fast search method such as binary search? Well, you could, but normally latency times (the time needed to load a node from external memory) are typically so much higher than processing times that simple linear search is fast enough.

**Restructuring:** In an earlier lecture, we showed how to restructure 2-3 trees. We had three mechanisms: splitting nodes, merging nodes, and subtree adoption. We will generalize each of these operations to general B-trees.

**Key Rotation (Adoption):** Recall that a node in a B-tree can have from $\lceil m/2 \rceil$ up to $m$ children, and the number of keys is smaller by one. As a result of insertion or deletion, a node may acquire one too many ($m+1$ children and hence, $m$ keys) or one too few ($\lceil m/2 \rceil - 1$ children and hence, $\lceil m/2 \rceil - 2$ keys).

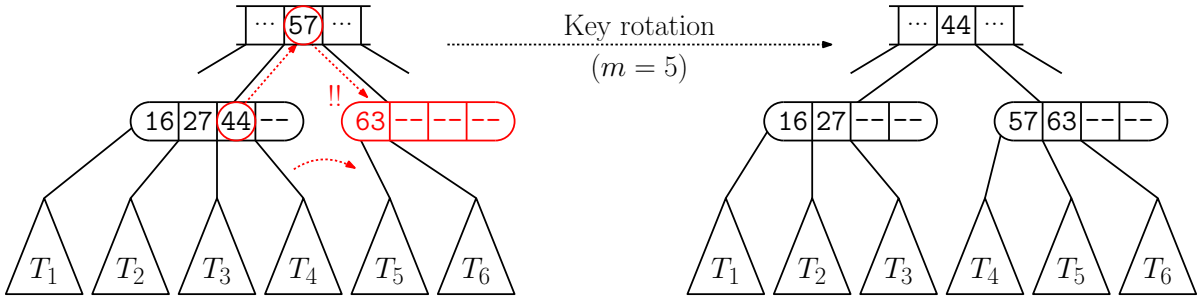



Fig. 101: Key rotation for a B-tree of order $m = 5$.

The easiest way in which to remedy the imbalanced is to move a child into or from one of your siblings, assuming that you have a sibling can absorb this change. This is called *key rotation* (or as I call it, *adoption*). For example, in Fig. 101, the node in red has too few children, and since its left sibling can spare a child, we move this node's rightmost child over, sliding the associated key value up to the parent and we take the parent's key value.

This operation is not always possible, because it depends on the existence of a sibling with a proper number of keys. Because allocating and deallocating nodes is a relatively expensive operation, *this is the preferred rebalancing operation.*

**Node Splitting:** As the result of insertion, a node may acquire one too many children ($m+1$ children and hence, $m$ keys). When this happens and key rotation is not available, we split the node into two nodes, one having $m' = \lceil m/2 \rceil$ children and the other having the remaining $m'' = m + 1 - \lceil m/2 \rceil$ children. (For example, if $m = 8$, when a node has $m + 1 = 9$ children it is split into one of size $m' = 4$ and the other $m'' = 5$.) Clearly, the first node has an acceptable number of children. The following lemma demonstrates that the other node has an acceptable number of children as well.

**Lemma:** For all $m \geq 2$, $\lceil m/2 \rceil \leq m + 1 - \lceil m/2 \rceil \leq m$.

**Proof:** If $m$ is even, then $\lceil m/2 \rceil = m/2$, and the middle expression in the inequality reduces to $m + 1 - m/2 = m/2 + 1$. Thus, the claim is equivalent to

$$\frac{m}{2} \;\leq\; \frac{m}{2} + 1 \;\leq\; m,$$

which is clearly true for any even $m \geq 2$. On the other hand, if $m$ is odd then $\lceil m/2 \rceil = (m+1)/2$, and the middle expression in the inequality reduces to $m + 1 - (m + 1)/2 = (m + 1)/2$. Thus, the claim is equivalent to

$$\frac{m+1}{2} \;\leq\; \frac{m+1}{2} \leq m,$$

which is also clearly true for any $m \geq 1$.

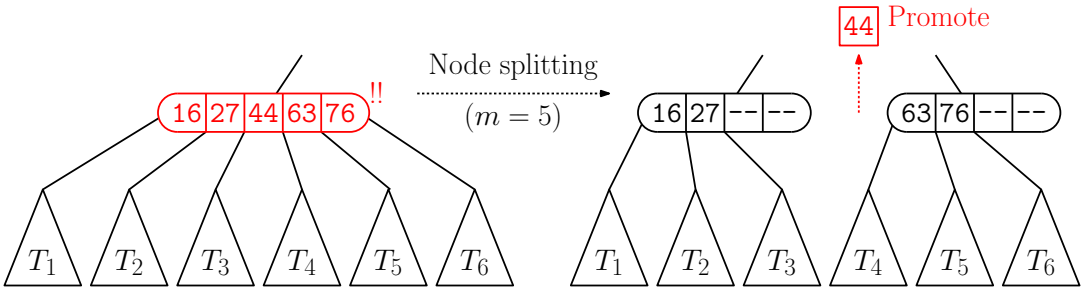



Fig. 102: Node splitting for a B-tree of order $m = 5$.

Returning to node splitting, we create two nodes and distribute the smallest $m'$ subtrees to the first and the remaining $m''$ to the second node (see Fig. 102). Among the $m - 1$ keys, $m' - 1$ smallest keys go with the first node and the $m'' - 1$ largest keys go with the other node. Since $(m' - 1) + (m'' - 1) = m - 2$, we have one extra key that does not fit into either of these nodes. If we assume that the keys are indexed starting with 0, the key to be promoted has index $m' - 1$. (When $m$ is odd, both nodes get the same number of children and the same number of keys. When $m$ is even, the left node gets one fewer child and one fewer key compared to the right node.) This key is promoted to the parent node. (As with 2-3 trees, if we do not have a parent, we create a new root node with this single key and just two children. By the way, this is the reason that we allowed the root to have fewer than $\lceil m/2 \rceil$ children.)

Since the parent acquires an extra key and extra child, the splitting process may propagate to the parent node.

**Node Merging:** As the result of deletion, a node may have one too few children ($\lceil m/2 \rceil - 1$ children and hence, $\lceil m/2 \rceil - 2$ keys). When this happens and key rotation is not available,

we may infer that its siblings have the minimum number $\lceil m/2 \rceil$ children. We merge this node with either of its siblings into a single node having a total of $m' = (\lceil m/2 \rceil - 1) + \lceil m/2 \rceil = 2\lceil m/2 \rceil - 1$ children. The following lemma demonstrates that the resulting node has an acceptable number of children.

**Lemma:** For all $m \geq 2$, $\lceil m/2 \rceil \leq 2\lceil m/2 \rceil - 1 \leq m$.

**Proof:** If $m$ is even, then $\lceil m/2 \rceil = m/2$, and the middle expression in the inequality reduces to $2(m/2) - 1 = m - 1$. Thus, the claim is equivalent to

$$\frac{m}{2} \;\leq\; m - 1 \;\leq\; m,$$

which is easily true for any $m \geq 2$. On the other hand, if $m$ is odd then $\lceil m/2 \rceil = (m+1)/2$, and the middle expression in the inequality reduces to $m$. Thus, the claim is equivalent to

$$\frac{m+1}{2} \;\leq\; m \;\leq\; m,$$

which is easily true for any $m \geq 1$.

Returning to node merging, we merge the two nodes into a single node having $m'$ children (see Fig. 103). The number of keys from the two initial nodes is $\lceil m/2 \rceil - 2 + \lceil m/2 \rceil = 2\lceil m/2 \rceil - 2 = m' - 2$, which is one too few. We demote the appropriate key from the parent's node to yield the desired number of keys.

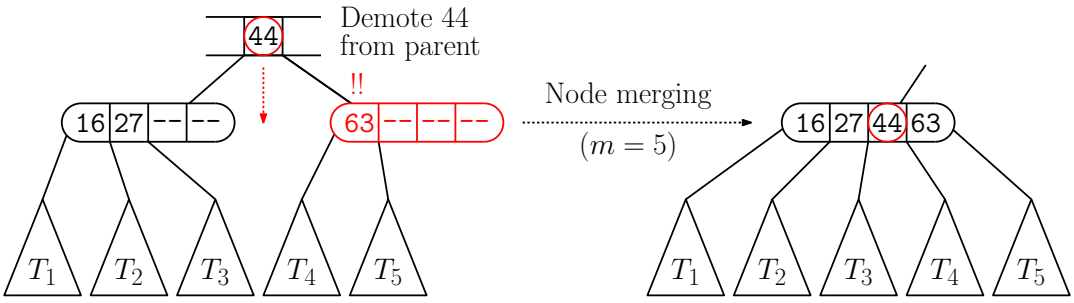



Fig. 103: Node merging for a B-tree of order $m = 5$.

Since the parent has lost a key and a child, the merging process may propagate to the parent node.

Given these operations, we can now describe how to perform the various dictionary operations.

**Insertion:** In the case of 2-3 trees, we would always split a node when it had too many keys. With B-trees, creating nodes is a more expensive operation. So, whenever possible we will try to employ key rotation to resolve nodes that are too full, and we will fall back on node splitting only when necessary.

To insert a key into a B-tree of order $m$, we perform a search to find the appropriate leaf into which to insert the node. If we find the key, then we signal a duplicate-key error. Otherwise, if the leaf is not at full capacity (it has fewer than $m - 1$ keys) then we simply insert it and are done. Note that this will involve sliding keys around within the leaf node to make room for the new entry, but since $m$ is assumed to be a constant (e.g., the size of one disk page), we ignore this extra cost.

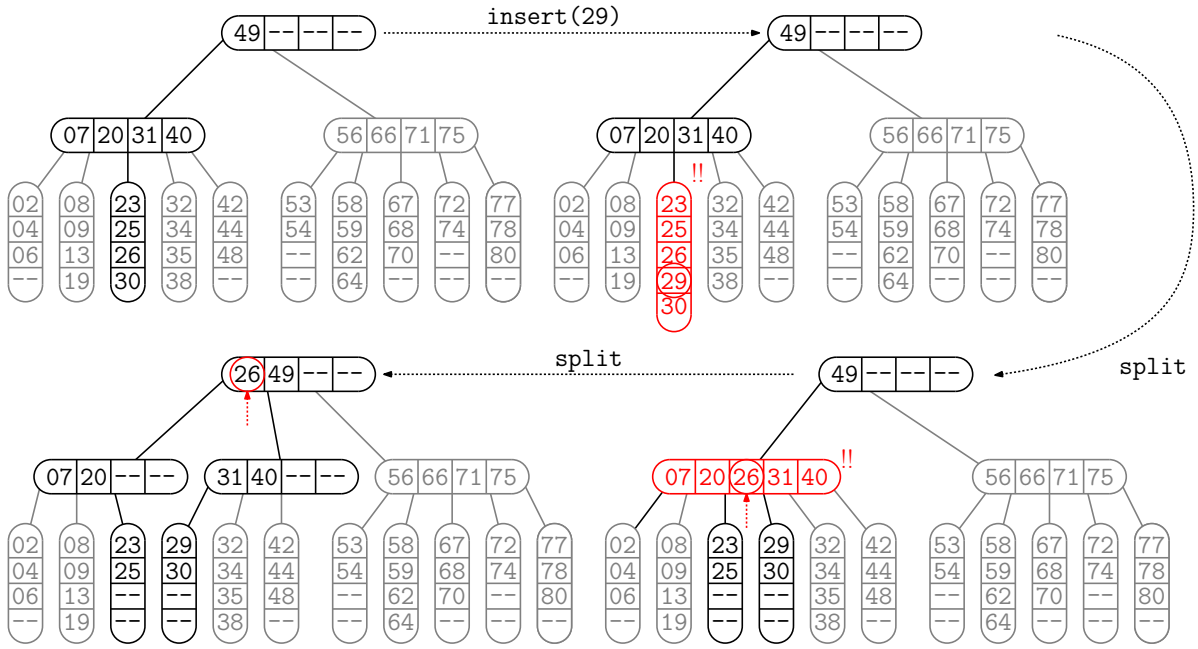



Fig. 104: Insertion of key 29 (order $m = 5$). It is inserted into the leaf node $(23, 25, \ldots)$ which splits, causing the middle key 26 to be promoted to the parent, which also splits, causing the middle key 26 to be promoted to the root.

Otherwise the node *overflows* and to remedy the situation, we first check whether either sibling is less than full. If so, we perform a rotation moving the extra key and child into this sibling. Otherwise, we perform a node split as described above (see Fig. 104). When this happens, the parent acquires a new child and new key, and thus the splitting process may continue with the parent node.

**Deletion:** As in binary tree deletion we begin by finding the node to be deleted. We need to find a suitable replacement for this node. This is done by finding the largest key in the left child (or equivalently the smallest key in the right child), and moving this key up to fill the hole. This key will always be at the leaf level. This creates a hole at the leaf node. If this leaf node still has sufficient capacity (at least $\lceil m/2 \rceil - 1$ keys) then we are done.

Otherwise, we have an underflow situation at this node. As with insertion we first check whether a *key rotation* is possible. If one of the two siblings has at least one key more than the minimum, then we rotate the extra key into this node, and we are done (see Fig. 105).

If this is not possible, then any siblings of ours must have the minimum number of $\lceil m/2 \rceil$ children, and so we can apply a node node merge (see Fig. 106).

The removal of a key from the parent's node may cause it to underflow. Thus, the process may need to be repeated recursively up to the root. If the root now has only one child, and we make this single child the new root of the B-tree.

**B+ trees:** B-trees have been very successful, and a number of variants have been proposed. A particularly popular one for disk storage is called a *B+ tree*. The key differences with the standard B-tree as the following:

- Internal and leaf nodes are different in structure:

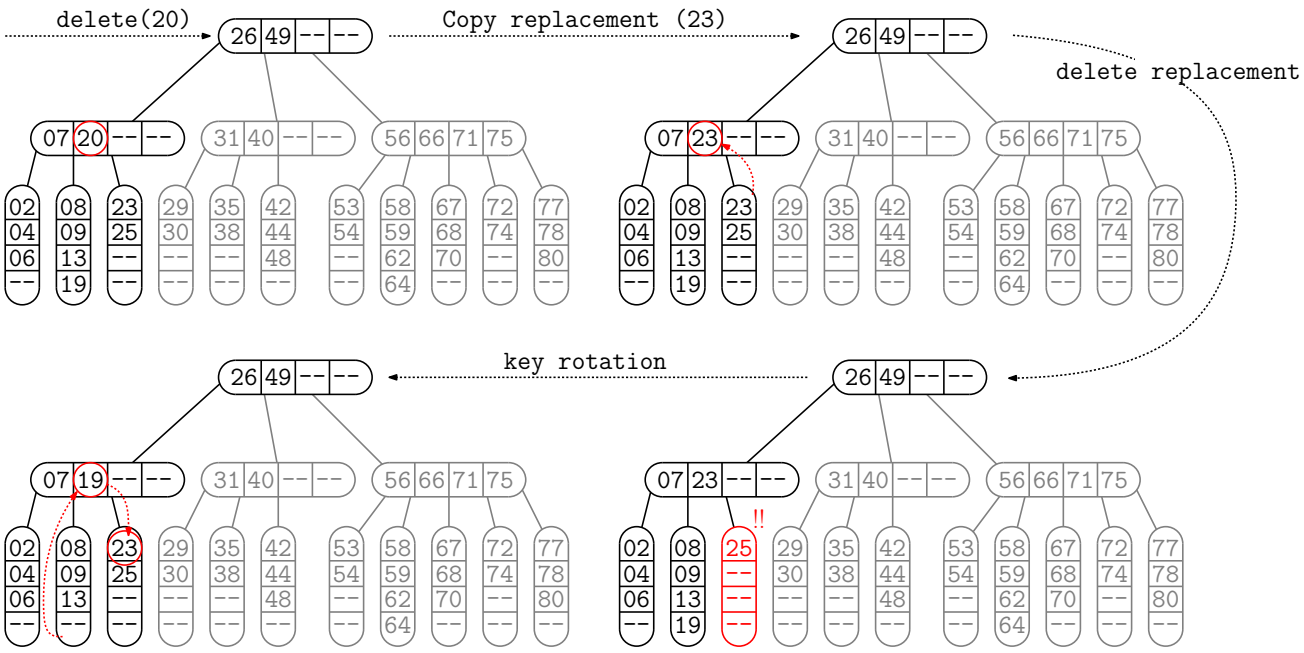



Fig. 105: Deletion of key 20 (order $m = 5$). Since 20 is not a leaf, we find the replacement 23 (inorder successor) and copy it there. We then delete 23 from its leaf, which underflows. The sibling rotates the key 19 to the parent and the parent gives up its key 23.

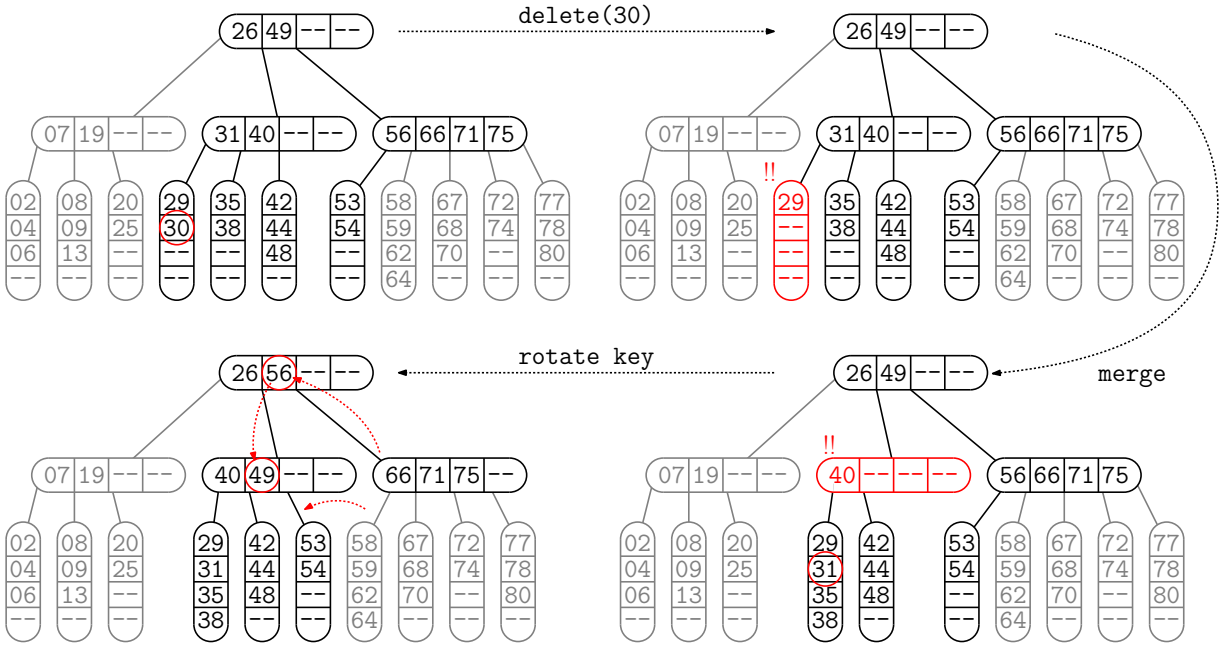



Fig. 106: Deletion of key 30 (order $m = 5$). Its leaf is underfull, but its sibling cannot give up a key, so we merge with $(35, 38)$, demoting the middle key 31 from the parent. Now the parent 40 is underfull, but its right sibling can spare a key, so 56 is rotated to the parent and 49 is rotated down.

- Internal nodes store keys only, no values. The keys in the internal nodes are used solely for locating the leaf node containing the actual data, so it is not necessary that every key appearing in an internal node need correspond to an actual key-value pair.
- All the key-value pairs are stored in the leaf nodes. There is no need for child pointers. (This also saves space.)

• Each leaf node has a *next-leaf* pointer, which points to the next leaf in sorted order.

Storing keys only in the internal nodes saves space, and allows for increased fan-out. This means the tree height is lower, which reduces number of disk accesses. Thus, the internal nodes are merely an *index* to locating the actual data, which resides at the leaf level. (The policy regarding which keys a subtree contains are changed. Given an internal node with keys $\langle a_1, \ldots, a_{j-1} \rangle$, subtree $T_j$ contains keys $x$ such that $a_{i-1} < x \leq a_i$.)

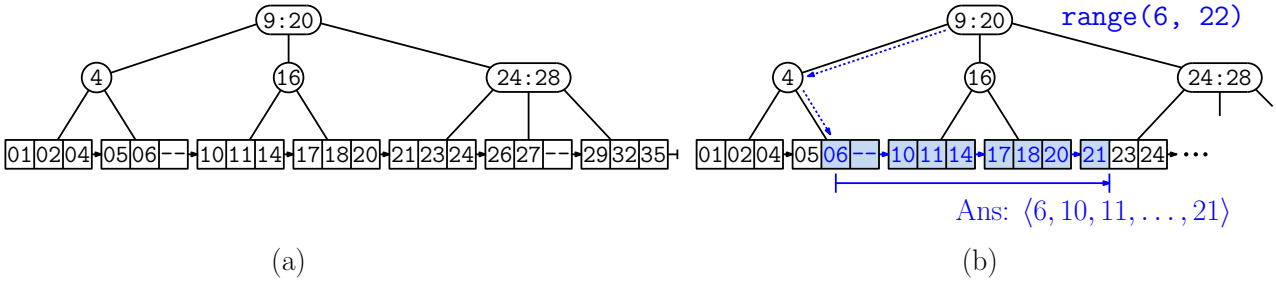



Fig. 107: B+ tree of order $m = 3$, where leaves can hold up to 3 keys.

The next-leaf links enable efficient *range reporting* queries. In such a query, we are asked to list all the keys in a range $[x_{\min}, x_{\max}]$. We simply find the leaf node for $x_{\min}$ and then follow next-leaf links until exceeding $x_{\max}$.

## Lecture 17: Treaps

**Randomized Data Structures:** A common design technique in the field of algorithm design involves the notion of using randomization. A *randomized algorithm* employs a pseudo-random number generator to inform some of its decisions. Randomization has proved to be a remarkably useful technique, and randomized algorithms are often the fastest and simplest algorithms for a given application. In the case of data structures, randomization affects the balance of the structure, and hence the running time. The results are guaranteed to be correct.

In an earlier lecture, we have already seen an example of a randomized data structure in the form of the skip list. In this lecture, we will consider another one, called the *treap*.

**Background:** This treap data structure's name is a portmanteau (combination) of "tree" and "heap." It was developed by Raimund Seidel and Cecilia Aragon in 1989. (This 1-dimensional data structure is closely related to two 2-dimensional data structures, the *Cartesian tree* by Jean Vuillemin and the *priority search tree* of Edward McCreight, both discovered in 1980.)

Because the treap is a randomized data structure, its running time depends on the random choices made by the algorithm. We will see that all the standard dictionary operations take

$O(\log n)$ *expected time*. The expectation is taken over all possible random choices that the algorithm may make. You might be concerned, since this allows for rare instances where the running time can be is very bad. While this is always a possibility, a more refined analysis shows that (assuming $n$ is fairly large) the probability of poor performance is so insanely small that it is not worth worrying about.

**Treaps:** The intuition behind the treap is easy to understand. Recall back when we discussed standard (unbalanced) binary search trees that if keys are inserted in *random order*, the expected height of the tree is $O(\log n)$. The problem is that your user may not be so accommodating to insert keys in this order. A treap is a binary search tree whose structure arises "as if" the keys had been inserted in random order.

Huh? How is that possible? Let's recall how standard binary tree insertion works. When a new key is inserted into such a tree, it is inserted at the leaf level. If we were to label each node with a "timestamp" indicating its insertion time, as we follow any path from the root to a leaf, the timestamp values must increase monotonically (see Fig. 108(b)). We know of a data structure that has this very property—a *heap*!

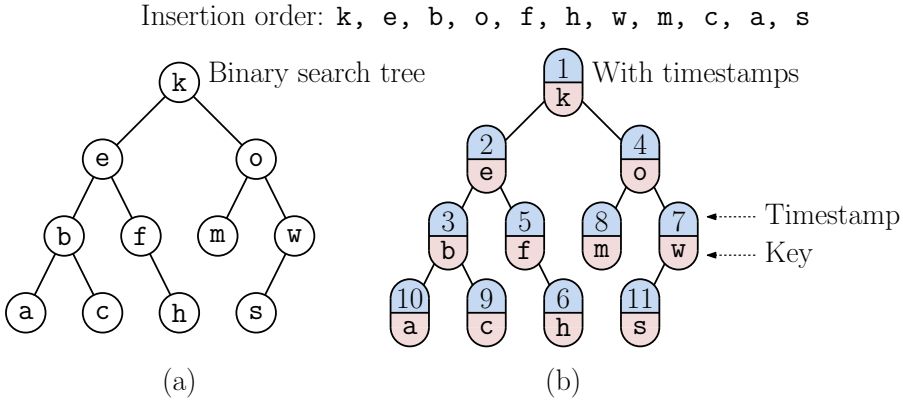



Fig. 108: (a) A binary search tree and (b) associating insertion timestamps with each node.

This suggests the following simple idea: When first inserted, each key is assigned a *random priority*, call it `p.priority`. As in a standard binary tree, keys are sorted according to an inorder traversal. But, the priorities are maintained according to heap order. Since the priorities are random, it follows that the tree's structure is consistent with a tree resulting from a sequence of random insertions. Thus, we have the following:

**Theorem:** A treap storing $n$ nodes has height $O(\log n)$ in expectation. (Here, the expectation is over all $n!$ possible orderings of the random priorities present in the tree.)

Since priorities are random, you might wonder about possibility of two priorities being equal. This might happen, but if the domain of random numbers if much larger than $n$ (say at least $n^2$) then these events are so rare that they won't affect the tree's performance. We will show that it is possible to maintain this structure quite easily.

**Geometric Interpretation:** While Seidel and Aragon designed the treap as a 1-dimensional search structure, the introduction of numeric priorities suggests that we can interpret each key-priority pair as a point in 2-dimensional space. We can visualize a treap as a subdivision of 2-dimensional space as follows. Place all the points in rectangle, where the $y$-coordinates

(ordered top to bottom) are the priorities and the $x$-coordinates are the keys, suitably mapped to numbers (see Fig. 109). Now, draw a horizontal line through the root. Because there are no points of lower priority, all the other points lie in the lower rectangle. Now, shoot a vertical ray down from this point. This splits the rectangle in two, with the points of the left subtree lying in the left rectangle and the points of the right subtree lying in the right rectangle. Now, repeat the process recursively on each of the two halves.

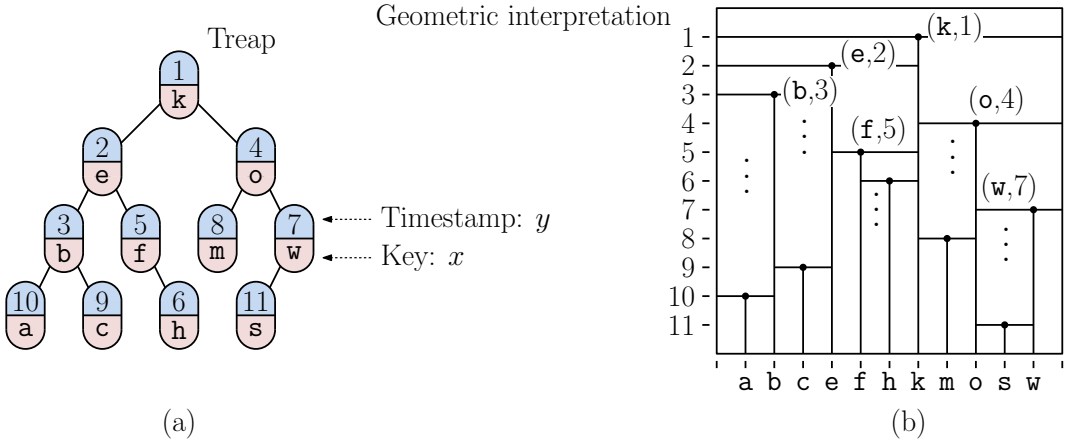



Fig. 109: (a) A treap and (b) geometric interpretation as a (randomized) priority search tree.

The resulting subdivision is used by a geometric data structure called the *priority search tree*, which can be used for answering various geometric range-searching queries in 2-dimensional space.

**Treap Insertion:** Insertion into the treap is remarkably simple. First, we apply the standard binary-search-tree insertion procedure. When we "fall out" of the tree, we create a new node $p$, and set its priority, `p.priority`, to a random integer. We then retrace the path back up to the root (as we return from the recursive calls). Whenever we come to a node $p$ whose child's priority is smaller than $p$'s, we apply an appropriate single rotation (left or right, depending on which child it is), thus reversing their parent-child relationship. We continue doing this until the newly inserted key node is lifted up to its proper position in heap order. The code is very simple and is given below. **Beware:** This is different from the sift-up operation used in the heap data structure. Rotations are needed to maintain the key ordering.

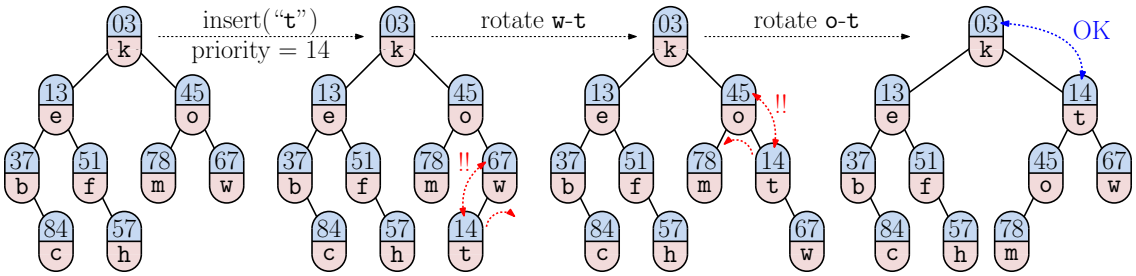



Fig. 110: Treap insertion.

**Why Not Sift?** You might recall that when working with binary heaps, we used operations `sift-up` and `sift-down` to put nodes in proper heap order. Why not here? The issue is

that these operations do not preserve the inorder ordering of keys required by all binary search trees.

**Treap Deletion (Replacement approach):** Deletion is also quite easy, but as usual it is a bit more involved than insertion. If the deleted node is a leaf or has a single child, then we can remove it in the same manner that we did for binary trees, since the removal of the node preserves the heap order. However, if the node has two children, then normally we would have to find the replacement node, say its inorder successor and copy its contents to this node. The newly copied node will then be out of priority order, and rotations will be needed to restore it to its proper heap order. (The code presented at the end of the lecture notes uses this approach.)

**Treap Deletion (Adjustment approach):** Here is another approach for performing deletions, which is a bit slower in practice but is easier to describe. We first locate the node in the tree and then set its priority to $\infty$ (see Fig. 111). We then apply rotations to "sift it down" the tree to the leaf level, where we can easily unlink it from the tree.

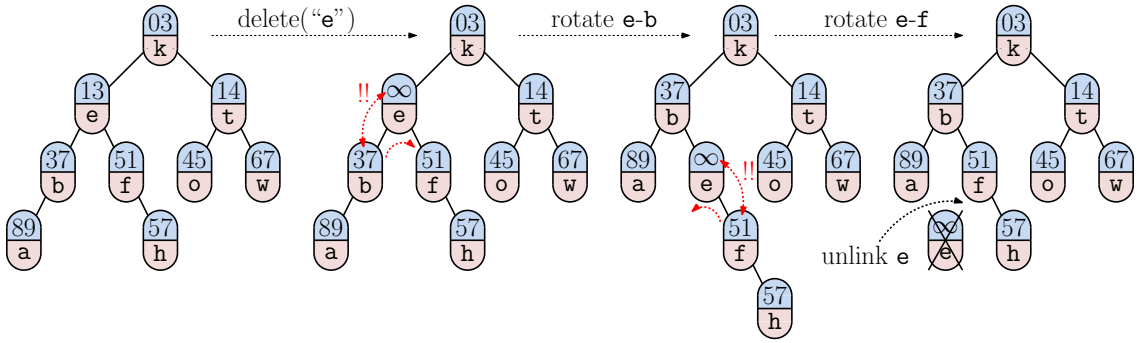



Fig. 111: Treap deletion.

**Performance:** As mentioned above, a treap behaves in the same manner as an unbalanced binary search tree. This means that the expected height is $O(\log n)$. In terms of constant factors, this is pretty close to $2 \ln n \approx 1.4 \lg 2)$. Thus, all the dictionary operations are very fast. The treap is particularly easy to implement because we never have to worry about adjusting the priority fields. For this reason, treaps are among the fastest balanced tree-based dictionary structures.

**Implementation:** We can implement a treap by modifying our implementation of other binary search trees. First, the node structure is similar to that of a standard (unbalanced) binary search tree, but we include the priority value, called `priority`:

Treap Node Structure

```
private class TreapNode {
    Key key                 // key
    Value value             // value
    final int priority      // random priority (set on creation and never changed)
    TreapNode left          // left child
    TreapNode right         // right child
}
```

The right and left rotation functions are the same as for AVL trees (and we omit them). We introduce three utility functions:

- `getPriority(p)`: If `p` is not `null` it returns the node's priority, and $\infty$ otherwise.
- `lowestPriority(p)`: Returns the node `p`, `p.left`, and `p.right` with the lowest priority.
- `restructure(p)`: Restructure the tree locally about `p` by rotating up a child having lower priority than `p`.

Treap Restructuring Operations

```
int getPriority(TreapNode p) { return (p == null ? MAX_PRIORITY : p.priority)  }

TreapNode lowestPriority(TreapNode p) {      // lowest priority of p, p.left, p.right
    TreapNode q = p
    if (getPriority(p.left) < getPriority(q))  q = p.left
    if (getPriority(p.right) < getPriority(q)) q = p.right
    return q
}

TreapNode restructure(TreapNode p) {         // restore priority at p
    if (p == null) return p                  // nothing to do
    TreapNode q = lowestPriority(p)          // get child to rotate
    if (q == p.left) p = rotateRight(p)      // rotate as needed
    else if (q == p.right) p = rotateLeft(p)
    return p                                 // return updated subtree
}
```

**Insertion:** The insert function has exactly the same form the insert function for standard (unbalanced) binary search trees, but it invokes `restructure` as it returns up the tree. Notice that once we get to a node where a rotation is not needed, it will not be needed at any higher node.

Treap Insertion

```
TreapNode insert(Key x, Value v, TreapNode p) {
    if (p == null)                            // fell out of the tree?
        p = new TreapNode(x, v, Math.random()) // leaf with random priority
    else if (x < p.key)                       // x is smaller?
        p.left = insert(x, v, p.left)         // ...insert left
    else if (x > p.key)                       // x is larger?
        p.right = insert(x, v, p.right)       // ...insert right
    else
        Error - Duplicate key!
    return restructure(p)                     // restructure (if needed)
}
```

**Deletion:** As mentioned above, there are two approaches for deletion. We present the method based on adjusting the key value to $+\infty$ and rotating it down to the leaf. We first apply the standard recursive process to locate the node containing the key to be deleted. When we find it, we set its priority to $+\infty$ and invoke the utility `rotateDown`, which rotates `p` down to the the leaf level. This utility function finds the smaller of the two children and performs

a rotation to bring this child up and push p down. We then invokve the function recursively on p.

```
TreapNode delete(Key x, TreapNode p) {
    if (p == null) Error - Nonexistent key!               // key not found
    else {
        if      (x < p.key) p.left  = delete(x, p.left)   // delete from left
        else if (x > p.key) p.right = delete(x, p.right)  // delete from right
        else {                                            // found it!
            p.priority = +INFINITY
            return rotateDown(p)
        }
    }
    return p
}

TreapNode rotateDown(TreapNode p) {                       // rotate p down to leaf
    if (p.left == null && p.right == null) return null    // leaf?...unlink it
    else {
        TreapNode q = lowestPriority(p)                   // get lower child
        if (q == p.left) {
            rotateRight(p)                                // rotate p down on right
            q.right = rotateDown(p)
        }
        else {
            rotateLeft(p)                                // rotate p down on left
            q.left = rotateDown(p)
        }
        return q
    }
}
```

## Lecture 18: Tries and Digital Search Trees

**Strings and Digital Data:** In earlier lectures, we studied binary search trees, which store keys from an ordered domain. Each node stores a splitter value, and we decide whether to visit the left or right subtree based on a comparison with the splitter. Many times, data is presented in digital form, that is, as a sequence of binary bits. These may be organized into groups, for example as characters in a string. When data is presented in this form, an alternative is to design trees that branch in a radix-based manner on these digital values. This can be advantageous with respect to the data structure's speed and the simplicity of performing update operations. Generically, we refer to these structures as *digital search trees*. In this lecture, we will investigate a few variations on this idea.

**Tries:** The trie (pronounced "try") and its variations are widely used for storing string data sets. Tries were introduced by René de la Briandais in 1959, and the term "trie" was later coined by Edward Fredkin, derived from the middle syllable of the work "re**trie**val." (It is said that Fredkin pronounced it the same as "tree," but it was changed to "try" to avoid confusion.)

Throughout, let $\Sigma$ denote our set of symbols, which we call our *alphabet*. For example, $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. Let $k = |\Sigma|$ denote the number of letters in our alphabet. In its simplest

form, each internal node of a trie has $k$ children, one for each letter of the alphabet. We cast each character to an integer in the range $\{0, \ldots, k-1\}$, which we can use as an index to an array to access the next child. This way, we can search for a word of length $m$ by visiting $m$ nodes of our tree. Let us also assume Java's convention that we index the characters of a string starting with 0 as the first (leftmost) character of the string. Given a string `str`, we will refer to its leftmost character as `str[0]`, and in Java this would be `str.charAt(0)`.

An example is shown in Fig. 112, where we store a set of strings over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ where $\texttt{a} = 0$, $\texttt{b} = 1$, and $\texttt{c} = 2$.

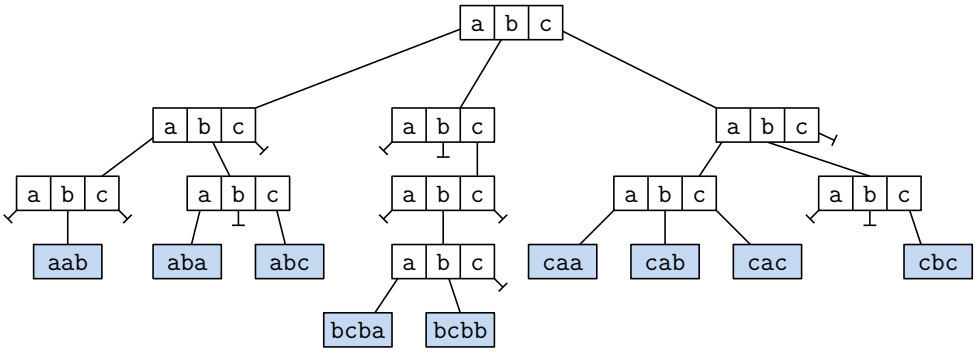



Fig. 112: A trie containing the set $\{\texttt{aab}, \texttt{aba}, \texttt{abc}, \texttt{bcba}, \texttt{bcbb}, \texttt{caa}, \texttt{cab}, \texttt{cac}, \texttt{cbc}\}$.

We would like each distinct search path to terminate at a different leaf node. This is not generally possible when one string is a prefix of another. A simple remedy is to add a special *terminal character* to the end of each string. (Later, we will use `$` as our terminal character, but for now we will simply avoid storing keys where one is a prefix of another.)

The drawing shown in Fig. 112 is true to the trie's implementation, but this form of drawing is rather verbose. When drawing tries and other digital search trees, we will adopt the manner shown below, where we label edges with the character. But remember that this is not a different representation or a different data structure, it is just a different way of drawing the tree.

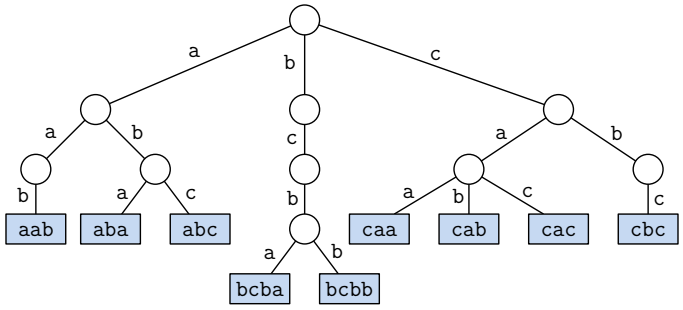



Fig. 113: Alternative (but equivalent) method of drawing the trie of Fig. 112.

**Analysis:** It is easy to see that we can search for a string in time proportional to the number of characters in the string. This can be a significant savings over binary search trees. For example, suppose that you wanted to store a dictionary of English words. A large dictionary can have around 500,000 entries, and $\log_2 500,000 \approx 19$, but the length of the average English word is well less than 10.

Space is an issue, however. In the worst case, the number of nodes in the data structure can be as high as the total number of characters in all the strings. Observe, for example, that if we have one key containing 100 characters, we must devote a search path for this string, which will require 100 nodes and $100k$ total space, where $k = |\Sigma|$. Clearly, this is an absurd situation, but it points to the main disadvantage of tries, namely their relatively high space requirements. (By the way, if you wanted to perform exact-match searches, hashing would be both more time and space efficient.)

**de la Briandais Trees:** One idea for saving space is to convert the $k$-order trie into a form that is similar to the first-child/next-sibling representation that we introduced for multiway trees. A node in this structure stores a single character. If the next character of the string matches, we descend to the next lower level and advance to the next character of the search string. Otherwise, we visit the next node at this same level. Eventually, we will either reach the desired leaf node or else we will run off the end of some level. If the latter happens, then we report that the key is not found. These are called *de la Briandais trees* (see Fig. 114).

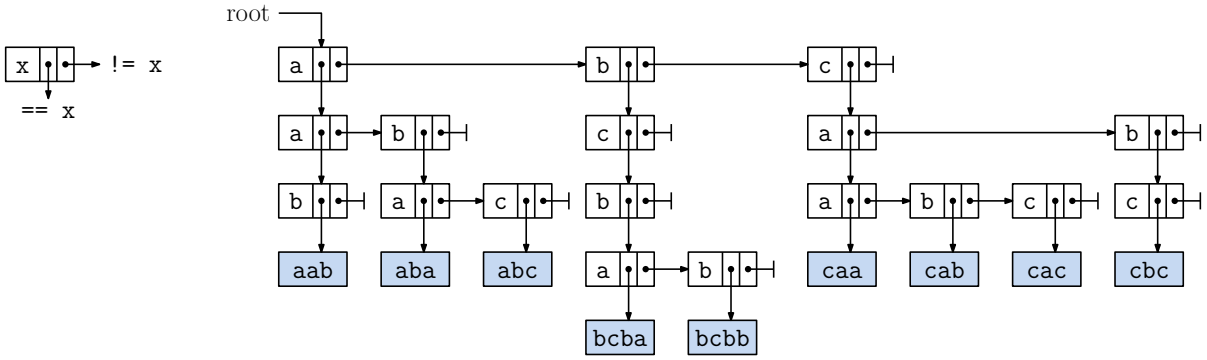



Fig. 114: A de la Briandais tree containing the same strings as in Fig. 112.

The de la Briandais tree trades off a factor of $k$ in the space for a factor of $k$ in the search time. While the number of nodes can be as large as the total number of characters in all the strings, the size of each node is just a constant, independent of $k$. In contrast, the search time can be as high as $O(k)$ per level of the tree, in contrast to $O(1)$ per level for the traditional trie.

**Patricia Tries:** One issue that arises with tries and digital search trees is that we may have long paths where the contents of two strings are the same. Consider a very degenerate situation where we have a small number of very long keys. F, that we have the keys "`dysfunctional`" and "`dysfunctioning`" (see Fig. 115). The standard trie structure would have a long sequence where there is no branching. This is very wasteful considering that each of these nodes requires $O(k)$ space, where $k$ is the size of our dictionary.

When keys are closely clustered (in the sense that they share lengthy common prefexes), digital search trees can suffer from this phenomenon. The solution is to perform *path compression*, which succinctly encodes long degenerate paths. This is the idea behind a trie variant called a *patricia trie*. (The word 'patricia' is an acronym for *Practical Algorithm To Retrieve Information Coded In Alphanumeric*.) This concept was introduced in the late 1960's by Donald R. Morrison, and was independently discovered by Gernot Gwehenberger.

A patricia trie uses the same node structure as the standard trie, but in addition each node contains an *index field*. This field is the index of the *discriminating character*, which is used

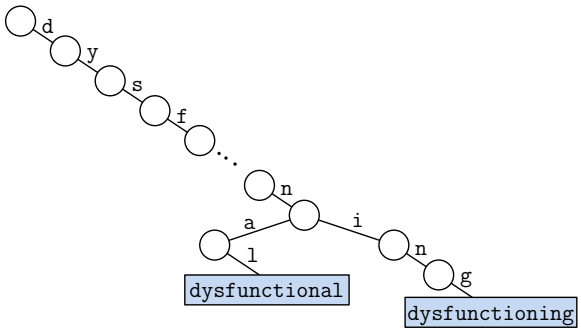



Fig. 115: Space wastage due to long degenerate paths in traditional tries storing "`dysfunctional`" and "`dysfunctioning`".

to determine how to branch at this node. This index field value increases as we descend the tree. (A traditional trie can be viewed as a special case where this index increases by one with each level.) In the patricia trie, the index field is set to the next index such where there is a nontrivial split (that is, the next level where the node has two or more children). Note that once the search path has uniquely determined the string of the set, we just store a link directly to the leaf node. An example is shown in Fig. 116. Observe we start by testing the 0th character. All the strings that start with "`e`" are on the left branch from the root node. Since they all share the next symbol in common, we continue the search with the character at index 2.

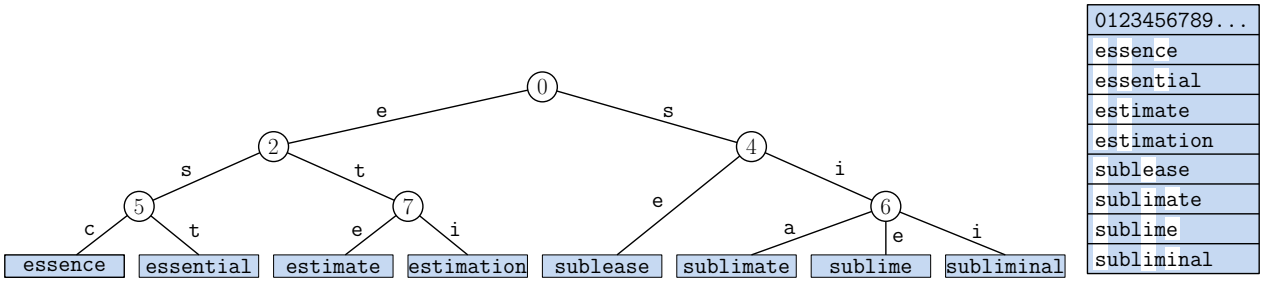



Fig. 116: A patricia trie for a set of strings. Each node is labeled with the index of the character and each edge is labeled with the matching character for this branch.

As we did with standard tries, there is a more intuitive way of drawing patricia tries. Rather than listing the index of the discriminating character using for branching, we instead list the entire substring from the (see Fig. 117). This is convenient because we can read the substrings from the drawing without having to refer back to the original strings. As in the previous case, this is not a different representation or a different data structure, just a different drawing.

**Analysis:** The patricia trie is superior to the standard trie in terms of both worst-case space and worst-case query time. For the space, observe that because the tree splits at least two ways with each node, and easy induction argument shows that the total number of nodes is proportional to the number of strings (not the total number of characters in these strings, which was the case for the standard trie). As with the standard trie, the worst-case search time is equal to the length of the query string, but note that it may generally be much smaller, since we only query character positions that are relevant to distinguishing between two keys of the set.

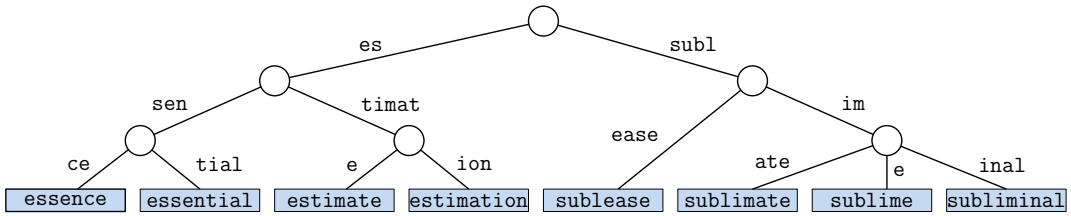



Fig. 117: Alternative (but equivalent) method of drawing the patricia trie of Fig. 116.

**Suffix trees:** So far, we have been considering data structures for storing a set of strings. Another common application involves storing a single, very long string, for which we want to perform substring queries. For example, "Given the string "`abracadabra`", how often does the substring "ab" occur?" Such queries are useful in applications such as genomics, where queries are applied to a genome sequence.

Consider a string $S =$ "$a_0 a_1 \ldots a_{n-1}$\$", which we call the *text*. For $0 \leq i \leq n$, define the *ith suffix* to be the substring $S_i =$ "$a_i a_{i+1} \ldots a_{n-1}$\$". We have intentionally placed a special terminator character "\$" at the end of the string so that every suffix is distinct from any other substring appearing in $S$.

For each position $i$, $0 \leq i \leq n$, there is a minimum length substring starting at index $i$ that uniquely identifies $S_i$. For example, in the string "`yabbadabbadoo$`", there are two suffixes ($S_1$ and $S_6$) that start with "`abbad`", and so this substring does not uniquely identify a suffix. But there is only one suffix (namely $S_6$) that starts with "`abbado`", and therefore this substring is minimum length unique identifier of a suffix.

To make this more formal, for $0 \leq i \leq n$, define the $i$th *substring identifier*, denoted $\text{id}_i$, to be the shortest substring starting at index $i$ that is unique among all substrings in the text string. for position $i$. Note that because the end of string character is unique, every position has a unique substring identifier. An example is shown in the following figure. The set of all substring identifiers for this string are shown on the left side of Fig. 118.

A *suffix tree* for a text $S$ is a patricia trie in which we store each of the $n + 1$ substring identifiers for the string $S$. We illustrate this on the right side of Fig. 118. We have adopted the convention from Fig. 117 for drawing the patricia trie. Each leaf of the tree is associated with the suffix $S_i$ it identifies, and it is labeled with the associated index $i$.

| Index | Substring ID | Index | Substring ID |
|---|---|---|---|
| 0 | y | 7 | bbado |
| 1 | abbada | 8 | bado |
| 2 | bbada | 9 | ado |
| 3 | bada | 10 | do |
| 4 | ada | 11 | oo |
| 5 | da | 12 | o$ |
| 6 | abbado | 13 | $ |

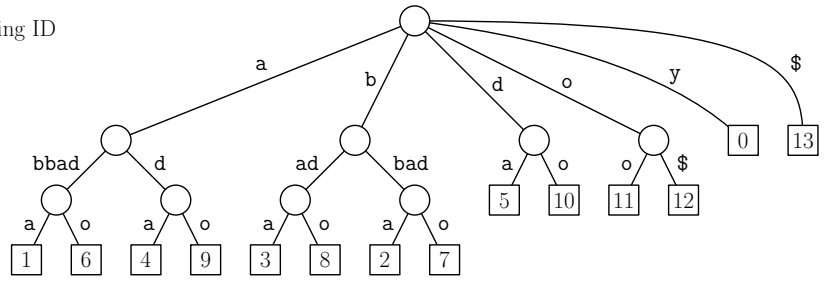



Fig. 118: A suffix tree for the string "`yabbadabbadoo$`". (Labels are placed on the edges, but this can be implemented using a standard trie or patricia trie.)

**Efficient Construction:** Generally, the size and construction time of a patricia trie are both proportional to the total length of the strings being inserted. If you are given a text $S$ of

length $n$, there are $n$ suffixes of lengths $0, 1, 2, \ldots, n$. Thus, the total number of characters over all suffixes is $\sum_{i=1}^{n} i = O(n^2)$. This would suggest that the suffix tree has size $O(n^2)$ and takes $O(n^2)$ time to compute. While we will not present the proof, it can be shown that, due to the the special structure of the suffix tree, it has $O(n)$ size, and it can be built in $O(n)$ time.[9]

By the way, suffix trees are not usually the method of choice when storing suffix information. There is a related data structure, called a *suffix array*, which encodes the same information more succinctly (by a constant factor), and is the method that is usually used in practice when answering the same sorts of queries.

**Suffix-Tree Queries:** As an example of answering a query, suppose that we want to know how many times the substring "abb" occurs within the text string $S$. To do this we search for the string "abb" in the suffix tree. If we fall out of the tree, then it does not occur. Otherwise the search ends at some node $u$. The number of leaves descended from $u$ is equal to the number of times "abb" occurs within $S$.

In our example from Fig. 118, the search ends on the leftmost path in the tree, midway along the edge labeled "`bbad`". Since the subtree rooted here has two nodes, the answer to the query is 2. By traversing the entire subtree, we can report that the two instances start at indices 1 and 6. (Recall that we index the string starting with 0.)

**Geometric Digital Search Trees:** Before leaving the topic of digital search trees, we should mention an interesting connection to geometric data structures. In an earlier lecture we discussed point quadtree and point kd-trees as two ways of storing geometric point sets. We can extend the idea of digital search trees to the geometric setting. We will do this in a 2-dimensional setting, but the generalization to arbitrary dimensions is straightforward.

Our approach will be to define a transformation that maps a 2-dimensional point $(x, y)$ into a string, in a manner that preserves geometric structure. To do this, let us first assume that we have applied a scaling transformation so our point coordinates lie within the interval $[0, 1)$. (For example, we can add a sufficiently large number that our coordinates are positive, and then we can scale by dividing by the largest possible coordinate value.) By doing this, we may assume that each point $(x, y)$ satisfies $0 \leq x, y < 1$.

Our next step is to represent coordinate as a binary fraction. For example, the decimal point $(x, y) = (0.356, 0.753)$ can be represented in binary form as $(0.01011\ldots, 0.11000\ldots)$. We can treat the binary strings $x = 01011\ldots$ and $y = 11000\ldots$ as strings over the 2-symbol alphabet $\Sigma = \{0, 1\}$.

But, how do we convert two coordinates into a single digital string? Recall that in kd-trees, we alternated splitting on $x$ and then $y$. This suggests that we can map two coordinates $x$ and $y$ into a single binary string by *alternating* their binary bits. More formally, if $x = 0.a_1a_2a_3\ldots$ and $y = 0.b_1b_2b_3\ldots$ in binary, we create the binary string $a_1b_1a_2b_2a_3b_3\ldots$. In our previous example, given our point $(x, y) = (0.356, 0.753)$, we would interleave the bits of their binary fractions to obtain the binary string "`0111001010...`". We we refer to this as the *bit interleaving transformation*. Let us apply this transformation to every point in our 2-dimensional point data set. We can then store the resulting binary strings in a digital search tree, like a patricia trie.

[9] Recall that the size of a patricia tree is proportional to the number of strings, irrespective of their sizes. This directly bounds the suffix tree's size. The trick for constructing the tree efficiently is to start with the last suffix $S_n$ = "`$`" and work backwards to the first suffix, using the partially built tree to assist in the construction.

Wow! Is this crazy transformation from geometric data to digital data meaningful in any way? It turns out that not only is meaningful, it actually corresponds to one of the most fundamental geometric geometric data structures, called a *PR kd-tree*. In a 2-dimensional PR kd-tree, we assume that the data lies within a square. Let's assume that this square is our unit square $[0,1)^2$. At the root, we split vertically through the midpoint of this square, creating two children (West and East). Each child is associated with a rectangular cell. For each child, if its cell contains two or more points, we split the cell horizontally through its midpoint into two cells (South and North). We repeat the process, splitting alternately between $x$ and $y$, always splitting through the midpoint of the current cell, until the cell has either zero or one points. An example of the resulting subdivision of space is shown in Fig. 119(a).

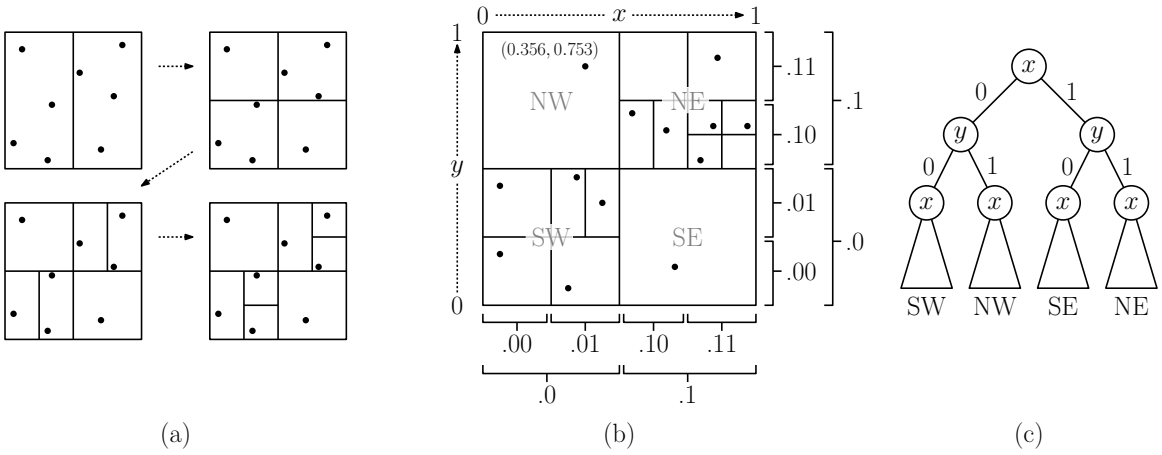



Fig. 119: The PR kd-tree as a digital search tree using bit interleaving.

Observe that after two levels of splitting in our PR kd-tree, we have subdivided the original square into four quadrants, each of half the side length of the original square. We can label these SW, NW, SE, and NE (see Fig. 119(b)). Let's consider what can be said about the points lying in any one of these quadrants. For concreteness let's consider the point with coordinates $(0.356, 0.753)$ in the NW quadrant (see Fig. 119(b)). Since it lies in this quadrant we know that the leading bits of its $x$-coordinate is 0 (smaller than half) and the leading bit of its $y$-coordinate is 1 (larger than half). Therefore, its digital code starts with "`01...`". Observe that our digital tree will put this point and all points in the NW quadrant into the 01 grandchild subtree of the root. It is easy to verify that each of the four grandchildren of the root correspond to each of the four quadrants. Furthermore, as we descend the tree, with each two levels, the digital search tree partitions the points into subtrees in exactly the same manner as the PR kd-tree does.

In summary, the PR kd-tree data structure, is equivalent to a digital search tree for the points after applying the bit-interleaving transformation! While we think of a patricia tree as a data structure for storing strings, it turns out that this data structure can also be interpreted as geometric data structure, and it can be used for answering most of the same queries (such as orthogonal range and nearest neighbor queries) that the kd-tree can answer.

# Lecture 19: Memory Management

**Memory Management:** One of the major systems issues that arises when dealing with data structures is how storage is allocated and deallocated as objects are created and destroyed. Although *memory management* is really a operating systems issue, we will discuss this topic because there are a number interesting data structures issues that arise. Sometimes for the sake of efficiency, it is desirable to design a special-purpose memory management system for your data structure application, rather than using the system's memory manager.

**System-Level:** We will not discuss the issue of how the runtime system maintains memory in great detail. Basically what you need to know is that there are two principal components of dynamic memory allocation. The first is the *stack*. When procedures are called, arguments and local variables are pushed onto the stack, and when a procedure returns these variables are popped. The stacks grows and shrinks in a very predictable way.

In contrast, whenever objects are allocated through the Java "`new`" operator, they are stored in a different section of memory called the *heap*. (In spite of the similarity of names, this heap has nothing to do with the binary heap data structure, which is used for priority queues.) As elements are allocated and deallocated, the heap storage becomes fragmented into pieces. How to maintain the heap efficiently is the main issue that we will consider.

**Approaches:** There are two basic aproaches of doing memory management. This has to do with whether storage deallocation is done *explicitly* or *implicitly*. Explicit deallocation is used in languages like C (resp., C++). Memory is allocated through `malloc` (resp., `new`), and is explicitly released to the system by invoking `free` (resp., `delete`). While these systems provide the user a high degree of control, they also impose a strong burden on the programmer. Blocks of memory may be *leaked* in the sense that the memory is inaccessible, and has not been deallocated. A more significant programming bug is the result of *aliasing*, where (unknown to the programmer) two pointers reference the same block of memory. (This often happens when a *shallow copy* results in a pointer being copied, rather than making a copy of the underlying object.) Now, when one of these pointers is used to release the block, the other pointer still references this chunk of released memory.

In contrast languages like Java uses implicit deallocation. It is up to the system to determine which objects are no longer accessible and reclaim their storage. This process is called *garbage collection*. In both cases there are a number of choices that can be made, and these choices can have significant impacts on the performance of the memory allocation system. Unfortunately, there is not one system that is best for all circumstances. We begin by discussing explicit deallocation systems.

**Explicit Allocation/Deallocation:** There is one case in which explicit deallocation is very easy to handle. This is when all the objects being allocated are of the same size. A large contiguous portion of memory is used for the heap, and we partition this into *blocks* of size $b$, where $b$ is the size of each object. For each unallocated block, we use one word of the block to act as a *next* pointer, and simply link these blocks together in linked list, called the *available space list*. For each `alloc` request, we extract a block from the available space list and for each `dealloc` we return the block to the list.

If the records are of different sizes then things become much trickier. We will partition memory into blocks of varying sizes. Each block (allocated or available) contains information indicating how large it is. Available blocks are linked together to form an available space list.

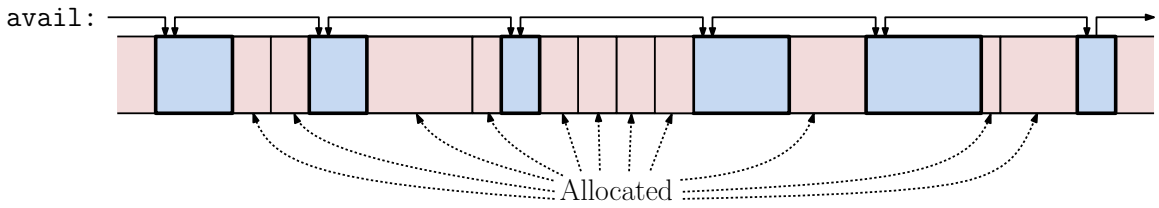



Fig. 120: Dynamic memory allocation block structure.

The main questions are: (1) what is the best way to allocate blocks for each `alloc` request, and (2) what is the fastest way to deallocate blocks for each `dealloc` request.

The biggest problem in such systems is the fact that after a series of allocation and deallocation requests the memory space tends to become *fragmented* into small blocks of memory. This is called *external fragmentation*, and is inherent to all dynamic memory allocators. Fragmentation increases the memory allocation system's running time by increasing the size of the available space list, and when a request comes for a large block, it may not be possible to satisfy this request, even though there is enough total memory available in these small blocks. Observe that it is not usually feasible to compact memory by moving fragments around. This is because there may be pointers stored in local variables that point into the heap. Moving blocks around would require finding these pointers and updating them, which is a very expensive proposition. We will consider it later in the context of garbage collection. A good memory manager is one that does a good job of controlling external fragmentation.

**Overview:** When allocating a block we must search through the list of available blocks of memory for one that is large enough to satisfy the request. The first question is, assuming that there does exist a block that is large enough to satisfy the request, what is the best block to select? There are two common but conflicting strategies:

**First-fit:** Search the available blocks sequentially until finding the first one that is large enough to satisfy the request.

**Best-fit:** Search all available blocks and select the smallest block that is large enough to fulfill the request.

Both methods work well in some instances and poorly in others. Remarkably, in spite of its name, Best-fit often performs worse in practice than First-fit. The reasons are twofold. First, First-fit is faster to execute, since the search can stop at the first block of sufficiently large size, rather than having to search the entire available list. Second, best-fit tends to produce a good deal of fragmentation by selecting blocks that are just barely larger than the requested size, but this results in a small residual block, or *sliver*, left over, and there may not be any future requests that are small enough that this small block can be used.

One method which is commonly used to reduce fragmentation is when a request is just barely filled by an available block that is slightly larger than the request, we allocate the entire block (more than the request) to avoid the creation of the sliver. This keeps the list of available blocks free of large numbers of tiny fragments which increase the search time. The additional waste of space that results because we allocate a larger block of memory than the user requested is called *internal fragmentation* (since the waste is inside the allocated block).

When deallocating a block, it is important that if there is available storage we should merge the newly deallocated block with any neighboring available blocks to create large blocks of

free space. This process is called *merging*. Merging is trickier than one might first imagine. For example, we want to know whether the preceding or following block is available. How would we do this? We could walk along the available space list and see whether we find it, but this would be very slow. We might store a special bit pattern at the end and start of each block to indicate whether it is available or not, but what if the block is allocated that the data contents happen to match this bit pattern by accident? Let us consider the implementation of this scheme in greater detail.

**Notation and Assumptions:** Memory is usually allocated in bytes, but it is common to align all allocated blocks to a *word* (typically 32-bits) or a *double-word* (typically 64 bits) boundary. This way, we don't need to know what the data is being used for (bytes, ints, floats, doubles, etc.). In our examples, we will make the simplifying assumption that requests for memory allocation are given in terms of a number of words, and the block of memory that we return will be aligned at a word boundary. Given a pointer `p` to memory, we will use `*p` to refer to the contents of the word of memory at this address. Given a pointer `p` and integer $i$, we will use `p + i` to refer to the memory location that is $i$ words beyond `p`. We will use the expression `void*` to indicate the "type" of a pointer to a location of physical memory.

**Block Structure:** The main issues are how to we store the available blocks so we can search them quickly (skipping over used blocks), and how do we determine whether we can merge with neighboring blocks or not. We want the operations to be efficient, but we do not want to use up excessive pointer space (especially in used blocks). Here is a sketch of a solution (one of many possible).

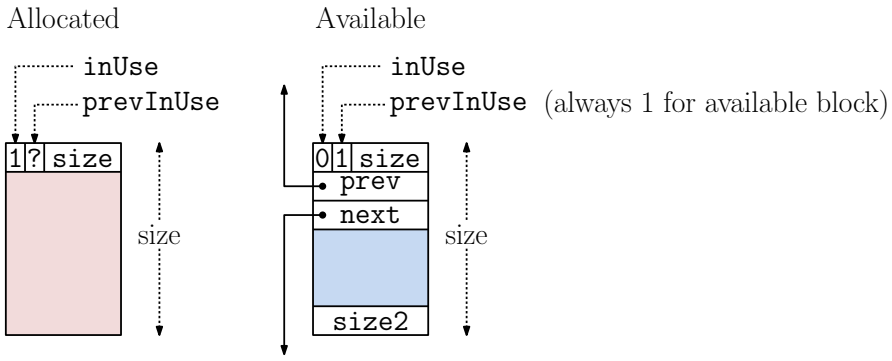



Fig. 121: Block structure for dynamic storage allocation.

**Allocated blocks:** For each block of used memory we record the following information. It can all fit in the first word of the block.

`size`: The size of the block of storage (including space for the following fields).

`inUse`: A single bit that is set to 1 (true) to indicate that this block is in use.

`prevInUse`: A single bit that is set to 1 (true) if the previous block is in use and 0 (false) otherwise. (Later we will see why this is useful.)

**Available blocks:** For an available block we store more information, which is okay because the user is not using this space. These blocks are stored in a doubly-linked circular list, called `avail`. Note that the available space list need not sorted by physical memory addresses. For example, it might be ordered by some other criteria, such as the sizes of the blocks.

`size`: The size of the block of storage (including space for the following fields).

`inUse`: A bit that is set to 0 (false) to indicate that this block is not in use.

`prevInUse`: A bit that is set to 1 (true) if the immediately preceding block (not the same as `prev`) is in use (which should always be true, since we should never have two consecutive unused blocks).

`prev`: A pointer to the previous block on the available space list. (It need not be the block immediately preceding in memory.)

`next`: A pointer to the next block on the available space list. (It need not be the block immediately following in memory.)

`size2`: Contains the same value as `size` at the head of the block. This field is stored in the *last* word of the block. For block $p$ it can be accessed as `*(p + p.size - 1)`.

Note that available blocks require more space for all this extra information. The system will never allow the creation of a fragment that is so small that it cannot contain all this information.

You will observe that the run-time system trusts that the user's program will not overwrite any of the block header fields or previous/next pointers. What is to keep this from happening? The answer (at least with C and C++) is *nothing*! If the program accidentally overwrites a memory location outside of the allocated block, it can destroy the memory system's integrity, and very soon everything fails. Usually, the result is an abort with just a cryptic message, such as "Segmentation fault." Failure to check for these faults can result in undetected *buffer-overflow* writes, a common technique for hacking into software systems.

**Allocation:** To allocate a block we search through the linked list of available blocks until finding one of sufficient size (see Fig. 122). If the request is about the same size (or perhaps slightly smaller) as the block, we remove the block from the list of available blocks (performing the necessary relinkings) and return a pointer to it. We also may need to update the `prevInUse` bit of the next block since this block is no longer available. Otherwise we split the block into two smaller blocks, return one to the user, and leave the other on the available space list.

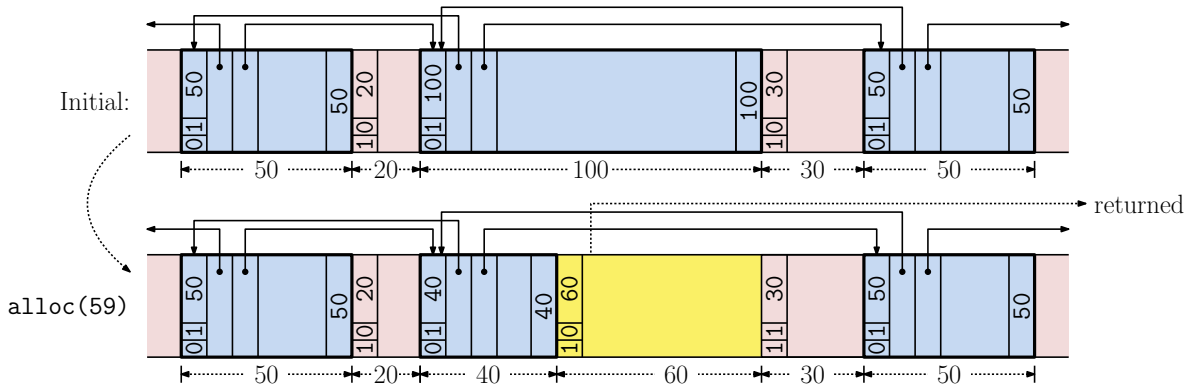



Fig. 122: An example of block allocation.

The following code block presents an algorithm for allocating a new block of memory. Note that we make use of pointer arithmetic here. The argument `b` is the desired size of the allocation. Because we reserve one word of storage for our own use we increment this value on entry to the procedure. We keep a constant `TOO_SMALL`, which indicates the smallest allowable

fragment size. If the allocation would result in a fragment of size less than this value, we return the entire block. The procedure returns a generic pointer to the newly allocated block. The utility function `avail.unlink(p)` simply unlinks block $p$ from the doubly-linked available space list. An example is provided in Fig. 122. Shaded blocks are available.

Allocate a block of storage

```
(void*) alloc(int b) {                          // allocate block with b words
    b += 1;                                     // extra space for system overhead
    p = search available space list for block of size at least b;
    if (p == null) { ...Error! Insufficient memory...}
    if (p.size - b < TOO_SMALL) {               // remaining fragment too small?
        avail.unlink(p);                        // remove entire block from avail list
        q = p;                                  // this is block to return
    }
    else {                                      // split the block
        p.size -= b;                            // decrease size by b
        *(p + p.size - 1) = p.size;             // set new block's size2 field
        q = p + p.size;                         // offset of start of new block
        q.size = b;                             // size of new block
        q.prevInUse = 0;                        // previous block is unused
    }
    q.inUse = 1;                                // new block is used
    (q + q.size).prevInUse = 1;                 // adjust prevInUse for following block
    return q + 1;                               // offset the link (to avoid header)
}
```

**Deallocation:** To deallocate a block, we check whether the next block or the preceding blocks are available. For the next block we can find its first word and check its `inUse` field. For the preceding block we use our own `prevInUse` field. (This is why this field is present). If the previous block is not in use, then we use the size value stored in the last word to find the block's header. If either of these blocks is available, we merge the two blocks and update the header values appropriately. If both the preceding and next blocks are available, then this result in one of these blocks being deleting from the available space list (since we do not want to have two consecutive available blocks). If both the preceding and next blocks are in-use, we simply link this block into the list of available blocks (e.g. at the head of the list).

The deletion function is shown in the following code block. Let `avail` denote the head of the available space list. There are four different cases depending on whether the blocks that immediate precede or follow $p$ are allocated or available.

- If the following block $q$ is available, then we merge it with $p$. (See Fig. 123, upper half.) To do this we update $p$'s size, and move $q$'s record in the available space list to $p$, using a utility function `move(q, p)`. This copies $q$'s previous and next fields to $p$ and appropriately update the entries in the available space list that point to $q$. (E.g., `q.prev.next = p`.) Otherwise we add $p$ to the available space list, which we assume will set its previous and next pointers. Now we may assume that $p$ is on the available space list.
- If the preceding block is not in use, we merge this block with $p$ (see Fig. 123, lower half), and unlink $p$ from the available space list. Note that we do need to alter `p.prevInUse`. If the previous block was available, then we merged $p$ with it and $p$ has gone away, and otherwise $p$'s original value was correct.

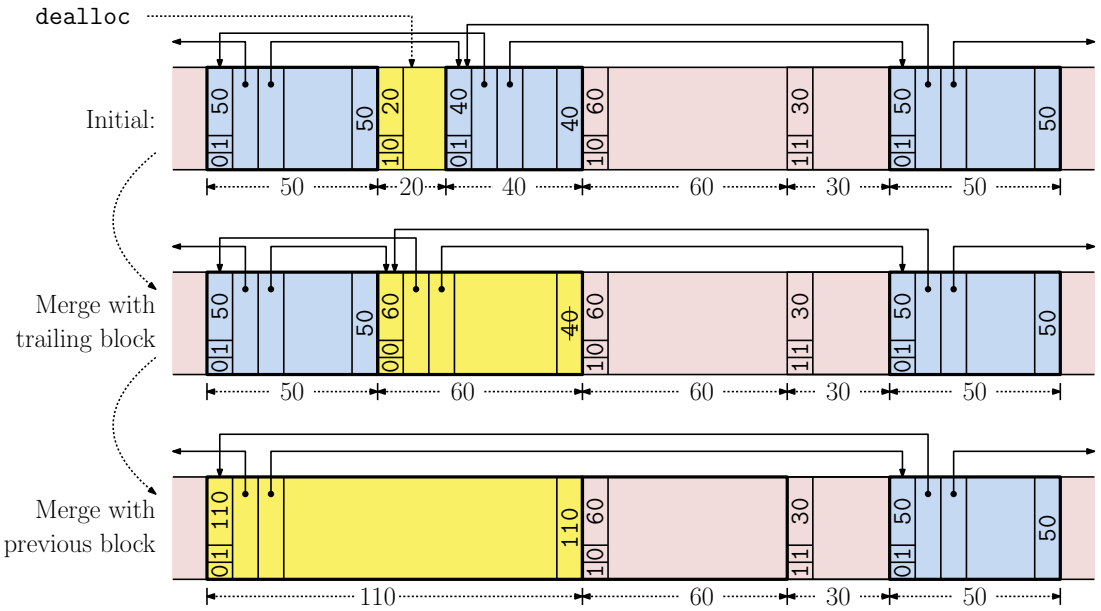



Fig. 123: An example of block deallocation and merging.

**Analysis:** There is not much theoretical analysis of this method of dynamic storage allocation. Because the system has no knowledge of the future sequence of allocation and deallocation requests, it is possible to contrive situations in which either first fit or best fit (or virtually any other method you can imagine) will perform poorly. Empirical studies based on simulations have shown that this method achieves utilizations of around 2/3 of the total available storage before failing to satisfy a request. Even higher utilizations can be achieved if the blocks are small on average and block sizes are similar (since this limits fragmentation). A rule of thumb is to allocate a heap that is at least 10 times larger than the largest block to be allocated.

**Buddy System:** The dynamic storage allocation method described last time suffers from the problem that long sequences of allocations and deallocations of objects of various sizes tends to result in a highly fragmented space. The *buddy system* is an alternative allocation system which limits the possible sizes of blocks and their positions, and so tends to produce a more well-structured allocation. Because it limits block sizes, *internal fragmentation* (the waste caused when an allocation request is mapped to a larger block size) becomes an issue.

The buddy system works by starting with a block of memory whose size is a power of 2 and then hierarchically subdivides each block into blocks of equal sizes (see Fig. 125). To make this intuition more formal, we introduce the two key elements of the buddy system:

(i) The sizes of all blocks (both allocated and available) are powers of 2. When a request comes for an allocation, the request (including the overhead space needed for storing block size information) is artificially rounded up to the next higher power of 2. Note that the allocated size is never more than twice the size of the request.

(ii) Blocks of size $2^k$ start at memory addresses that are multiples of $2^k$. (We assume that addressing starts at 0, but it is easy to update this scheme to start at any arbitrary address by simply shifting addresses by an appropriate offset.)

Note that the above requirements limits the ways in which blocks may be merged. For example Fig. 125 below illustrates a buddy system allocation of blocks, where the blocks of

```
void dealloc(void* p) {                    // deallocate block at p
    p--;                                   // back up to the header
    q = p + p.size;                        // the following block
    if (!q.inUse) {                        // is following available?
        p.size += q.size;                  // merge q into p
        avail.move(q, p);                  // move q to p in avail space list
    }
    else avail.insert(p);                  // insert p into avail space list
                                           // p is now in avail space list
    p.inUse = 0;                           // p is now available
    *(p + p.size - 1) = p.size;            // set our size2 value

    if (!p.prevInUse) {                    // previous is available?
         q = p - *(p-1);                   // get previous block using size2
         q.size += p.size;                 // merge p into q
         *(q + q.size - 1) = q.size;       // store new size2 value
         avail.unlink(p);                  // unlink p from avail space list
         (q + q.size).prevInUse = 0;       // notify next that we are avail
     }
}
```

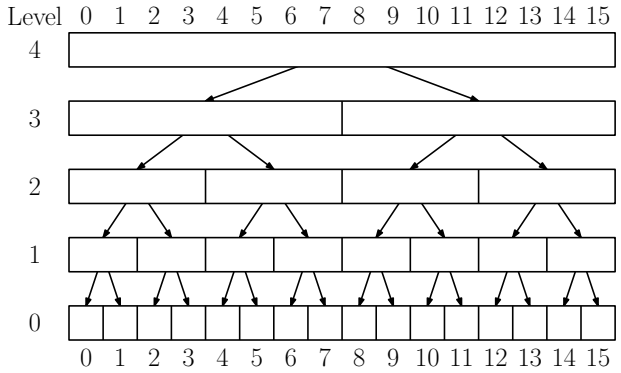



Fig. 124: Buddy system block structure.

size $2^k$ are shown at the same level. Available blocks shaded blue and allocated blocks are shaded red. The two available blocks at addresses 5 and 6 (the two white blocks between 4 and 8) cannot be merged because the result would be a block of length 2, starting at address 5, which is not a multiple of 2. For each size group, there is a separate available space list.

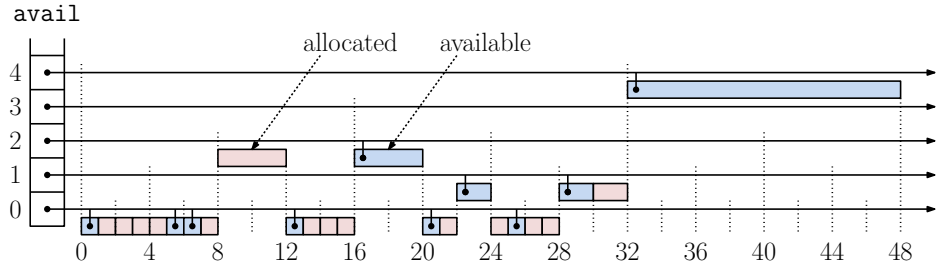



Fig. 125: Buddy system example. Allocated blocks are shaded red and available blocks are shaded blue. Available blocks are linked into the available-block list of the appropriate size.

For every block there is exactly one other block with which this block can be merged with. This is called its *buddy*. In general, if a block $b$ is of size $2^k$, and is located at address $x$, then its buddy is the block of size $2^k$ located at address

$$\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } 2^{k+1} \text{ divides } x \\ x - 2^k & \text{otherwise.} \end{cases}$$

This is easy to compute via bit manipulation. Basically the buddy's address is formed by complementing bit $k$ in the binary representation of $x$ (where the lowest order bit is bit 0). In Java, this can be expressed as `buddy(k,x) = (1<<k)^x`. For example, for $k = 3$ the blocks of length 8 at addresses 208 and 216 are buddies. If we look at their binary representations we see that they differ in bit position 3 (four bits from the right). Because they must be multiples of 8, bits 0–2 are zero.

$$\begin{aligned} 80 &= 1010000_2 \\ 88 &= 1011000_2. \end{aligned}$$

So, $\text{buddy}_3(80) = 88$ and vice versa.

**Putting the Pieces Together:** As we mentioned earlier, one principle advantage of the buddy system is that we can exploit the regular sizes of blocks to search efficiently for available blocks. We maintain an array of linked lists, one for the available block list for each size group. In particular, `avail[k]` is the header pointer to a doubly linked list of available blocks of size $2^k$.

Here is how the basic operations work. We assume that each block has the same structure as described in the dynamic storage allocation example from last time. The `prevInUse` bit and the size field at the end of each available block are not needed given the extra structure provided in the buddy system. Each block stores its size (actually the log base 2 of its size is sufficient) and a bit indicating whether it is allocated or not. Also each available block has links to the previous and next entries on the available space list. There is not just one available space, but rather there are multiple lists, one for each level of the hierarchy (something like a skip list). This makes it possible to search quickly for a block of a given size.

**Buddy System Allocation:** We will give a high level description of allocation and dealloction. To allocate a block of size $b$, let $k = \lceil \lg(b+1) \rceil$. (Recall that we need to include one extra word for the block size. However, to simplify our figures we will ignore this extra word.) We will allocate a block of size $2^k$. In general there may not be a block of exactly this size available, so find the smallest $j \geq k$ such that there is an available block of size $2^j$. If $j > k$, repeatedly split this block until we create a block of size $2^k$. In the process we will create one or more new blocks, which are added to the appropriate available space lists.

For example, in Fig. 126 we request a block of length 2. There are no available blocks of this size, so we use the smallest available block of the next larger size, that is, the block of length 16 at address 32. We remove it from its available space list. We recursively split it into subblocks of sizes 8, 4, 2, 2, until we get to a block of the desired size. We return one of the blocks of size 2, and we insert the remaining blocks (of sizes 2, 4, and 8) into the available space lists of the appropriate sizes.

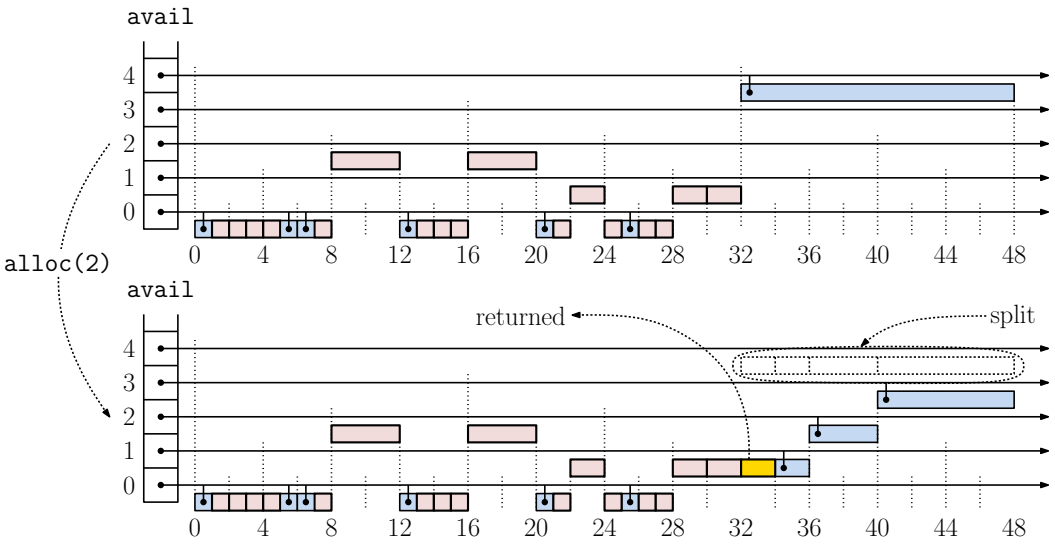



Fig. 126: Example of allocating a block in the buddy system.

**Deallocation:** To deallocate a block, we first mark this block as being available. We then check to see whether its buddy is available. This can be done in constant time. If so, we remove the buddy from its available space list, and merge them together into a single free block of twice the size. This process is repeated until we find that the buddy is allocated.

Fig. 127 shows the deallocation of the block of size 1 at address 21. It is merged with its buddy of size 1 at address 20, thus forming a block of size 2 at 20. This is then merged with its size-2 buddy at 22, forming a block of size 4 at 20. Finally, this is merged with its size-4 buddy at 16, forming a block of size 8 at 16. This block's buddy (the block of size 8 starting at 0) is not available, so the merging process stops. We insert this final block into the appropriate level of the available space list.

**Fibonacci Buddy System:** An interesting variant of the buddy system is designed to reduce fragmentation by using a hierarchy that is "denser" with respect to the sizes available. The *Fibonacci Buddy System* uses Fibonacci numbers, rather than powers of 2.

Recall that $F(0) = 0$, $F(1) = 1$, and generally $F(i) = F(i-1) + F(i-2)$. The first few Fibonacci numbers are: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$. In this system `avail[k]` stores available

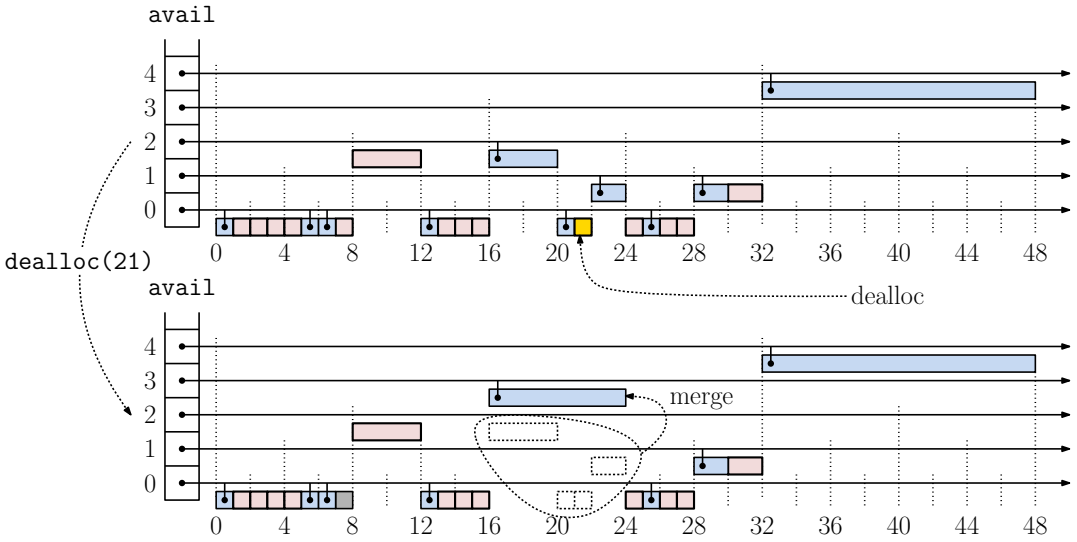



Fig. 127: Example of deallocating a block in the buddy system.

blocks of size $F(k)$. Each allocation request (after adding +1 for the header) is rounded up to the next larger Fibonacci number, say $F(k)$. If no block of size $F(k)$ is available, we find next larger available size, say $F(j)$. We then repeatedly split $F(j)$ using Fibonacci numbers to obtain a block of the desired size. For example, if $F(k) = F(3) = 2$ is the desired size and $F(j) = F(9) = 34$ is the smallest available, we split the $F(j)$ block up as:

$$F(9) = 34 = F(7)+F(8) = F(5)+F(6)+F(8) = F(3)+F(4)+F(6)+F(8) = 2+3+8+21.$$

We remove the block of size 34 from `avail[9]`, and split it up as described above. We return the block of size 2, and insert the blocks of sizes 3, 8, and 21 into `avail[4]`, `avail[6]`, `avail[8]`, respectively.

## Lecture 20: Bloom Filters

**Filtering:** In previous lectures, we have discussed randomized algorithms, that is, algorithms that use a random number generator to compute results. In those data structures (skip lists and treaps), the random number generator affected the running time, but the answers were always correct. Today, we will study a data structure in which the random choices may actually affect the correctness of the answer. We can make the probability of an erroneous answer as small as we like (with a modest increase in the running time).

The data structure we will consider is used in an application called *filtering*. Abstractly, imagine that we are given a large set $X$ of $n$ items drawn from a *universe* $U$. We want to answer set-membership queries of the form, "Is a given element $x$ in $X$?". There are many applications of this data structure. Here are a couple:

- The head of IT at your new company tells you that too many users are picking passwords that are easy to crack (like "password123"). You are given a list of one million easily breakable passwords. When a user creates a new password, you want to check whether it is in this list, and if so, tell them to pick again.

- Search engines (Google Chrome or Microsoft Bing) maintain a large list of URL's of malicious sites used in cyber-attacks. You are given a large list of URL's of known malicious sites, and you need to check each connection request against this list.

Of course, this is just a special case of the dictionary problem. (We will allow insertions into $X$, but no deletions. Also, there are no associated values, just keys. We just want a "yes/no" answer.)

We know many efficient dictionary data structures. For example, hashing takes $O(n)$ words of space and $O(1)$ (expected) query time. Let us imagine, however, that $n$ is very large, and we would like something much more space-efficient. Our goal will be a data structure that uses only around $O(n)$ *bits*, rather than $O(n)$ *words*.

**False Positive/Negative:** A *false positive* is when you erroneously report that $x \in X$, even though it isn't. A *false negative* occurs when you report that $x \notin X$, even though it is. False negatives are potentially very costly (allowing a user to use a weak password or allowing a connection to a malicious site).

We will present a new randomized data structure for this problem, called a *Bloom filter*. This data structure is very fast, it never suffers from false negatives, but it may (with low probability) suffer a false positive. That is, if $x \in X$, we are guaranteed to say "yes." If $x \notin X$, we will almost always say "no," but may occasionally say "yes" by mistake.

**Bloom Filter:** This data structure was invented in 1970 by Burton Howard Bloom. Recall that we wish to store a set $X \subseteq U$, where $|X| = n$. Our approach will be to use a collection hash functions, but unlike standard hashing, when collisions occur, we do not bother to resolve them. Formally, given a pair of positive integers $m$ and $k$ (to be determined later) a *Bloom filter* consists of a bit array `B[0..m-1]` together with $k$ hash functions $h_1, \ldots, h_k$. Each hash function maps an element of $U$ to an index $\{0, 1, \ldots, m-1\}$ into the array $B$. We'll denote the value of the $i$th hash function on argument $x$ as $h(i, x)$.

The algorithm is really amazingly simple. Initially, all the bits of `B` are set to 0. To insert an element, we apply each of our $k$ hash functions, and set all of these bits to 1. When we wish to find an element, we repeat the hashing process. We know that if it was inserted, then each of the $k$ hash entries must be 1. So, if even one of these has 0, we know (for sure) that $x$ is not in the set. See the code block below and the examples in Fig. 128.

Bloom Filter Operations

```
void insert(Key x) {                    // insert x into the Bloom filter
    for (i = 0; i < k; i++) {           // hash it k times
        B[h(i,x)] = 1                   // ...and set each bit to 1
    }
}

boolean find(Key x) {                   // find x in the the Bloom filter
    for (i = 0; i < k; i++) {           // hash it k times
        if (B[h(i,x)] == 0) return false // failed even once? - not found
    }
    return true                         // succeeded all k times
}
```

A purely practical point to make is that the $k$ hash functions do not depend on each other. So, if you are running on a typical modern machine with multiple cores, they can be computed

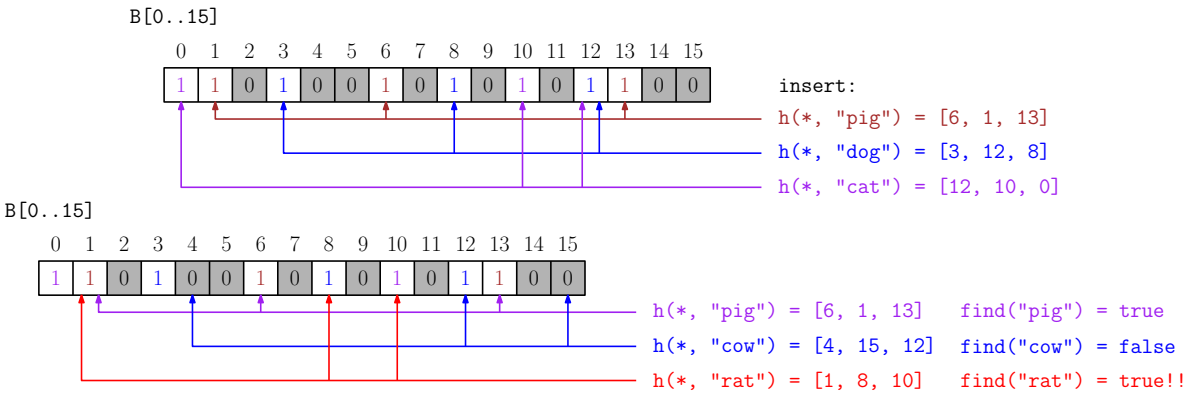



Fig. 128: Insertion and finding in a Bloom filter with $m = 16$ and $k = 3$.

in parallel. (This is not true for almost all of the data structures we have seen so far. Each step follows in sequence from the previous.)

**Partial Correctness:** Consider what happens during `find(x)`. If $x$ was already inserted, we set all $k$ of its hashed bits to 1, and when we search again, they will still be 1, so we are guaranteed to report that $x$ is present. (No false negatives.)

On the other hand, if all $k$ hashed entries contain a 1, it might indeed be the case that $x$ was inserted, and this is why all these bits are set. But, it could also have happened by coincidence. When other keys were inserted, they happened to hit all $k$ of these bits. As a result, we will incorrectly report that $x$ is present when it really is not. (We may have a false positive.)

**Controlling the False-Positive Rate:** It is not surprising that the probability that things go wrong will depend on the values of $k$ and $m$. As $m$ gets larger, it becomes less likely that two hash functions will accidentally collide, but this increases our space. As $k$ gets larger, it becomes less likely that two different keys will coincidentally hit the same bit positions, but we fill in more entries of `B` with 1's, and so coincidental matches are more likely. Need to find a good balance.

The analysis will require that we delve into a bit of probability theory. We will make use of a few basic facts:

- If $|z|$ is small $1 + z \approx e^z$.
- If $\ell$ independent events each have a probability $p$ of occurring, then the probability that they all occur is $p^\ell$.

Let $n = |X|$. Let's make the assumption that our hash functions are mutually independent and uniformly random. The probability of hitting any one of the $m$ bits of the $B$ vector is exactly $\frac{1}{m}$, and therefore the probability of missing it is $1 - \frac{1}{m}$. So, for any $x \in U$ and any $i$ and $j$, where $1 \leq i \leq k$ and $0 \leq j \leq m - 1$, we have

$$\Pr[h(i, x) \neq j] \;=\; 1 - \frac{1}{m}.$$

Suppose that after inserting all of our keys, we find that some entry of the bit vector is 0. This means every one of $kn$ attempts at hitting it failed. (There are $n$ keys inserted and each

applies $k$ different hash functions.) Clearly, this occurs with probability

$$\Pr[B[j] = 0] \;=\; \left(1 - \frac{1}{m}\right)^{kn}.$$

We may assume that $m$ is large, meaning that $\frac{1}{m}$ is small, an hence $1 - \frac{1}{m} \approx e^{-1/m}$. Therefore

$$\Pr[B[j] = 0] \;\approx\; (e^{-1/m})^{kn} \;=\; e^{-kn/m}.$$

Since we will use this value repeatedly, let's define $p = e^{-kn/m}$. (Later, we'll discover that the best choice is $p = 1/2$, so think of it intuitively as just a coin flip.)

Since the probability that any bit of $B$ is zero is $p$, it follows that the expected number of 0-bits is just $mp$. We will employ (without proof) an important fact from probability theory, which states that when dealing with a large number of independent trials, the actual number of times an event occurs is very close to its expected value. (This can be formally quantified using something called the *Chernoff bounds*. Since this is not a theory course, we'll just keep things simple, and assume the "ideal case" that the number of 0-bits in $B$ is exactly $mp$.)

So, what is the probability of a false positive when we search for a key $x$? In order for a false positive to happen, it must be the case that all of the bits that $x$ hashes to, that is, $h(1, x), \ldots, h(k, x)$, have already all been set to 1 by the other keys. Since we have $B[j] = 0$ with probability $p$, we have $B[j] = 1$ with probability $1 - p$. In order to get a false positive, this unfortunate event must reoccur $k$ times in a row. Each of these is an independent event. Therefore, the probability of this happening is the $k$-fold product of $(1 - p)$. Letting Pr[FP] denote the probability of a false positive, we have

$$\Pr[\text{FP}] \;=\; (1 - p)^k.$$

Let's assume that $n$ and $m$ are fixed (based on the amount of storage we can devote, given the size of our set). The question is what is the best choice for $k$ to minimize the false-positive rate. We claim that this will happen when $p = 1/2$. To show this, let's take the logarithm of this quantity. (Remember the that log function is monotonically increasing, so minimizing the log is equivalent to minimizing the original.)

$$\ln(\Pr[\text{FP}]) \;=\; \ln(1 - p)^k \;=\; k \ln(1 - p).$$

Given our definition $p$, we can equivalently express $\ln p = -k(n/m)$, or in other words, $k = -(m/n) \ln p$. Plugging this in, we find that the log of the false-positive probability is

$$\ln(\Pr[\text{FP}]) \;=\; -\frac{m}{n} \ln p \cdot \ln(1 - p).$$

Since we assume that $n$ and $m$ are fixed, we just need to minimize the quantity $-\ln p \cdot \ln(1 - p)$. Observe that this quantity is symmetric about $p = 1/2$. It stands to reason that the minimum value will be achieved at the symmetric point where $p = (1 - p) = 1/2$. (This is not immediately obvious. To be rigorous, you should take the derivative with respect to $p$ and set it to zero. Alternatively, just enter the function into any online graphing software, and you'll see right away that it is minimized at $p = 1/2$.)

By setting $p = 1/2$ and solving, we conclude that (for fixed values of $n$ and $m$) the optimum number of hash functions is $k = (\ln 2)(m/n)$. This gives a false positive rate of

$$\Pr[\text{FP}] \;=\; (1 - p)^k \;=\; \left(\frac{1}{2}\right)^{(\ln 2)(m/n)} \;\approx\; (0.61850)^{m/n}. \tag{1}$$

**Adjusting the Bit-Vector Size:** Rather than figuring out the false positive rate, suppose that the data structure designer tells us that they can tolerate a false-positive probability of $\delta > 0$. Solving Eq. (1) for $m$, we have the following.

$$m = \left\lceil \frac{\lg(1/\delta)}{\ln 2} n \right\rceil = O(n \log(1/\delta)).$$

Setting the table size to this value guarantees the desired false-positive rate. So, for example, if we wanted to achieve a 1% false-positive rate for a set of size $n$, we could do this with a Bloom filter involving $10n$ bits and 7 hash functions. (Assuming that each key is a 32-bit word, this is much smaller than the space needed to store the individual keys explicitly.)

# Supplemental Lectures

# Supplemental Lecture 1: Mathematical Preliminaries

**Mathematics:** In this lecture we will present a brief overview of the mathematical tools that will be needed to reason about the data structures and algorithms we will be working with. A good understanding of mathematics helps greatly in the ability to design good data structures, since through mathematics it is possible to get a clearer understanding of the nature of the data structures, and a general feeling for their efficiency in time and space. Last time we gave a brief introduction to asymptotic (big-"Oh" notation), and later this semester we will see how to apply that. Today we consider a few other preliminary notions: summations and proofs by induction.

**Summations:** Summations are important in the analysis of programs that operate iteratively. For example, in the following code fragment

```
for (i = 0; i < n; i++) { ... }
```

Where the loop body (the "...") takes $f(i)$ time to run the total running time is given by the summation

$$T(n) \;=\; \sum_{i=0}^{n-1} f(i).$$

Observe that nested loops naturally lead to nested sums. Solving summations breaks down into two basic steps. First simplify the summation as much as possible by removing constant terms (note that a constant here means anything that is independent of the loop variable, $i$) and separating individual terms into separate summations. Then each of the remaining simplified sums can be solved. Some important sums to know are

$$\begin{aligned}
\sum_{i=1}^{n} 1 &= n && \text{(The constant series)}\\
\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} && \text{(The arithmetic series)}\\
\sum_{i=1}^{n} \frac{1}{i} &= \ln n + O(1) && \text{(The harmonic series)}\\
\sum_{i=0}^{n} c^i &= \frac{c^{n+1}-1}{c-1} \qquad c \neq 1 && \text{(The geometric series)}
\end{aligned}$$

Note that complex sums can often be broken down into simpler terms, which can then be solved. For example

$$\begin{aligned}
T(n) &= \sum_{i=n}^{2n-1} (3+4i) \;=\; \sum_{i=0}^{2n-1} (3+4i) - \sum_{i=0}^{n-1} (3+4i)\\
&= \left(3\sum_{i=0}^{2n-1} 1 + 4\sum_{i=0}^{2n-1} i\right) - \left(3\sum_{i=0}^{n-1} 1 + 4\sum_{i=0}^{n-1} i\right)\\
&= \left(3(2n) + 4\frac{2n(2n-1)}{2}\right) - \left(3(n) + 4\frac{n(n-1)}{2}\right) \;=\; n + 6n^2.
\end{aligned}$$

The last summation is probably the most important one for data structures. For example, suppose you want to know how many nodes are in a complete 3-ary tree of height $h$. The *height* of a tree is the maximum number of edges from the root to a leaf.) One way to break this computation down is to look at the tree level by level. At the top level (level 0) there is 1 node, at level 1 there are 3 nodes, at level 2, 9 nodes, and in general at level $i$ there $3^i$ nodes. To find the total number of nodes we sum over all levels, 0 through $h$. Plugging into the above equation with $h = n$ we have:

$$\sum_{i=0}^{h} 3^i = \frac{3^{h+1} - 1}{2} \in O(3^h).$$

Conversely, if someone told you that he had a 3-ary tree with $n$ nodes, you could determine the height by inverting this. Since $n = (3^{(h+1)} - 1)/2$ then we have

$$3^{(h+1)} = (2n + 1)$$

implying that

$$h = (\log_3(2n + 1)) - 1 \in O(\log n).$$

Another important fact to keep in mind about summations is that they can be approximated using integrals.

$$\sum_{i=a}^{b} f(i) \approx \int_{x=a}^{b} f(x)dx.$$

Given an obscure summation, it is often possible to find it in a book on integrals, and use the formula to approximate the sum.

**Recurrences:** A second mathematical construct that arises when studying recursive programs (as are many described in this class) is that of a recurrence. A recurrence is a mathematical formula that is defined recursively. For example, let's go back to our example of a 3-ary tree of height $h$. There is another way to describe the number of nodes in a complete 3-ary tree. If $h = 0$ then the tree consists of a single node. Otherwise that the tree consists of a root node and 3 copies of a 3-ary tree of height $h-1$. This suggests the following recurrence which defines the number of nodes $N(h)$ in a 3-ary tree of height $h$:

$$\begin{aligned} N(0) &= 1 \\ N(h) &= 3N(h-1) + 1 \qquad \text{if } h \geq 1. \end{aligned}$$

Although the definition appears circular, it is well grounded since we eventually reduce to $N(0)$.

$$\begin{aligned} N(1) &= 3N(0) + 1 = 3 \cdot 1 + 1 = 4 \\ N(2) &= 3N(1) + 1 = 3 \cdot 4 + 1 = 13 \\ N(3) &= 3N(2) + 1 = 3 \cdot 13 + 1 = 40, \end{aligned}$$

and so on.

There are two common methods for solving recurrences. One (which works well for simple regular recurrences) is to repeatedly expand the recurrence definition, eventually reducing

it to a summation, and the other is to just guess an answer and use induction. Here is an example of the former technique.

$$\begin{aligned} N(h) &= 3N(h-1)+1 \\ &= 3(3N(h-2)+1)+1 \;=\; 9N(h-2)+3+1 \\ &= 9(3N(h-3)+1)+3+1 \;=\; 27N(h-3)+9+3+1 \\ &\vdots \\ &= 3^k N(h-k) + (3^{k-1} + \ldots + 9 + 3 + 1) \end{aligned}$$

When does this all end? We know that $N(0) = 1$, so let's set $k = h$ implying that

$$N(h) = 3^h N(0) + (3^{h-1} + \ldots + 3 + 1) = 3^h + 3^{h-1} + \ldots + 3 + 1 = \sum_{i=0}^{h} 3^i.$$

This is the same thing we saw before, just derived in a different way.

**Proofs by Induction:** The last mathematical technique of importance is that of proofs by induction. Induction proofs are critical to all aspects of computer science and data structures, not just efficiency proofs. In particular, virtually all correctness arguments are based on induction. From courses on discrete mathematics you have probably learned about the standard approach to induction. You have some theorem that you want to prove that is of the form, "For all integers $n \geq 1$, blah, blah, blah", where the statement of the theorem involves $n$ in some way. The idea is to prove the theorem for some basis set of $n$-values (e.g. $n = 1$ in this case), and then show that if the theorem holds when you plug in a specific value $n - 1$ into the theorem then it holds when you plug in $n$ itself. (You may be more familiar with going from $n$ to $n + 1$ but obviously the two are equivalent.)

In data structures, and especially when dealing with trees, this type of induction is not particularly helpful. Instead a slight variant called *strong induction* seems to be more relevant. The idea is to assume that if the theorem holds for all values of $n$ that are strictly less than $n$ then it is true for $n$. As the semester goes on we will see examples of strong induction proofs.

Let's go back to our previous example problem. Suppose we want to prove the following theorem.

**Theorem:** Let $T$ be a complete 3-ary tree with $n \geq 1$ nodes. Let $H(n)$ denote the height of this tree. Then

$$H(n) = (\log_3(2n+1)) - 1.$$

**Basis Case:** (Take the smallest legal value of $n$, $n = 1$ in this case.) A tree with a single node has height 0, so $H(1) = 0$. Plugging $n = 1$ into the formula gives $(\log_3(2 \cdot 1 + 1)) - 1$ which is equal to $(\log_3 3) - 1$ or 0, as desired.

**Induction Step:** We want to prove the theorem for the specific value $n > 1$. Note that we cannot apply standard induction here, because there is no complete 3-ary tree with 2 nodes in it (the next larger one has 4 nodes).

We will assume the induction hypothesis, that for all smaller $n'$, $1 \leq n' < n$, $H(n')$ is given by the formula above. (This is sometimes called *strong induction*, and it is good to learn since most induction proofs in data structures work this way.)

Let's consider a complete 3-ary tree with $n > 1$ nodes. Since $n > 1$, it must consist of a root node plus 3 identical subtrees, each being a complete 3-ary tree of $n' < n$ nodes. How many nodes are in these subtrees? Since they are identical, if we exclude the root node, each subtree has one third of the remaining number nodes, so $n' = (n-1)/3$. Since $n' < n$ we can apply the induction hypothesis. This tells us that

$$\begin{aligned} H(n') &= (\log_3(2n'+1)) - 1 = (\log_3(2(n-1)/3+1)) - 1 \\ &= (\log_3(2(n-1)+3)/3) - 1 = (\log_3(2n+1)/3) - 1 \\ &= \log_3(2n+1) - \log_3 3 - 1 = \log_3(2n+1) - 2. \end{aligned}$$

Note that the height of the entire tree is one more than the heights of the subtrees so $H(n) = H(n') + 1$. Thus we have:

$$H(n) = \log_3(2n+1) - 2 + 1 = \log_3(2n+1) - 1,$$

as desired.

This may seem like an awfully long-winded way of proving such a simple fact. But induction is a very powerful technique for proving many more complex facts that arise in data structure analyses.

You need to be careful when attempting proofs by induction that involve $O(n)$ notation. Here is an example of a common error.

**(False) Theorem:** For $n \geq 1$, let $T(n)$ be given by the following summation

$$T(n) = \sum_{i=0}^{n} i,$$

then $T(n) \in O(n)$. (We know from the formula for the linear series above that $T(n) = n(n+1)/2 \in O(n^2)$. So this must be false. Can you spot the error in the following "proof"?)

**Basis Case:** For $n = 1$ we have $T(1) = 1$, and 1 is $O(1)$.

**Induction Step:** We want to prove the theorem for the specific value $n > 1$. Suppose that for any $n' < n$, $T(n') \in O(n')$. Now, for the case $n$, we have (by definition)

$$T(n) = \sum_{i=0}^{n} i = \left(\sum_{i=0}^{n-1} i\right) + n = T(n-1) + n.$$

Now, since $n-1 < n$, we can apply the induction hypothesis, giving $T(n-1) \in O(n-1)$. Plugging this back in we have

$$T(n) \in O(n-1) + n.$$

But $(n-1) + n \leq 2n - 1 \in O(n)$, so we have $T(n) \in O(n)$.

What is the error? Recall asymptotic notation applies to arbitrarily large $n$ (for $n$ in the limit). However induction proofs by their very nature only apply to specific values of $n$. The proper way to prove this by induction would be to come up with a concrete expression, which does not involve $O$-notation. For example, try to prove that for all $n \geq 1$, $T(n) \leq 50n$. If you attempt to prove this by induction (try it!) you will see that it fails.

## Supplemental Lecture 2: Quake Heaps

**Priority Queues and Heaps:** A *priority queue* is an abstract data structure storing key-value pairs. The basic operations involve inserting a new key-value pair (where the key represents the *priority*) and extracting the entry with the smallest priority value. These operations are called *insert* and *extract-min*, respectively. Priority queues are often implemented by a tree data structure called a *heap*, where keys decrease monotonically along any path to the root.

In addition to insert and extract-min, there are a number of operations that we may like to perform on priority queues, and heaps are often augmented with different capabilities (such as the capability to merge two heaps). As with many data structures, the objective is to perform these operations in time $O(\log n)$, where $n$ is the number of keys in the data structure.

**Decrease-Key Operation:** One fundamental operation, called *decrease key*, involves decreasing the key value of a given entry in the heap by a given amount. The decrease-key operation arises when heaps are used in algorithms such as Dijkstra's algorithm for shortest paths in graphs and Prim's algorithm for computing minimum-cost spanning trees. In such algorithms, the decrease-key operation is performed more frequently than the extract-min operation, and so it is critical that it be performed efficiently. (In Dijkstra's algorithm, decrease-key is performed once for every edge of the graph whereas extract-min is performed once for every vertex. A graph with $n$ vertices may have $O(n^2)$ edges, so there may be many more decrease-key operations compared to extract-mins.)

It is not hard to implement the operations insert, extract-min, and decrease-key using a standard binary heap (the same data structure used in heapsort) so that they all run in $O(\log n)$ time each. The question that an implementer of Dijkstra's algorithm would like to know is whether it is possible to implement decrease-key more efficiently, ideally to run in just $O(1)$ time. It is not known how to do this in the worst case, but it is possible to perform decrease-key operations in $O(1)$ *amortized time*, which is sufficient when the data structure is used within an algorithm.

The most famous heap structure supporting decrease-key in $O(1)$ amortized time is the *Fibonacci heap*, which was discovered by Michael Fredman and Robert Tarjan in 1984. Unfortunately, Fibonacci heaps are rather complicated structures to describe and analyze. People have studied simpler alternatives. In this lecture, we will present such a structure called a *quake heap*. It was discovered by Timothy Chan around 2013. Our description of the data structure is a bit different and more detailed that Chan's original description (which is only 5 pages long!), but the ideas are essentially the same.

**But not "increase-key"?** Take note that heaps are asymmetric with respect to key orders. While decrease-key can be implemented to run in $O(1)$ amortized time (assuming a min-heap), there is no min-heap (of which I am aware) that supports the complementary operation of *increase-key* is $O(1)$ amortized time.

**Quake-Heap Specifications:** Before listing the Quake Heap operations, we should first discuss how to specify the element on which decrease-key and delete are to be applied. Unlike dictionaries, heaps do not support fast searching. Thus, if we insert a key $x$, there is no efficient way to later find out where it is in the heap. For this reason, the insert function returns a reference to the node containing $x$, called a *locator*. When we later want to refer to a key, we do so through its locator.

`Locator r = insert(Key x)`: Insert the key $x$ into the heap, and return a reference indicating its location in the heap.

`Key x = extract-min()`: Remove the item with the minimum key $x$ from the heap, returning its key value.

`void decrease-key(Locator r, Key y)`: Decrease the key of the item referenced by $r$ to $y$. (If $y$ is larger than the current key an exception is thrown.)

These are the most basic operations. Note that, unlike a binary search tree, duplicate keys are allowed, and the extract-min operation may select among ties arbitrarily. We shall see that the quake heap implements insert and decrease-key in $O(1)$ worst-case time and extract-min is $O(\log n)$ amortized time. Extract-min can take $O(n)$ time in the worst case.

Why is it called "quake heap"? The reason is that the data structure allows itself to slowly go out of balance. When it determines that it is very badly out of balanced, it massively reorganizes itself by "flattening" the entire structure, like a city hit by an earthquake.

**Quake Heap Structure:** The quake heap is represented as a collection of binary trees, where each node stores a key value. The nodes of these trees are organized into *levels*. All the leaves reside on level 0, and each key in the heap is stored in one leaf of some tree (the shaded blue square nodes in Fig. 129). Each internal node has a left child and an optional right child. Its key value is that of its left child. If the right child exists, its key value is greater than or equal to the left child. Thus, the root of each tree holds the smallest key value in the tree, which is that of the leftmost leaf.

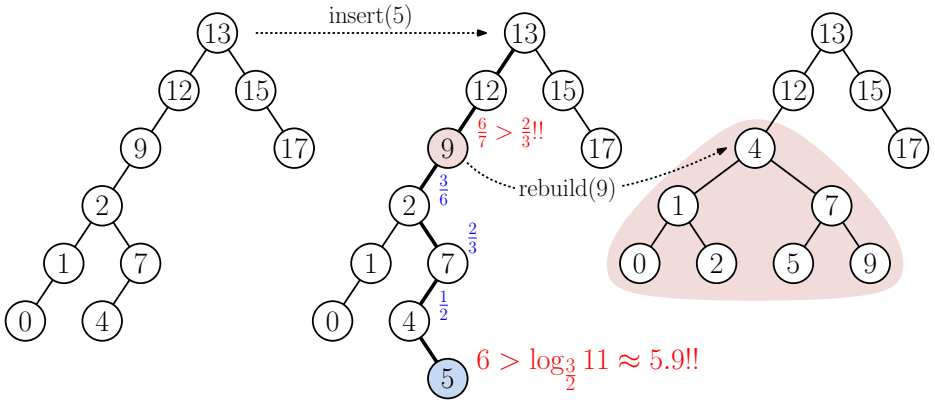



Fig. 129: An example of a quake heap consisting of four trees, storing a total of 13 keys. (As usual, values are omitted.) On the left we show arrays of node counts and roots lists.

Each node stores its key, child pointers, parent pointer, and its level. We maintain two additional arrays organized by level. First, `roots` is an array of linked lists storing the roots of each level. (In Fig. 129 these are shown as singly linked lists in blue, but it would be better to use a doubly linked list for fast insertion and deletion). Second, `nodeCt` is an array of integers storing the total node count (not to be confused with the root count) at each level.

**Linking and Cutting:** Here are a few useful utility operations, which are applied to manipulate quake heaps. After each operation, the root lists and node counts need to be updated. See the following code fragments for details.

`void make-root(Node u)`: Converts node $u$ into a root by setting its parent link to `null` and adding it to the list of roots.

`Node trivial-tree(Key x)`: Creates a trivial single-node tree storing the key `x`.

`Node w = link(Node u, Node v)`: Link two root nodes `u` and `v` at the same level by joining them under a common root one level higher. The root with the smaller key goes on the left side of the new parent (see Fig. 130).

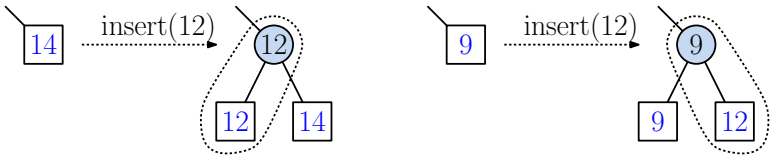



Fig. 130: The operations (a) link and (b) cut.

`void cut(Node w)`: If `w`'s right child is `null`, then do nothing. Otherwise, cut the link between `w` and right child, causing this child to become the root of a new tree one level lower. Node `w` now has just a single left child (see Fig. 130).

Trivial Tree and Link Utilities

```
void make-root(Node u) {                    // make u a root node
    u.parent = null                         // null out parent link
    add u to roots[u.level]                 // add it to the list of roots
}

Node trivial-tree(Key x) {                  // create a trivial single-node tree
    Node u = new leaf node with key=x at level=0
    nodeCt[0] += 1                          // increment node count
    make-root(u)                            // make it a root
    return u
}

Node link(Node u, Node v) {                 // link u and v into new tree
    int lev = u.level + 1                   // new node's level
    if (u.key <= v.key)                     // u's key is smaller?
        Node w = new Node(u.key, lev, u, v) // new root with u's key
    else
        Node w = new Node(v.key, lev, v, u) // new root with v's key
    nodeCt[lev] += 1                        // increment node count
    u.parent = v.parent = w                 // w is the new parent
    return w
}

void cut(Node w) {                          // cut off w's right child
    Node v = w.right;
    if (v != null) {
        w.right = null;                     // cut off v
        makeRoot(v);                        // ... and make it a root
    }
}
```

Observe that if properly implemented, all three operations can be performed in $O(1)$ time

**Quake Heap Updates:** Using the above utilities, we will show how to perform the various quake-heap operations. The overall design philosophy is to be as "lazy" as possible with all the operations except for `extract-min`, which is responsible for maintaining proper structure.

`Locator r = insert(Key x):` Create a new single-node tree `x` and return a reference to it. (In general, there should both a key and value, but we omit values here.)

`void decrease-key(Locator r, Key newKey):` Let `oldKey` denote the key value at the node indicated by leaf node `r`. Starting at this leaf node, walk up the path of nodes containing `oldKey`, updating their key values as we go (see Fig. 131). We stop when we first encounter a node having a different key value. Generally, we may be out of heap order with respect to this node, and so we apply cut at this point (see Fig. 131, right)). As described this operation takes time proportional to the height of the tree, but with some cleverness, it is possible to implement it in $O(1)$ time. We will discuss this below.

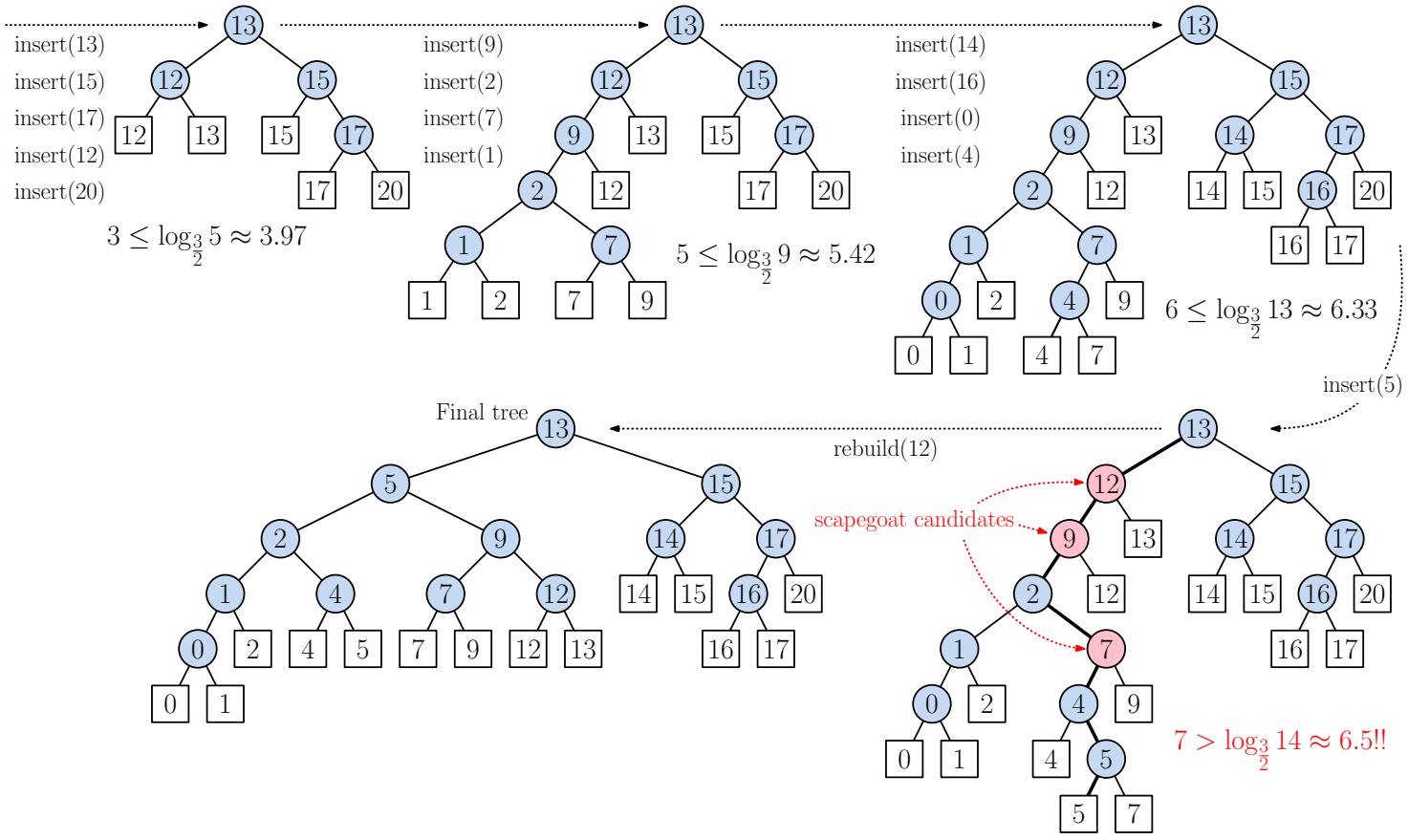



Fig. 131: The operation `decrease-key(r,2)`, which decreases key 7 to 2.

`Key x = extract-min():` This operation is the most complex because it is responsible keeping the data structure in good balance. It involves the following steps:

**Find minimum root:** First, we visit all the roots of all the trees (traversing the `roots` lists) and find the one with the smallest key. (Among 8, 4, and 7 in Fig. 132(a), this is 4.) Let this root node be `u` and let `x` be the associated (minimum) key.

**Delete left path:** Next, the function `delete-left-path(u)` traverse the path from `u` down to its leftmost leaf, which contains `x`, and we remove all the nodes along this

```
Locator insert(Key x) {                          // insert key x in heap
    Node u = trivial-tree(x)                     // create a one-node tree storing x
    return new Locator(u)                        // (the locator stores a link to u)
}

void decrease-key(Locator r, Key newKey) {       // decrease key to newKey
    Node u = r.get()                             // leaf node to be changed
    Node uChild = null                           // u's child on path
    do {
        u.key = newKey                           // update key value
        uChild = u; u = u.parent                 // move up a level
    } while (u != null && uChild == u.left)      // until end of left path
    if (u != null) cut(u)                        // cut us off from parent
}
```

path (see Fig. 132(b)). As a result, we may obtain a number of new trees. (In Fig. 132(c), we have new trees rooted at 5, 20, and 9.)

**Merge trees:** We do not want to have too many trees, so we next apply a tree-merging step. The function `merge-trees()` works bottom-up. Whenever we see two trees of the same level, we link them, thus creating a tree at the next higher level. We repeat this until each level has either zero or one trees. (In Fig. 132(d) we have redrawn the trees for clarity. We merge 8 and 20 on level 1, which produces a node 8 on level 2. We merge 5 with 8 on level 2, resulting in node 5 on level 3 (see Fig. 132(e)).

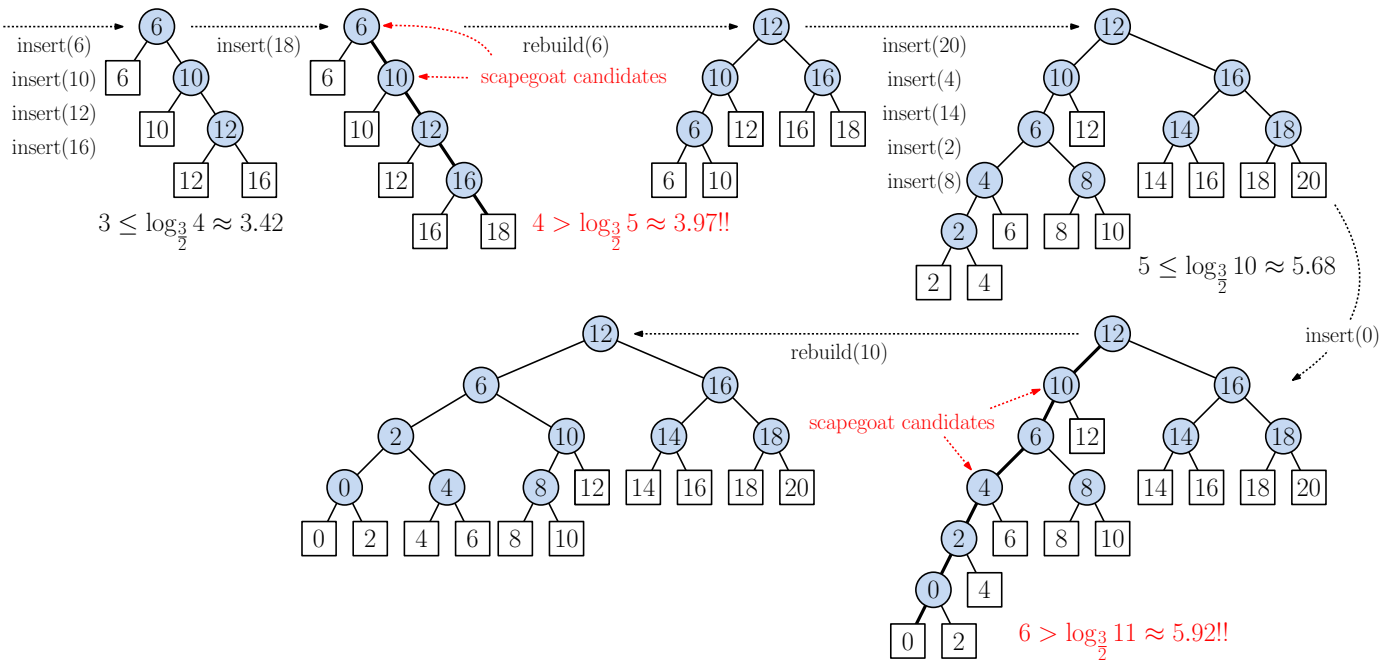



Fig. 132: The operation `extract-min()`.

**Quake:** When deleting nodes or performing cuts, we tend to produce more and more nodes that have just a single child. To remedy this, we the `quake()` function searches for the smallest level (if any) such that `nodeCt[lev+1]` $> \frac{3}{4} \cdot$ `nodeCt[lev]`. If we

find such a level, we remove all nodes at levels `lev` + 1 and higher.[10] (In Fig. 132(e) we see that `nodeCt[2]` $= 4 > \frac{3}{4} \cdot$ `nodeCt[1]` $= \frac{3}{4} \cdot 5 = 3.75$ and in Fig. 132(f) we show the result removing all nodes at level 2 and higher. This generates 5 new trees, one for each node on level 1.)

**Result:** Finally, return the key from the first step as the final result.

The entire extract-min process is illustrated in the code block below. The member variable `nLevels` stores the number of levels in the data structure. Ideally, the number of levels will expand as more items are added to the heap (e.g., through the use of a Java ArrayList), but here we just assume it is fixed.

Extract-Min and Utilities

```
Key extract-min() {                                 // extract the minimum key
    Node u = find-root-with-smallest-key()          // find the min root (exercise)
    Key result = u.key                              // final return result
    delete-left-path(u)                             // delete entire left path
    remove u from roots[u.level]                    // remove u as a root
    merge-trees()                                   // merge tree pairs
    quake()                                         // perform the quake operation
    return result
}

void delete-left-path(Node u) {                     // delete left path to leaf
    while (u != null) {                             // repeat all the way down
        cut(u)                                      // cut off u's right child
        nodeCt[u.level] -= 1                        // one less node on this level
        u = u.left                                  // go to the left child
    }
}

void merge-trees() {                                // merge trees bottom-up in pairs
    for (int lev = 0; lev < nLevels-1; lev++) {     // process levels bottom-up
        while (roots[lev] size is >= 2) {           // at least two trees?
            Node u = remove any from roots[lev]     // remove any two
            Node v = remove any from roots[lev]
            Node w = link(u, v)                     // ... and merge them
            make-root(w)                            // ... and make this a root
        }
    }
}

void quake() {                                      // flatten if needed
    for (lev = 0; lev < nLevels-1; lev++) {         // process levels bottom-up
        if (nodeCt[lev+1] > 0.75 * nodeCt[lev])     // too many?
            clear-all-above-level(lev)              // clear all nodes above level lev
    }
}
```

There are two additional utilities. The first is called `find-root-with-smallest-key()`. It searches all the roots for the one with smallest key. (We have omitted this.) The second is called `clear-all-above-level(top)`. It removes all nodes strictly above level `top`,

[10] By the way, there is nothing magic about the number $\frac{3}{4}$. The analysis works for any constant $\alpha$, where $\frac{1}{2} < \alpha < 1$.

and makes all the nodes of this level into roots. This visits all the roots at levels `top+1` and higher and "un-roots" by making its children into roots.

Clear Above Level Utility

```
void clear-all-above-level(int top) {          // clear all nodes above top
    for (int lev = nLevels - 1; lev > top; lev--) { // process all levels
        for-each (Node u in roots[lev]) {      // apply to each tree
            remove u from roots[lev]
            unroot(u)
        }
        nodeCt[lev] = 0                        // zero out the node count
    }
}

void unroot(Node u) {                          // remove u as a root
    if (u.left != null)
        make-root(u.left)                      // make u's children roots
    if (u.right != null)
        make-root(u.right)
    Node ll = u.leftmostLeaf                   // u's leftmost leaf
    ll.topLeftNode = u.left                    // update top-left node to u
}
```

This is everything that you need to know to implement the data structure (but if you want the best performance, see the next section on how to implement decrease-key in $O(1)$ time).

**Faster decrease-key:** (Optional) As we described it, `decrease-key` requires time proportional to the highest level of the node whose key is being decreased. We will show below that each tree is of height $O(\log n)$, so this is already not bad. But the main reason for presenting the Quake Heap was to show that `decrease-key` can be performed in $O(1)$ time! So can we speed it up? Yes, this can be achieved with two simple modifications to our implementation:

**Left-leaf pointers:** Since each key appears in multiple nodes, it takes time to update these values. Instead, we store the key only once, in the associated leaf node. Every internal node along the chain of left-child links that leads to this leaf would normally store this key. Instead, each stores a pointer to the leaf (see Fig. 133(a)). As a result, we need only change the one occurrence of the key in the leaf node. This way, when we wish to change the key, we need only change it in the leaf node in $O(1)$ time.

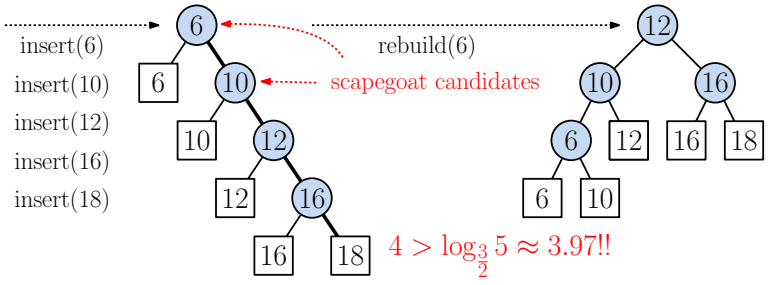



Fig. 133: Enhancements for faster `decrease-key()`.

**Highest left-side ancestor:** To avoid searching level-by-level from a leaf node up to the highest ancestor along the left-child chain, each leaf node stores a link to the highest node containing this key (see Fig. 133(b)).

It is an easy exercise to modify the `link` function to implement these changes in $O(1)$ time. (Note that `cut` does not need to be changed, since it only cuts right child links.) With these two modifications, we can implement `decrease-key` in $O(1)$ time as follows. First, given the leaf node whose key is to be changed, we first change the key in the leaf in $O(1)$ time. Next, we access the leaf's thread, to find its highest ancestor in $O(1)$ time, and we then apply `cut()` on its parent (assuming it has one), which takes $O(1)$ time.

**Amortized Analysis: (Optional)** For the remainder of the lecture, let's analyze the data structure's running time. On looking at Fig. 132(f) you might protest that this data structure cannot possibly be efficient. The `quake` function could potentially flatten the entire structure all the way down to the leaf level (hence the name "Quake Heap"). We will prove this high degree of flattening is *extremely rare*, and the cost of extract-min, when amortized over a sequence of operations, is only $O(\log n)$.

To make this formal, consider any sequence of $m$ operations (insert, extract-min, decrease-key) starting from an empty heap, and let $n \leq m$ denote the total number of items inserted into the heap over all these operations. For $1 \leq i \leq m$, let $T_i$, denote *actual time* needed to perform the $i$th operation. We will ignore constant factors and write 1 for $O(1)$ and $\log n$ for $O(\log n)$.

For any sequence of $m$ operations, the *total time* is $T(m) = \sum_{i=1}^{m} T_i$. The *amortized time* is defined to be the average time over the entire sequence, that is, $A(m) = \frac{1}{m}T(m)$. Here is our main result:

**Theorem:** Given any sequence of $m$ operations on a Quake Heap involving at most $n$ entries, each insert and decrease-key operation takes $O(1)$ time in the worst-case and extract-min takes $O(\log n)$ amortized time.

**How high?** Before getting into the proof, we should first derive a bound on how high the trees of the heap might grow.

**Lemma:** Consider any binary tree, where there are $n$ leaves all at level 0, and for each level $i \geq 0$, the number of nodes at this level, denoted $n_i$, satisfies the property that $n_{i+1} \leq \alpha n_i$, for some constant $\alpha < 1$. Then the tree has height at most $\log_{1/\alpha} n = O(\log n)$.

We will leave the proof as an easy exercise, but the key idea is that if the number of nodes decreases by a constant factor at each level, we can't keep this up for more than a logarithmic number of levels.

The subtrees that remain after any quake operation satisfy the above lemma for $\alpha = \frac{3}{4}$, and therefore the maximum tree height is $\log_{4/3} n = (\lg n)/\lg(4/3) \approx 2.4 \lg n$. It is easy to verify that the other operations do not increase tree heights, so this bound applies to the entire structure.

**Review of Potential-Based Analyses:** Our proof employs a *potential-based analysis*. Each instance of the data structure will be associated with a nonnegative function $\Psi$, called its *potential*. Intuitively, low potential means the structure is well balanced and high values mean it is poorly structured. (We'll present the precise definition below.) For $1 \leq i \leq m$,

let $\Psi_i$ denote the value of the potential for the data structure after the $i$th operation, and let $\Psi_0$ denote the initial potential. Define the *change in potential* $\Delta_i = \Psi_i - \Psi_{i-1}$. The *amortized time* of the $i$th operation, denoted $A_i$, is the sum of the actual cost and the change in potential, that is $A_i = T_i + \Delta_i$.

Why are we doing this? Intuitively, some operations (insert and decrease-key) are cheap in the sense that the actual time $T_i$ is small, but they can make the data structure less balanced, resulting in an increase to the potential. On the other hand, some operations (extract-min in particular) may take much more time, so $T_i$ can be very large. However, these operations improve the data structure's balance, meaning that potential decreases by quite a bit. Thus, there is a trade-off between operations that are cheap (but sloppy) and operations that are expensive (but beneficial).

What do the individual amortized times $A_i$ have to do with the overall amortized time $A(m)$?. To see this, let's write out the sum of the amortized time:

$$\begin{aligned}
\sum_{i=1}^{m} A_i &= \sum_{i=1}^{m}(T_i + \Delta_i) = \sum_{i=1}^{m}(T_i + (\Psi_i - \Psi_{i-1})) = \sum_{i=1}^{m} T_i + \sum_{i=1}^{m}(\Psi_i - \Psi_{i-1}) \\
&= \sum_{i=1}^{m} T_i + (-\Psi_0 + \cancel{\Psi_1} - \cancel{\Psi_1} + \cancel{\Psi_2} - \cancel{\Psi_2} + \cdots - \cancel{\Psi_{m-1}} + \Psi_m) \\
&= \sum_{i=1}^{m} T_i + (\Psi_m - \Psi_0) = T(m) + (\Psi_m - \Psi_0).
\end{aligned}$$

Therefore, the sum of amortized times is equal to the total time plus the overall increase in potential. We will see that the potential of an empty structure is zero, so $\Psi_0 = 0$. Since $\Psi$ is nonnegative, we conclude that $\Psi_m - \Psi_0 \geq 0$, and therefore

$$A(m) = \frac{1}{m}T(m) \leq \frac{1}{m}\big(T(m) + (\Psi_m - \Psi_0)\big) = \frac{1}{m}\sum_{i=1}^{m} A_i.$$

So in summary, in order to bound the total amortized cost $A(m)$, it suffices to bound the individual amortized costs $A_i$. The above holds for any data structure. Next, we will show that for the Quake Heap, insert and decrease-key operations, $A_i = O(1)$, and for extract-min $A_i = O(\log n)$.

**Quake-Heap Amortized Analysis:** In order to bound the amortized cost of an operation, we need to define our potential function. Ideally, our tree would consist of just a single tree (a single root node), and all internal nodes have two children. We say that an internal is *bad* if it has only one child. Consider the tree at any given moment. Let $N$ denote the current number of nodes, $R$ the current number of root nodes, and $B$ the current number of bad nodes. We want to penalize heaps that have multiple roots and lots of bad nodes. Define our potential to be $\Psi = N + 2R + 4B$. (Our definition is a bit different from Chan's, but the analysis is essentially the same.)

Now, let's consider what happens when we perform an operation. Let $\Psi' = N' + 2R' + 4B'$ denote the new values just after completing this operation. We are interested in the actual work $T$ and the change in the potential. Define the change in the number of nodes to be $\Delta N = N' - N$, and define $\Delta R$, $\Delta B$, and $\Delta\Psi$ analogously.

`insert`: Insert takes $T = O(1)$ actual time (just create a node and add it to the leaf-level roots list). We get one new node and one new root, so $\Delta N = \Delta R = 1$, and $\Delta B = 0$. Thus, (ignoring constants) the amortized cost is $A = T + \Delta\Psi = 1 + (1 + 2 \cdot 1 + 4 \cdot 0) = 4$, which is $O(1)$.

`decrease-key`: Assuming fast decrease-key, this operation can be implemented in $T = O(1)$ actual time. The number of roots and number of bad nodes each increase by at most 1. The number of nodes does not change. Therefore, amortized cost is $A = T + \Delta\Psi \leq 1 + (0 + 2 \cdot 1 + 4 \cdot 1) = 8 = O(1)$.

`extract-min`: As might be expected, this is the most complex to analyze. Recall that the maximum level number is $O(\log n)$. Let's ignore the constant factor, and just call this "$\lg n$".

To complete this part of the analysis, we will show that each of its basic elements has an amortized cost of at most $O(\log n)$.

**Find-min-root and delete-left-path:** We will show that the amortized cost of these operations combined is $R + O(\log n)$. First, observe that finding the minimum key takes actual time proportional to the number of roots $R$. The delete-left-path operation takes actual time proportional to the maximum tree height, which we have shown to be $\lg n$ (ignoring constant factors). Thus, the total actual time is $T \leq R + O(\log n)$.

How does the potential change? In the process of performing cuts for delete-left-path, we have only decreased the number of nodes and number of bad nodes (both of which are beneficial, but we'll ignore them). We have also converted up to $\lg n$ nodes into new roots. Thus, the increase in potential is at most

$$\Delta\Psi \;=\; \Delta N + 2\Delta R + 4\Delta B \;\leq\; 0 + 2\lg n + 0 \;=\; O(\log n).$$

Therefore, the total amortized cost is $T + \Delta\Psi = R + O(\log n)$.

**Merge-trees:** The $R$ term in the above amortized cost can be very large, but merge-trees comes to our rescue. After running it, we have at most one root at each of the $\lg n$ levels, that is, $R' \leq \lg n$. Thus, just considering the root portion of the potential, the change is at most $\Delta R = R' - R \leq (\lg n) - R$. (When $R$ is very large, this represents a large decrease.) Let's apply this potential drop to pay for extra $R$ term in find-min and delete-left-path. We have a total amortized cost of

$$A \;=\; T + \Delta\Psi \;\leq\; (R + \lg n) + (\lg n) - R \;=\; 2\lg n \;=\; O(\log n).$$

Of course, we also need to account for the time to perform merge-trees, but we claim that its amortized time is zero. To see why, observe that each time we merge a pair of trees (which takes $O(1)$ time), we create one new node (which is also a root node), but we have also eliminated two root nodes. We never create single-child nodes. So, the contribution to the change in potential is

$$\Delta\Psi \;=\; \Delta N + 2\Delta R + 4\Delta B \;=\; 1 + 2(1 - 2) + 4 \cdot 0 \;=\; -1.$$

The amortized time for merge-trees is $T + \Delta\Psi = +1 - 1 = 0$.

**Quake:** We claim that the amortized time for this operation is also zero. Suppose that a quake occurs at level $j$, causing us to remove all the nodes at levels $j + 1$ and higher. Let $n_j$ denote the number of nodes at level $j$ (assuming that delete-left-path and merge-trees have already taken place). Let's define $n_{>j}$ to be the total number of nodes strictly

above level $j$. The actual work is proportional to the number of nodes removed, that is, $T \leq n_{>j}$. Since these nodes all are gone, the change in the number of nodes is $\Delta N = -n_{>j}$. Therefore, we have $T + \Delta N = n_{>j} - n_{>j} = 0$.

Each node at level $j$ becomes a new root, so the increase in the number of roots is at most $n_j$. (We have certainly lost roots at higher levels, but we can ignore these since they only make our potential change better.)

Most importantly, we have eliminated a lot of bad nodes. Let $b_j$ denote the number of bad nodes at level $j$. Every node at level $j+1$ could have potentially two children at level $j$, but each bad node has one less child, so we have $n_j \geq 2n_{j+1} - b_{j+1}$. (The number could be higher, because we may have roots at level $n_j$.) Equivalently, we have $b_{j+1} \geq 2n_{j+1} - n_j$. In order for quake to be triggered, we know that $n_{j+1} > \frac{3}{4}n_j$, and therefore $b_{j+1} > 2\left(\frac{3}{4}n_j\right) - n_j = \frac{1}{2}n_j$. All of these bad nodes are now gone, so we have $\Delta B \leq -b_{j+1} < -\frac{1}{2}n_j$. Recalling that $R$ increased by at most $n_j$, it follows that the net change in $2\Delta R + 4\Delta B$ is at most $2n_j - 4\left(\frac{1}{2}n_j\right) = 0$.

The amortized cost is $T + \Delta\Psi = (T + \Delta N) + (2\Delta R + 4\Delta B)$, but we have just shown that both of these values are at most zero. Therefore, the amortized cost of Quake is at most zero, completing the amortized analysis. Whew!

## Supplemental Lecture 3: Scapegoat Trees

**Scapegoat Trees:** We have previously studied the *splay tree*, a data structure that supports dictionary operations in $O(\log n)$ amortized time. Recall that this means that, over a series of operations, the average cost per operation is $O(\log n)$, even though the cost of any individual operation can be as high as $O(n)$. We will now study another example of a binary search tree that has good amortized efficiency, called a *scapegoat tree*. The idea underlying the scapegoat tree was due initially to Arne Anderson (of AA trees) in 1989. This idea was rediscovered by Galperin and Rivest in 1993, who made some refinements and gave it the name "scapegoat tree," which we will explain below.[11]

**Wreck it Ralph:** While amortized data structures often interesting in their own right, there is a particular motivation for studying the scapegoat tree. So far, all of the binary search trees that we have studied achieve balance "incrementally" through the use of rotation operation. Scapegoat trees are unique in that they do not rely on rotation. This is significant because there exist binary trees that cannot be balanced through the use of rotations. (This is often true of multidimensional data structures in geometry.) As we shall see, the scapegoat tree achieves good balance in a very different way—when a subtree is imbalanced, it is tossed out and rebuilt from scratch (or "wrecked and fixed").

Below, we will discuss the details of how the scapegoat tree works. Here is a high-level overview. A scapegoat tree is a binary search tree, which does not need to store any additional information in the nodes, other than the key, value, and left and right child pointers. Nonetheless, the height of a tree with $n$ nodes will always be $O(\log n)$. (Note that this is not the case for splay trees, whose height can grow to as high as $O(n)$.)

In an ideally balanced tree, each child has almost nearly exactly half as many nodes as its parent. Define the *size* of a node to be the total number of nodes in its subtree. A node is

[11] As you read further, you may wonder, "who would every think up such a weird data structure?" Keep in mind that two different researchers came up with essentially the same idea independently!

said to be *weight balanced* if the sizes of its two subtrees are within a constant fraction of either other (e.g., split no worse than $\frac{1}{3} \cdots \frac{2}{3}$). The scapegoat tree attempts to maintain this property. Insertion and deletions work roughly as follows.

**Insertion:**

- The key is first inserted just as in a standard (unbalanced) binary tree
- We monitor the depth of the inserted node after each insertion, and if it is too high, we can infer there must be at least one node on the search path that has poor weight balance (that is, its left and right children have very different sizes).
- In such a case, we find such a node, called the *scapegoat*,[12] and we completely rebuild the subtree rooted at this node so that it is perfectly balanced.

**Deletion:**

- The key is first deleted just as in a standard (unbalanced) binary tree
- Once the number of deletions performed is sufficiently large relative to the entire tree size, rebuild the entire tree so it is perfectly balanced.

**Why the Asymmetry?** You might wonder why there is a notable asymmetry between the rebuilding rules for insertion and deletion. The existence of a single very deep node is proof that a tree is out of balance. Thus, for insertion, we can use the fact that the inserted node is too deep to trigger rebuilding. However, observe that the converse does not work for deletion.

The natural counterpart would be "if the depth of the leaf node containing the deleted key is too small, then trigger a rebuilding operation." But the fact that a single node has a low depth, does not imply that the rest of the tree is out of balance. (It may just be that a single search path has low depth, but the rest of the tree is perfectly balanced.)

Could we instead apply the deletion rebuilding trigger to work for insertion? Again, this will not work. The natural counterpart would be, "given a newly rebuild tree with $n$ keys, we will rebuild it after inserting roughly $n/2$ new keys." However, if we are very unlucky, all these keys may fall along a single search path, and the tree's height would be as bad as $O((\log n) + n/2) = O(n)$, and this is unacceptably high.

**How to Rebuild a Subtree:** Before getting to the details of how the scapegoat tree works, let's consider the basic operation that is needed to maintain balance, namely rebuilding subtrees into balanced form. We shall see that if the subtree contains $n$ keys, this operation can be performed in $O(n)$ time (see Fig. 134). Letting `p` denote the root node of the subtree to rebuild, call this function `rebuild(p)`:

- Perform an inorder traversal of `p`'s subtree, copying the keys to an array `A[0..k-1]`, where `k` denotes the number of nodes in this subtree. Note that the elements of $A$ are sorted.
- Invoke the following recursive subtree-building function: `buildSubtree(A)`
  - Let `k = A.length`.
  - If `k == 0`, return an empty tree, that is, `null`.

[12] The colorful term "scapegoat" refers to an individual who is assigned the blame when something goes wrong. In this case, the unbalanced node takes the blame for the tree's height being too great.

– Otherwise, let `x` be the median key, that is, `A[j]`, where `j` $= \lfloor$`k/2`$\rfloor$. Recursively invoke `L = buildSubtree(A[0..j-1])` and `R = buildSubtree(A[j+1..k-1])`. Finally, create an internal node containing $x$ with left subtree $L$ and right subtree $R$. Return a pointer to $x$.

Note that if `A` is implemented as a Java `ArrayList`, there is a handy function called `sublist` for performing the above splits. The function is given in the code block below.

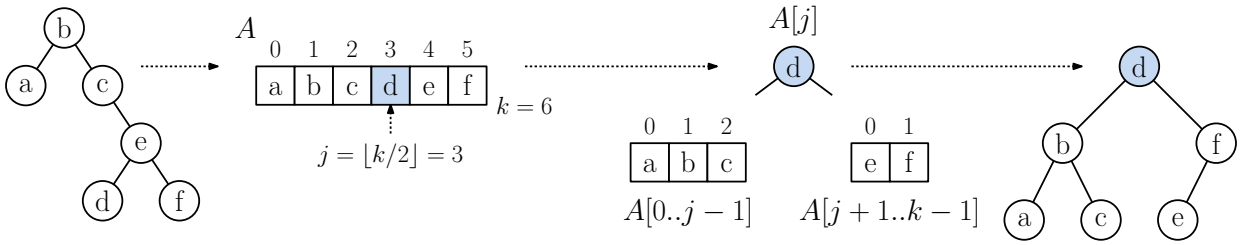



Fig. 134: Recursively building a balanced tree from a sorted array of keys.

Building a Balanced Tree from an Array

```
Node buildSubtree(Key[] A) {                    // A is a sorted array of keys
    k = A.length
    if (k == 0) return null                     // empty array
    else {
        j = floor(k/2)                          // median of the array
        Node p = new Node(A[j])                 // ...this is the root
        p.left = buildSubtree(A[0..j-1])        // build left subtree recursively
        p.right = buildSubtree(A[j+1..k-1])     // build right subtree recursively
        return p                                // return root of the subtree
    }
}
```

Ignoring the recursive calls, we spend $O(1)$ time in each recursive call, so the overall time is proportional to the size of the tree, which is $k$, so the total time is $O(k)$.

**Scapegoat Tree Operations:** In addition to the nodes themselves, the scapegoat tree maintains two integer values. The first, denoted by $n$, is just the actual number of keys in the tree. The second, denoted by $m$, is a special parameter, which is used to trigger the event of rebuilding the entire tree.

In particular, whenever we insert a key, we increment $m$, but whenever we delete a key we do not decrement $m$. Thus, $m \geq n$. The difference $m - n$ intuitively represents the number of deletions. When we reach a point where $m > 2n$ (or equivalently $m - n > n$) we can infer that the number of deletions exceeds the number of keys remaining in the tree. In this case, we rebuild the entire tree in balanced form.

We are now in a position to describe how to perform the dictionary operations for a scapegoat tree.

Initialization: $n \leftarrow m \leftarrow 0$ and the root is initialized to `null`.

`find(Key x):` The find operation is performed exactly as in a standard (unbalanced) binary search tree. We will show that the height of the tree never exceeds $\log_{3/2} n \approx 1.7 \lg n$, so this is guaranteed to run in $O(\log n)$ time.

`delete(Key x)`: This operates exactly the same as deletion in a standard binary search tree. After deleting the node, decrement $n$ (but do not change $m$). If $m > 2n$, rebuild the entire tree by invoking `rebuild(root)`, and set $m \leftarrow n$.

`insert(Key x, Value v)`: First, increment both $n$ and $m$. We start by applying the insertion process a standard (unbalanced) binary search tree. As we are tracing the search path to the insertion point, keep track of our depth in the tree. (Recall that depth is the number of edges to root.) If the depth of the inserted node exceeds $\log_{3/2} m$ then we trigger a *rebuilding event*. This involves the following:

- Walk back up along the insertion search path towards the root. Let `p` be the current node that is visited, and let `p.child` be the child of `p` that lies on the search path.
- Let `size(p)` denote the *size* of the subtree rooted at `p`, that is, the number of nodes in this subtree.
- If
$$\frac{\texttt{size(p.child)}}{\texttt{size(p)}} > \frac{2}{3},$$
then rebuild the subtree rooted at `p` by invoking `rebuild(p)`. The node `p` is the *scapegoat*. After the rebuild is done, we terminate the insertion process. Even if `p` has an ancestor that satisfies the scapegout condition, we do not do a second rebuild as part of this insertion.

An example of insertion is shown in Fig. 135. After inserting 5, the tree has $n = 11$ nodes. The newly inserted node is at depth 6, and since $6 > \log_{3/2} 11$ (which is approximately 5.9), we trigger the rebuilding event. We walk back up the search path. We find node 9 whose size is 7, but the child on the search path has size 6, and $6/7 > 2/3$, so we invoke rebuild on the node containing 9.

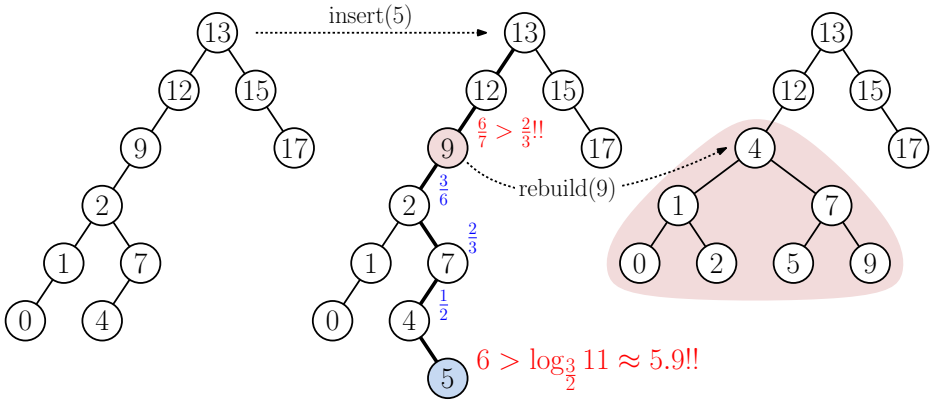



Fig. 135: Inserting a key into a scapegoat tree, which triggers a rebuilding event. The node containing 9 is the first scapegoat candidate encountered while backtracking up the search path and is rebuilt.

**Must there be a scapegoat?** The fact that a child has over 2/3 of the nodes of the entire subtree intuitively means that this subtree has (roughly) more than twice as many nodes as its sibling. We call such a node on the search path a *scapegoat candidate*. A short way of summarize the above process is "rebuild the scapegoat candidate that is closest to the insertion point.'

You might wonder whether we will necessarily encounter an scapegoat candidate when we trace back along the search path. The following lemma shows that this is always the case.

**Lemma:** Given a binary search tree of $n$ nodes, if there exists a node $p$ such that $\text{depth}(p) > \log_{3/2} n$, then $p$ has an ancestor (possibly $p$ itself) that is a scapegoat candidate.

**Proof:** The proof is by contradiction. Suppose to the contrary that no node from $p$ to the root is a scapegoat candidate. This means that for every ancestor node $u$ from $p$ to the root, we have $\text{size}(u.\text{child}) \leq \frac{2}{3} \cdot \text{size}(u)$.

We know that the root has a size of $n$. Therefore, its child on the search path has size at most $(2/3)n$, its grandchild has size at most $(2/3)((2/3)n) = (4/9)n$, and generally the node at depth $i$ along the search path as size at most $(2/3)^i n$.

Let $d$ denote the depth of $p$. We know what its subtree rooted at $p$ must have at least one node (namely $p$ itself), and therefore

$$1 \;\leq\; \text{size}(p) \;\leq\; \left(\frac{2}{3}\right)^d n.$$

Solving for $d$, we have

$$\left(\frac{3}{2}\right)^d \;\leq\; n \quad\Longrightarrow\quad d \;\leq\; \log_{3/2} n.$$

However, this violates our hypothesis that $p$'s depth exceeds $\log_{3/2} n$, yielding the desired contradiction.

Recall that $m \geq n$, and so if a rebuilding event is triggered, the insertion depth is at least $\log_{3/2} m$, which means that it is at depth at least $\log_{3/2} n$. Therefore, by the above lemma, there must be a scapegoat candidate along the search path.

**The Sizeless Size?** We mentioned earlier that the scapegoat tree does not store any information in the nodes other than the key, value, and left and right child pointers. So how can we compute `size(u)` for a node `u` during the insertion process, without this information? There is a clever trick for doing this on the fly.

Since we are doing this as we back up the search path, we may assume that we already know the value of $s' =$ `size(u.child)`, where this is the child that lies along the insertion search path. So, to compute `size(u)`, it suffices to compute the size of `u`'s other child. To do this, we perform a traversal of this child's subtree to determine its size $s''$. Given this, we have $\texttt{size}(u) = 1 + s' + s''$, where the $+1$ counts the node `u` itself.

You might wonder, how can we possibly expect to achieve $O(\log n)$ amortized time for insertion if we are using brute force (which may take as much as $O(n)$ time) to compute the sizes of the subtrees? The reason is to first recall that we do not need to compute subtree sizes unless a rebuild event has been triggered. Every node that we are visiting in the counting process will need to be visited again in the rebuilding process. Thus, the cost of this counting process can be "charged for" in the cost of the rebuilding process, and hence it essentially comes for free!

**Sizes the Easy Way:** By the way, there is a more direct method for computing subtree sizes. Just store the size value of each node explicitly within each node. The size of a node is easy to update whenever there are changes in the tree's structure, since we have:

```
size(u) = (u == null ? 0 : size(u.left) + size(u.right))
```

While we are at it, it is worth noting that the height is just as easy to store and update:

```
height(u) = (u == null ? 0 : 1 + max(height(u.left), height(u.right)))
```

**Amortized Analysis:** We will not present a formal analysis of the amortized analysis of the scapegoat tree. The following theorem (and the rather sketchy proof that follows) provides the main results, however.

**Theorem:** Starting with an empty tree, any sequence of $m$ dictionary operations (find, insert, and delete) to a scapegoat tree can be performed in time $O(m \log m)$. (Don't confuse $m$ here with the $m$ used in the algorithm.)

**Proof:** The proof is quite detailed, so we will just sketch the key ideas. In order for each rebuild operation to be triggered, you need to perform a lot of cheap (rebuild-free) operations to get into an unbalanced state. We will explain the results in terms of $n$, the number of items in the tree, but note that $m$ is generally larger.

**Find:** Because the tree's height is at most $\log_{3/2} m \leq \log_{3/2}(2n) = O(\log n)$ the costs of a find operation is $O(\log n)$ (unconditionally).

**Delete:** In order to rebuild the entire tree due to deletions, at least half the entries since the last full rebuild must have been deleted. (The value $m - n$ is the number of deletions, and a rebuild is triggered when $m > 2n$, implying that $m - n > n$.) By token-based analyses, it follows that the $O(n)$ cost of rebuilding the entire tree can be amortized against the time spent processing the (inexpensive) deletions.

**Insert:** This is analyzed by a potential argument. This is complicated by the fact that subtrees of various sizes can be rebuilt. Intuitively, after any subtree of size $k$ is rebuilt, it takes an additional $O(k)$ (inexpensive) operations to force this subtree to become unbalanced and hence to be rebuilt again. We charge the expense of rebuilding to against these "cheap" insertions.

## Supplemental Lecture 4: Geometric Preliminaries

**Geometric Information:** A large number of modern data structures and algorithms problems involve geometric data. The reason is that rapidly growing fields such as computer graphics, robotics, computer vision, computer-aided design, visualization, human-computer interaction, virtual reality, and others deal primarily with geometric data. Geometric applications give rise to new data structures problems, which we will be studying for a number of lectures.

Before discussing geometric data structures, we need to provide some background on what geometric data is, and how we compute with it. With nongeometric data we stated that we are storing records, and each record is associated with an identifying *key* value. These key values served a number of functions for us. They served a function of *identification* the the objects of the data structure. Because of the underlying ordering relationship, they also provided a means for searching for objects in the data structure, by giving us a way to direct the search by *subdividing* the space of keys into subsets that are greater than or less than the key.

In geometric data structures we will need to generalize the notion of a key. A geometric point object in the plane can be identified by its $(x, y)$ coordinates, which can be thought of as a type of key value. However, if we are storing more complex objects, such as rectangles, line segments, spheres, and polygons, the notion of a identifying key is not as appropriate. As with one-dimensional data, we also have associated application data. For example, in a

graphics system, the geometric description of an object is augmented with information about the object's color, texture, and its surface reflectivity properties. Since our interest will be in storing and retrieving objects based on their geometric properties, we will not discuss these associated data values.

**Primitive Objects:** Before we start discussing geometric data structures, we will digress to discuss a bit about geometric objects, their representation, and manipulation. Here is a list of common geometric objects and possible representations. The list is far from complete. Let $\mathbb{R}^d$ denote $d$-dimensional space with real coordinates.

**Scalar:** This is a single (1-dimensional) real number. It is represented as float or double.

**Point:** Points are locations in space. A typical representation is as a $d$-tuple of scalars, e.g. $p = (p_0, p_1, \ldots, p_{d-1}) \in \mathbb{R}^d$. Is it better to represent a point as an array of scalars, or as an object with data members labeled $x$, $y$, and $z$? The array representation is more general and often more convenient, since it is easier to generalized to higher dimensions and coordinates can be parameterized. You can define "names" for the coordinates. If the dimension might vary then the array representation is necessary.

For example, in Java we might represent a point in 3-space using something like the following. (Note that "`static final`" essentially means "constant" in this context.)

```
class Point {
  public static final int DIM = 3;
  protected float coord[DIM];
  ...
}
```

**Vector:** Vectors are used to denote direction and magnitude in space. Vectors and points are represented in essentially the same way, as a $d$-tuple of scalars, $\vec{v} = (v_0, v_1, \ldots, v_{d-1})$. It is often convenient to think of vectors as *free vectors*, meaning that they are not tied down to a particular origin, but instead are free to roam around space. The reason for distinguishing vectors from points is that they often serve significantly different functions. For example, velocities are frequently described as vectors, but locations are usually described as points. This provides the reader of your program a bit more insight into your intent.

**Line Segment:** A line segment can be represented by giving its two endpoints $(p_1, p_2)$. In some applications it is important to distinguish between the line segments $\overrightarrow{p_1p_2}$ and $\overrightarrow{p_2p_1}$. In this case they would be called *directed line segments*.

**Ray:** Directed lines in 3- and higher dimensions are not usually represented by equations but as rays. A ray can be represented by storing an origin point $p$ and a nonzero directional vector $\vec{v}$.

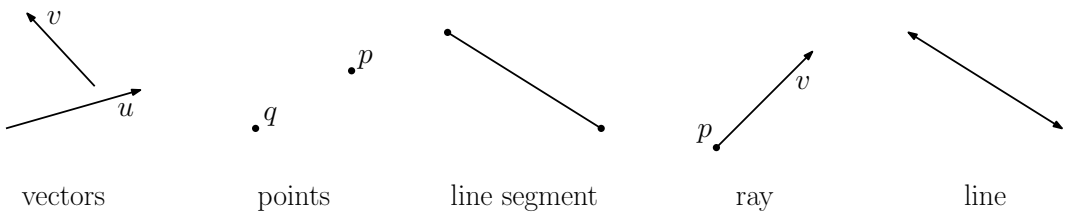



Fig. 136: Basic geometric objects.

**Line:** A line in the plane can be represented by a line equation

$$y = ax + b \qquad \text{or} \qquad ax + by = c.$$

The former definition is the familiar *slope-intercept* representation, and the latter takes an extra coefficient but is more general since it can easily represent a vertical line. The representation consists of just storing the pair $(a, b)$ or $(a, b, c)$.

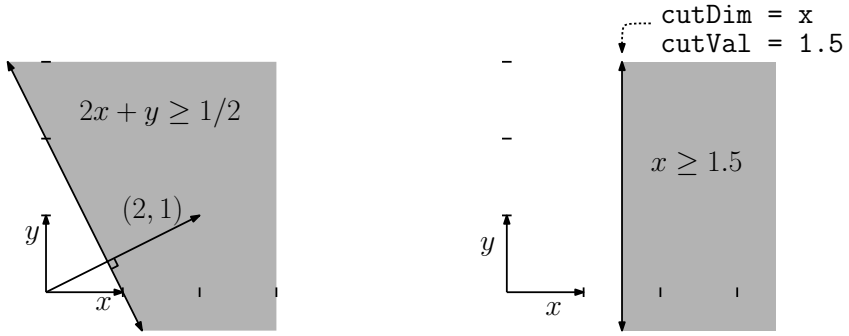
Another way to represent a line is by giving two points through which the line passes, or by giving a point and a directional vector. These latter two methods have the advantage that they generalize to higher dimensions.

**Hyperplanes and halfspaces:** In general, in dimension $d$, a linear equation of the form

$$a_0p_0 + a_1p_1 + \ldots + a_{d-1}p_{d-1} = c$$

defines a $d-1$ dimensional hyperplane. It can be represented by the $d$-tuple $(a_0, a_1, \ldots, a_{d-1})$ together with $c$. Note that the vector $(a_0, a_1, \ldots, a_{d-1})$ is orthogonal to the hyperplane.

The set of points that lie on one side or the other of a hyperplane is called a *halfspace*. The formula is the same as above, but the "=" is replaced with an inequality such as "$<$" or "$\geq$".




Halfspace Orthogonal Halfspace

Fig. 137: Planes and Halfspaces.

**Orthogonal Hyperplane:** In many data structures it is common to use hyperplanes that are orthogonal to one of the coordinate axes, called *orthogonal hyperplanes*. In this case it is much easier to store such a hyperplane by storing (1) an integer `cutDim` from the set $\{0, 1, \ldots, d-1\}$, which indicates which axis the plane is perpendicular to and (2) a scalar `cutVal`, which indicates where the plane cuts this axis.

**Simple Polygon:** Solid objects can be represented as polygons (in dimension 2) and polyhedra (in higher dimensions). A polygon is a cycle of line segments joined end-to-end. It is said to be *simple* if its boundary does not self-intersect. It is *convex* if it is simple and no internal angle is greater than $\pi$.

The standard representation of a simple polygon is just a list of points (stored either in an array or a circularly linked list). In general representing solids in higher dimensions involves more complex structures, which we will discuss later.

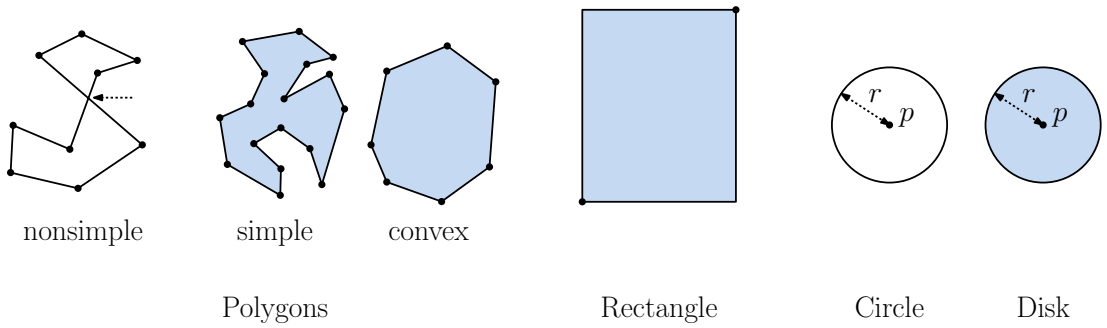



Fig. 138: Polygons, rectangles and spheres.

**Orthogonal Rectangle:** Rectangles can be represented as polygons, but rectangles whose sides are parallel to the coordinate axes are common. A couple of representations are often used. One is to store the two points of opposite corners (e.g. lower left and upper right).

**Circle/Sphere:** A $d$-dimensional sphere, can be represented by *point* $p$ indicating the center of the sphere, and a positive *scalar* $r$ representing its radius. A point $x$ lies within the sphere if

$$(x_0 - p_0)^2 + (x_1 - p_1)^2 + \ldots + (x_{d-1} - p_{d-1})^2 \ \leq \ r^2.$$

**Topological Notions:** When discussing geometric objects such as circles and polygons, it is often important to distinguish between what is inside, what is outside and what is on the boundary. For example, given a triangle $T$ in the plane we can talk about the points that are in the *interior* (*extitint*$(T)$) of the triangle, the points that lie on the *boundary* (*extitbnd*$(T)$) of the triangle, and the points that lie on the *exterior* (*extitext*$(T)$) of the triangle. A set is *closed* if it includes its boundary, and *open* if it does not. Sometimes it is convenient to define a set as being *semi-open*, meaning that some parts of the boundary are included and some are not. Making these notions precise is tricky, so we will just leave this on an intuitive level.

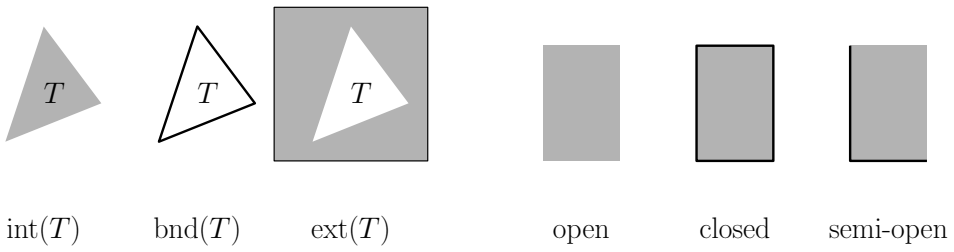



Fig. 139: Topological notation.

**Operations on Primitive Objects:** When dealing with simple numeric objects in 1-dimensional data structures, the set of possible operations needed to be performed on each primitive object was quite simple, e.g. compare one key with another, add or subtract keys, print keys, etc. With geometric objects, there are many more operations that one can imagine.

**Basic point/vector operators:** Let $\alpha$ be any scalar, let $p$ and $q$ be points, and $\vec{u}, \vec{v}, \vec{w}$ be vectors. We think of vectors as being free to roam about the space whereas points are tied down to a particular location. We can do all the standard operations on vectors you

learned in linear algebra (adding, subtracting, etc.) The difference of two points $p - q$ results in the vector directed from $q$ to $p$. The sum of a point $p$ and vector $\vec{u}$ is the point lying on the head of the vector when its tail is placed on $p$.

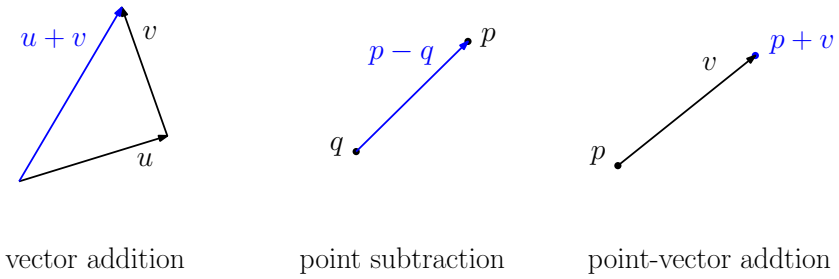



Fig. 140: Point-vector operators.

As a programming note, observe that C++ has a very nice mechanism for handling these operations on Points and Vectors using operator overloading. Java does not allow overloading of operators.

**Affine combinations:** Given two points $p$ and $q$, the point $(1-\alpha)p+\alpha q$ is point on the line joining $p$ and $q$. We can think of this as a weighted average, so that as $\alpha$ approaches 0 the point is closer to $p$ and as $\alpha$ approaches 1 the point is closer to $q$. This is called an *affine combination* of $p$ and $q$. If $0 \leq \alpha \leq 1$, then the point lies on the line segment $\overline{pq}$.

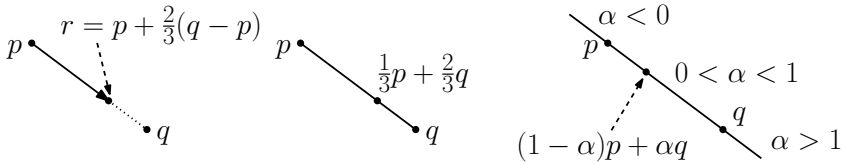



Fig. 141: Affine combinations.

**Length and distance:** The length of a vector $v$ is defined to be

$$\|\vec{v}\| = \sqrt{v_0^2 + v_1^2 + \ldots + v_{d-1}^2}.$$

The distance between two points $p$ and $q$ is the length of the vector between them, that is

$$extitdist(p, q) = \|p - q\|.$$

Computing distances between other geometric objects is also important. For example, what is the distance between two triangles? When discussing complex objects, distance usually means the closest distance between objects.

**Orientation and Membership:** There are a number of geometric operations that deal with the relationship between geometric objects. Some are easy to solve. (E.g., does a point $p$ lie within a rectangle $R$?) Some are harder. (E.g., given points $a$, $b$, $c$, and $d$, does $d$ lie within the unique circle defined by the other three points?) We will discuss these as the need arises.

**Intersections:** The other sort of question that is important is whether two geometric objects intersect each other. Again, some of these are easy to answer, and others can be quite complex.

**Example: Circle/Rectangle Intersection:** As an example of a typical problem involving geometric primitives, let us consider the question of whether a circle in the plane intersects a rectangle. Let us represent the circle by its center point $c$ and radius $r$ and represent the rectangle $R$ by its lower left and upper right corner points, $R_{lo}$ and $R_{hi}$ (for low and high).

Problems involving circles are often more easily recast as problems involving distances. So, instead, let us consider the problem of determining the distance $d$ from $c$ to its closest point on the rectangle $R$. If $d > r$ then the circle does not intersect the rectangle, and otherwise it does.

In order to determine the distance from $c$ to the rectangle, we first observe that if $c$ lies inside $R$, then the distance is 0. Otherwise we need to determine which point of the boundary of $R$ is closest to $c$. Observe that we can subdivide the exterior of the rectangle into 8 regions, as shown in the figure below. If $c$ lies in one of the four corner regions, then $c$ is closest to the corresponding vertex of $R$ and otherwise $c$ is closest to one of the four sides of $R$. Thus, all we need to do is to classify which region $c$ lies is, and compute the corresponding distance. This would usually lead a lengthy collection of if-then-else statements, involving 9 different cases.

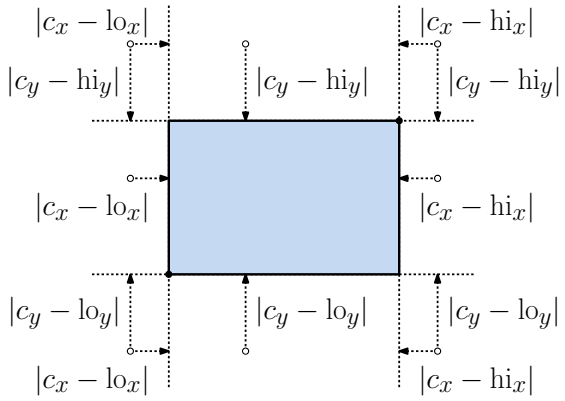



Fig. 142: Distance from a point to a rectangle, and squared distance contributions for each region.

We will take a different approach. Sometimes things are actually to code if you consider the problem in its general $d$-dimensional form. Rather than computing the distance, let us first concentrate on computing the squared distance instead. The distance is the sum of the squares of the distance along the $x$-axis and distance along the $y$-axis. Consider just the $x$-coordinates, if $c$ lies to are to the left of the rectangle then the contribution is $(R_{lo,x} - c_x)^2$, if $c$ lies to the right then the contribution is $(c_x - R_{hi,x})^2$, and otherwise there is no $x$-contribution to the distance. a similar analysis applies for $y$. This suggests the following code, which works in all dimensions.

Finally, once the distance has been computed, we test whether it is less than the radius $r$. If so the circle intersects the rectangle and otherwise it does not.

## Supplemental Lecture 5: Interval Trees

**Segment Data:** So far we have considered geometric data structures for storing points. However, there are many others types of geometric data that we may want to store in a data structure. Today we consider how to store orthogonal (horizontal and vertical) line segments in the

Distance from Point $c$ to Rectangle $R$

```
float distance(Point c, Rectangle R) {
    sumSq = 0                                  // sum of squares
    for (int i = 0; i < Point.DIM; i++) {
        if (c[i] < R.lo[i])                    // left of rectangle
            sumSq += square(R.lo[i] - c[i])
        else if (c[i] > R.hi[i])               // right of rectangle
            sumSq += square(c[i] - R.hi[i])
    }
    return sqrt(sumSq)
}
```

plane. We assume that a line segment is represented by giving its pair of *endpoints*. The segments are allowed to intersect one another.

As a basic motivating query, we consider the following *window query*. Given a set of orthogonal line segments $S$, which have been preprocessed, and given an orthogonal query rectangle $W$, count or report all the line segments of $S$ that intersect $W$. We will assume that $W$ is closed and solid rectangle, so that even if a line segment lies entirely inside of $W$ or intersects only the boundary of $W$, it is still reported. For example, given the window below, the query would report the segments that are shown with solid lines, and segments with broken lines would not be reported.

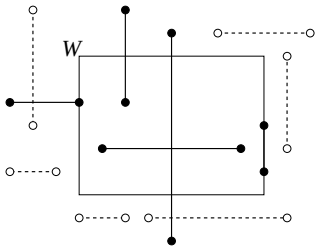



Fig. 143: Window Query.

**Window Queries for Orthogonal Segments:** We will present a data structure, called the *interval tree*, which (combined with a range tree) can answer window counting queries for orthogonal line segments in $O(\log^2 n)$ time, where $n$ is the number line segments. It can report these segments in $O(k + \log^2 n)$ time, where and $k$ is the total number of segments reported. The interval tree uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time.

We will consider the case of range reporting queries. (There are some subtleties in making this work for counting queries.) We will derive our solution in steps, starting with easier subproblems and working up to the final solution. To begin with, observe that the set of segments that intersect the window can be partitioned into three types: those that have no endpoint in $W$, those that have one endpoint in $W$, and those that have two endpoints in $W$.

We already have a way to report segments of the second and third types. In particular, we may build a range tree just for the $2n$ endpoints of the segments. We assume that each endpoint has a cross-link indicating the line segment with which it is associated. Now, by applying a range reporting query to $W$ we can report all these endpoints, and follow the cross-links to report the associated segments. Note that segments that have both endpoints in the

window will be reported twice, which is somewhat unpleasant. We could fix this either by sorting the segments in some manner and removing duplicates, or by marking each segment as it is reported and ignoring segments that have already been marked. (If we use marking, after the query is finished we will need to go back an "unmark" all the reported segments in preparation for the next query.)

All that remains is how to report the segments that have no endpoint inside the rectangular window. We will do this by building two separate data structures, one for horizontal and one for vertical segments. A horizontal segment that intersects the window but neither of its endpoints intersects the window must pass entirely through the window. Observe that such a segment intersects any vertical line passing from the top of the window to the bottom. In particular, we could simply ask to report all horizontal segments that intersect the left side of $W$. This is called a *vertical segment stabbing query*. In summary, it suffices to solve the following subproblems (and remove duplicates):

**Endpoint inside:** Report all the segments of $S$ that have at least one endpoint inside $W$. (This can be done using a range query.)

**Horizontal through segments:** Report all the horizontal segments of $S$ that intersect the left side of $W$. (This reduces to a vertical segment stabbing query.)

**Vertical through segments:** Report all the vertical segments of $S$ that intersect the bottom side of $W$. (This reduces to a horizontal segment stabbing query.)

We will present a solution to the problem of vertical segment stabbing queries. Before dealing with this, we will first consider a somewhat simpler problem, and then modify this simple solution to deal with the general problem.

**Vertical Line Stabbing Queries:** Let us consider how to answer the following query, which is interesting in its own right. Suppose that we are given a collection of horizontal line segments $S$ in the plane and are given an (infinite) vertical query line $\ell_q : x = x_q$. We want to report all the line segments of $S$ that intersect $\ell_q$. Notice that for the purposes of this query, the $y$-coordinates are really irrelevant, and may be ignored. We can think of each horizontal line segment as being a closed *interval* along the $x$-axis. We show an example in the figure below on the left.

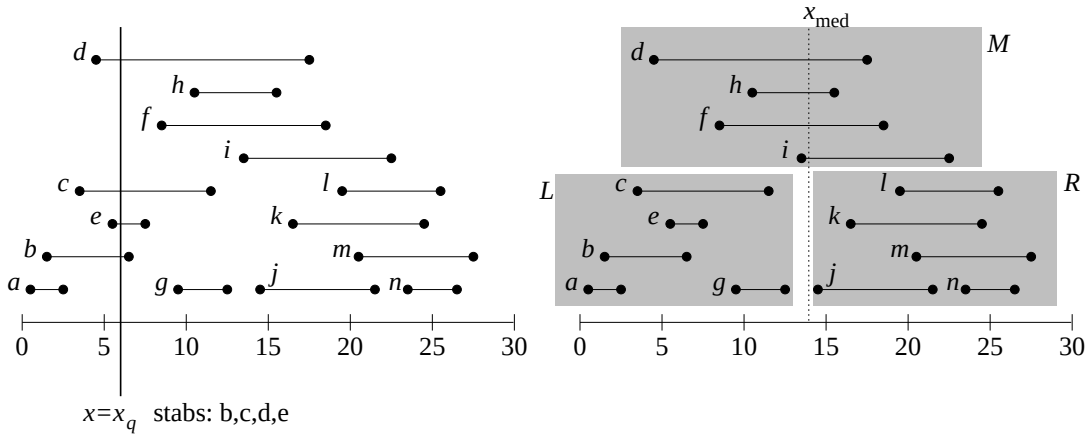



Fig. 144: Line Stabbing Query.

As is true for all our data structures, we want some balanced way to decompose the set of intervals into subsets. Since it is difficult to define some notion of order on intervals, we instead will order the endpoints. Sort the interval endpoints along the $x$-axis. Let $\langle x_1, x_2, \ldots, x_{2n} \rangle$ be the resulting sorted sequence. Let $x_{\text{med}}$ be the median of these $2n$ endpoints. Split the intervals into three groups, $L$, those that lie strictly to the left of $x_{\text{med}}$, $R$ those that lie strictly to the right of $x_{\text{med}}$, and $M$ those that contain the point $x_{\text{med}}$. We can then define a binary tree by putting the intervals of $L$ in the left subtree and recursing, putting the intervals of $R$ in the right subtree and recursing. Note that if $x_q < x_{\text{med}}$ we can eliminate the right subtree and if $x_q > x_{\text{med}}$ we can eliminate the left subtree. See the figure right.

But how do we handle the intervals of $M$ that contain $x_{\text{med}}$? We want to know which of these intervals intersects the vertical line $\ell_q$. At first it may seem that we have made no progress, since it appears that we are back to the same problem that we started with. However, we have gained the information that all these intervals intersect the vertical line $x = x_{\text{med}}$. How can we use this to our advantage?

Let us suppose for now that $x_q \leq x_{\text{med}}$. How can we store the intervals of $M$ to make it easier to report those that intersect $\ell_q$. The simple trick is to sort these lines in increasing order of their left endpoint. Let $M_L$ denote the resulting sorted list. Observe that if some interval in $M_L$ does not intersect $\ell_q$, then its left endpoint must be to the right of $x_q$, and hence none of the subsequent intervals intersects $\ell_q$. Thus, to report all the segments of $M_L$ that intersect $\ell_q$, we simply traverse the sorted list and list elements until we find one that does not intersect $\ell_q$, that is, whose left endpoint lies to the right of $x_q$. As soon as this happens we terminate. If $k'$ denotes the total number of segments of $M$ that intersect $\ell_q$, then clearly this can be done in $O(k' + 1)$ time.

On the other hand, what do we do if $x_q > x_{\text{med}}$? This case is symmetrical. We simply sort all the segments of $M$ in a sequence, $M_R$, which is sorted from right to left based on the right endpoint of each segment. Thus each element of $M$ is stored twice, but this will not affect the size of the final data structure by more than a constant factor. The resulting data structure is called an *interval tree*.

**Interval Trees:** The general structure of the interval tree was derived above. Each node of the interval tree has a left child, right child, and itself contains the median $x$-value used to split the set, $x_{\text{med}}$, and the two sorted sets $M_L$ and $M_R$ (represented either as arrays or as linked lists) of intervals that overlap $x_{\text{med}}$. We assume that there is a constructor that builds a node given these three entities. The following high-level pseudocode describes the basic recursive step in the construction of the interval tree. The initial call is `root = IntTree(S)`, where $S$ is the initial set of intervals. Unlike most of the data structures we have seen so far, this one is not built by the successive insertion of intervals (although it would be possible to do so). Rather we assume that a set of intervals $S$ is given as part of the constructor, and the entire structure is built all at once. We assume that each interval in $S$ is represented as a pair $(x_{\text{lo}}, x_{\text{hi}})$. An example is shown in the following figure.

We assert that the height of the tree is $O(\log n)$. To see this observe that there are $2n$ endpoints. Each time through the recursion we split this into two subsets $L$ and $R$ of sizes at most half the original size (minus the elements of $M$). Thus after at most $\lg(2n)$ levels we will reduce the set sizes to 1, after which the recursion bottoms out. Thus the height of the tree is $O(\log n)$.

Implementing this constructor efficiently is a bit subtle. We need to compute the median of the set of all endpoints, and we also need to sort intervals by left endpoint and right endpoint.

```
IntTreeNode IntTree(IntervalSet S) {
    if (|S| == 0) return null                              // no more

    xMed = median endpoint of intervals in S               // median endpoint

    L = {[xlo, xhi] in S | xhi < xMed}                     // left of median
    R = {[xlo, xhi] in S | xlo > xMed}                     // right of median
    M = {[xlo, xhi] in S | xlo <= xMed <= xhi}             // contains median
    ML = sort M in increasing order of xlo                 // sort M
    MR = sort M in decreasing order of xhi

    t = new IntTreeNode(xMed, ML, MR)                      // this node
    t.left  = IntTree(L)                                   // left subtree
    t.right = IntTree(R)                                   // right subtree
    return t
}
```

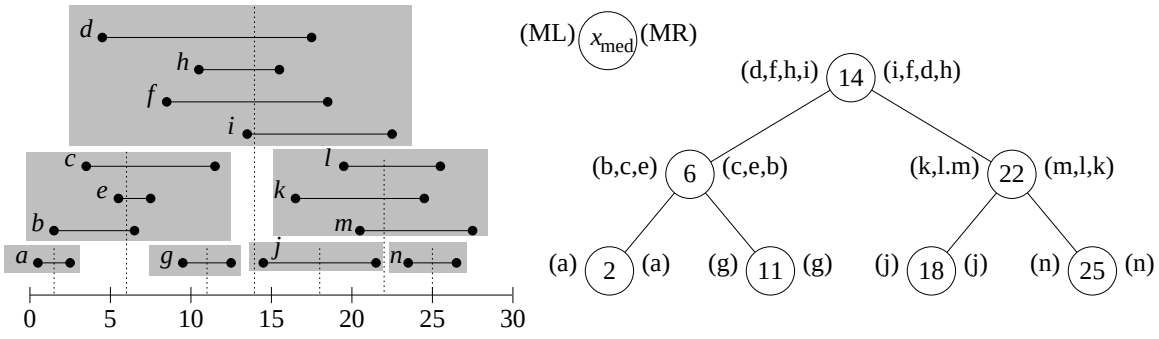



Fig. 145: Interval Tree.

The fastest way to do this is to presort all these values and store them in three separate lists. Then as the sets $L$, $R$, and $M$ are computed, we simply copy items from these sorted lists to the appropriate sorted lists, maintaining their order as we go. If we do so, it can be shown that this procedure builds the entire tree in $O(n \log n)$ time. The algorithm for answering a stabbing query was derived above. We summarize this algorithm below. Let $x_q$ denote the $x$-coordinate of the query line.

Line Stabbing Queries for an Interval Tree

```
stab(IntTreeNode t, Scalar xq) {
    if (t == null) return                       // fell out of tree
    if (xq < t.xMed) {                          // left of median?
        for (i = 0; i < t.ML.length; i++) {     // traverse ML
            if (t.ML[i].lo <= xq) print(t.ML[i])// ..report if in range
            else break                          // ..else done
        }
        stab(t.left, xq)                        // recurse on left
    }
    else {                                      // right of median
        for (i = 0; i < t.MR.length; i++) {     // traverse MR
            if (t.MR[i].hi >= xq) print(t.MR[i])// ..report if in range
            else break                          // ..else done
        }
        stab(t.right, xq)                       // recurse on right
    }
}
```

This procedure actually has one small source of inefficiency, which was intentionally included to make code look more symmetric. Can you spot it? Suppose that $x_q = t.x_{\text{med}}$? In this case we will recursively search the right subtree. However this subtree contains only intervals that are strictly to the right of $x_{\text{med}}$ and so is a waste of effort. However it does not affect the asymptotic running time.

As mentioned earlier, the time spent processing each node is $O(1 + k')$ where $k'$ is the total number of points that were recorded at this node. Summing over all nodes, the total reporting time is $O(k + v)$, where $k$ is the total number of intervals reported, and $v$ is the total number of nodes visited. Since at each node we recurse on only one child or the other, the total number of nodes visited $v$ is $O(\log n)$, the height of the tree. Thus the total reporting time is $O(k + \log n)$.

**Vertical Segment Stabbing Queries:** Now let us return to the question that brought us here. Given a set of horizontal line segments in the plane, we want to know how many of these segments intersect a vertical line segment. Our approach will be exactly the same as in the interval tree, except for how the elements of $M$ (those that intersect the splitting line $x = x_{\text{med}}$) are handled.

Going back to our interval tree solution, let us consider the set $M$ of horizontal line segments that intersect the splitting line $x = x_{\text{med}}$ and as before let us consider the case where the query segment $q$ with endpoints $(x_q, y_{\text{lo}})$ and $(x_q, y_{\text{hi}})$ lies to the left of the splitting line. The simple trick of sorting the segments of $M$ by their left endpoints is not sufficient here, because we need to consider the $y$-coordinates as well. Observe that a segment of $M$ stabs the query segment $q$ if and only if the left endpoint of a segment lies in the following semi-infinite

rectangular region.

$$\{(x, y) \mid x \leq x_q \text{ and } y_{\text{lo}} \leq y \leq y_{\text{hi}}\}.$$

This is illustrated in the figure below. Observe that this is just an orthogonal range query. (It is easy to generalize the procedure given last time to handle semi-infinite rectangles.) The case where $q$ lies to the right of $x_{\text{med}}$ is symmetrical.

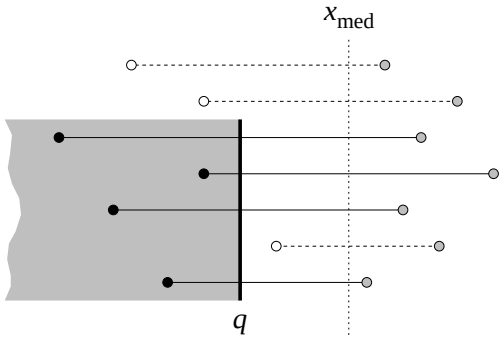



Fig. 146: The segments that stab $q$ lie within the shaded semi-infinite rectangle.

So the solution is that rather than storing $M_L$ as a list sorted by the left endpoint, instead we store the left endpoints in a 2-dimensional range tree (with cross-links to the associated segments). Similarly, we create a range tree for the right endpoints and represent $M_R$ using this structure.

The segment stabbing queries are answered exactly as above for line stabbing queries, except that part that searches $M_L$ and $M_R$ (the for-loops) are replaced by searches to the appropriate range tree, using the semi-infinite range given above.

We will not discuss construction time for the tree. (It can be done in $O(n \log n)$ time, but this involves some thought as to how to build all the range trees efficiently). The space needed is $O(n \log n)$, dominated primarily from the $O(n \log n)$ space needed for the range trees. The query time is $O(k + \log^3 n)$, since we need to answer $O(\log n)$ range queries and each takes $O(\log^2 n)$ time plus the time for reporting. If we use the spiffy version of range trees (which we mentioned but never discussed) that can answer queries in $O(k + \log n)$ time, then we can reduce the total time to $O(k + \log^2 n)$.

## Supplemental Lecture 6: Garbage Collection

**Garbage Collection:** In contrast to the explicit deallocation methods discussed in the previous lectures, in some memory management systems such as Java, there is no explicit deallocation of memory. In such systems, when memory is exhausted it must perform *garbage collection* to reclaim storage and sometimes to reorganize memory for better future performance. We will consider some of the issues involved in the implementation of such systems.

Any garbage collection system must do two basic things. First, it must detect which blocks of memory are unreachable, and hence are "garbage". Second, it must reclaim the space used by these objects and make it available for future allocation requests. Garbage detection is typically performed by defining a set of *roots*, e.g., local variables that point to objects in the heap, and then finding everything that is reachable from these roots. An object is *reachable* (or *live*) if there is some path of pointers or references from the roots by which the executing

program can access the object. The roots are always accessible to the program. Objects that are not reachable are considered garbage, because they can no longer affect the future course of program execution.

**Reference counts:** How do we know when a block of storage is able to be deleted? One simple way to do this is to maintain a *reference count* for each block. This is a counter associated with the block. It is set to one when the block is first allocated. Whenever the pointer to this block is assigned to another variable, we increase the reference count. (For example, the compiler can overload the assignment operation to achieve this.) When a variable containing a pointer to the block is modified, deallocated or goes out of scope, we decrease the reference count. If the reference count ever equals 0, then we know that no references to this object remain, and the object can be deallocated.

Reference counts have two significant shortcomings. First, there is a considerable overhead in maintaining reference counts, since each assignment needs to modify the reference counts. Second, there are situations where the reference count method may fail to recognize unreachable objects. For example, if there is a circular list of objects, for example, $X$ points to $Y$ and $Y$ points to $X$, then the reference counts of these objects may never go to zero, even though the entire list is unreachable.

**Mark-and-sweep:** A more complete alternative to reference counts involves waiting until space runs out, and then scavenging memory for unreachable cells. Then these unreachable regions of memory are returned to available storage. These available blocks can be stored in an available space list using the same method described in the previous lectures for dynamic storage allocation. This method works by finding all immediately accessible pointers, and then traces them down and *marks* these blocks as being accessible. Then we *sweep* through memory adding all unmarked blocks to the available space list. Hence this method is called *mark-and-sweep*. An example is illustrated in the figure below.

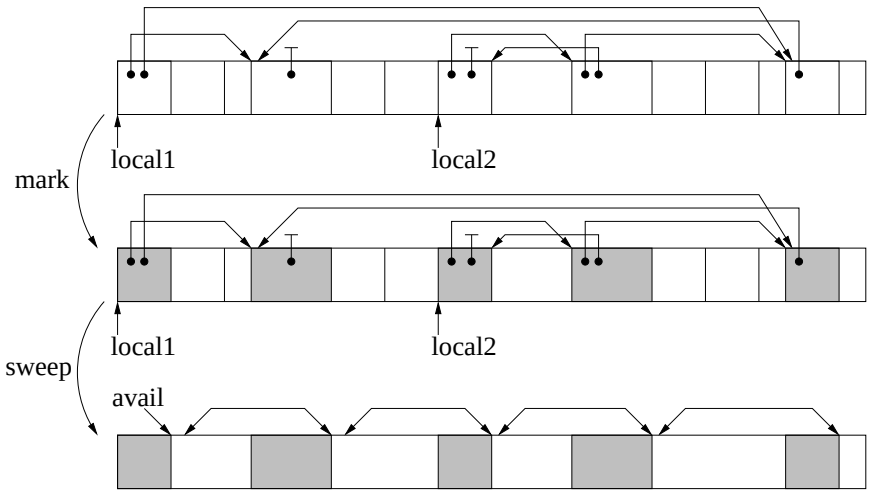



Fig. 147: Mark-and-sweep garbage collection.

How do we implement the marking phase? One way is by performing simple *depth-first traversal* of the "directed graph" defined by all the pointers in the program. We start from all the root pointers $t$ (those that are immediately accessible from the program) and invoke the following procedure for each. Let us assume that for each pointer $t$ we know its type

and hence we know the number of pointers this object contains, denoted `t.nChild` and these pointers are denoted `t.child[i]`. (Note that although we use the term "child" as if pointers form a tree, this is not the case, since there may be cycles.) We assume that each block has a bit value `t.isMarked` which indicates whether the object has been marked.

Recursive Marking Algorithm

```
mark(Pointer t) {
    if (t == null || t.isMarked) return     // null or already visited
    t.isMarked = true                       // mark t visited
    for (i = 0; i < t.nChild; i++)          // consider t's pointers
        mark(t.child[i])                    // recursively visit each one
}
```

The recursive calls persist in visiting everything that is reachable, and only backing off when we come to a null pointer or something that has already been marked. Note that we do not need to store the `t.nChild` field in each object. It is function of $t$'s type, which presumably the run-time system is aware of (especially in languages like Java that support dynamic casting). However we definitely need to allocate an extra bit in each object to store the mark.

**Marking using Link Redirection:** There is a significant problem in the above marking algorithm. This procedure is necessarily recursive, implying that we need a stack to keep track of the recursive calls. However, we only invoke this procedure if we have run out of space. So we do not have enough space to allocate a stack. This poses the problem of how can we traverse space without the use of recursion or a stack. There is no method known that is very efficient and does not use a stack. However there is a rather cute idea which allows us to dispense with the stack, provided that we allocate a few extra bits of storage in each object.

The method is called *link redirection* or the *Schorr-Deutsch-Waite method*. Let us think of our objects as being nodes in a multiway tree. (Recall that we have cycles, but because of we never revisit marked nodes, so we can think of pointers to marked nodes as if they are null pointers.) Normally the stack would hold the parent of the current node. Instead, when we traverse the link to the $i$th child, we redirect this link so that it points to the parent. When we return to a node and want to proceed to its next child, we fix the redirected child link and redirect the next child.

Pseudocode for this procedure is given below. As before, in the initial call the argument $t$ is a root pointer. In general $t$ is the current node. The variable $p$ points to $t$'s parent. We have a new field `t.currChild` which is the index of the current child of $t$ that we are visiting. Whenever the search ascends to $t$ from one of its children, we increment `t.currChild` and visit this next child. When `t.currChild == t.nChild` then we are done with $t$ and ascend to its parent. We include two utilities for pointer redirection. The call `descend(p, t, t.c)` moves us from $t$ to its child $t.c$ and saves $p$ in the pointer field containing $t.c$. The call `ascend(p, t, p.c)` moves us from $t$ to its parent $p$ and restores the contents of the $p$'s child $p.c$. Each are implemented by a performing a cyclic rotation of these three quantities. (Note that the arguments are reference arguments.) An example is shown in the figure below, in which we print the state after each descend or ascend.

$$\text{descend}(p,t,t.c): \begin{pmatrix} p \\ t \\ t.c \end{pmatrix} \longleftarrow \begin{pmatrix} t \\ t.c \\ p \end{pmatrix} \qquad \text{ascend}(p,t,p.c): \begin{pmatrix} p \\ t \\ p.c \end{pmatrix} \longleftarrow \begin{pmatrix} p.c \\ p \\ t \end{pmatrix}.$$

```
markByRedirect(Pointer t) {
    p = null                                        // parent pointer
    while (true) {
        if (t != null && !t.isMarked) {             // first time at t?
            t.isMarked = true                       // mark as visited
            if (t.nChild > 0) {                     // t has children
                t.currChild = 0                     // start with child 0
                descend(p, t, t.child[0])           // descend via child 0
            }
        }
        else if (p != null) {                       // returning to t
            j = p.currChild                         // parent's current child
            ascend(p, t, p.child[j])                // ascend via child j
            j = ++t.currChild                       // next child
            if (j < t.nChild) {                     // more children left
                descend(p, t, t.child[j])           // descend via child j
            }
        }
        else return                                 // no parent? we're done
    }
}
```

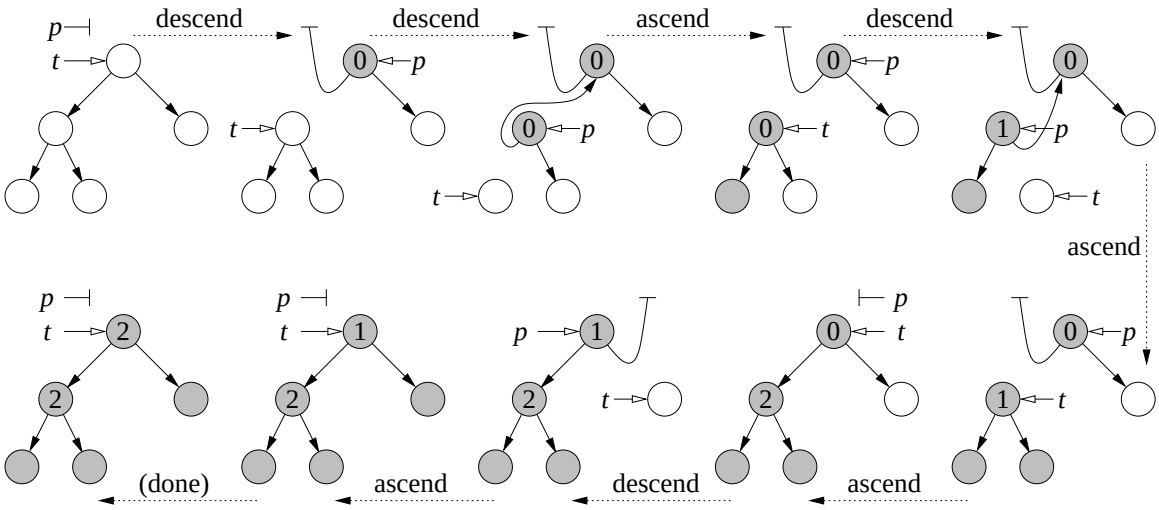



Fig. 148: Marking using link redirection.

As we have described it, this method requires that we reserve enough spare bits in each object to be able to keep track of which child we are visiting in order to store the `t.currChild` value. If an object has $k$ children, then $\lceil \lg k \rceil$ bits would be needed (in addition to the mark bit). Since most objects have a fairly small number of pointers, this is a small number of bits. Since we only need to use these bits for the elements along the current search path, rather than storing them in the object, they could instead be packed together in the form of a stack. Because we only need a few bits per object, this stack would require much less storage than the one needed for the recursive version of mark.

**Stop-and-Copy:** The alternative strategy to mark-and-sweep is called *stop-and-copy*. This method achieves much lower memory allocation, but provides for very efficient allocation. Stop-and-copy divides memory into two large *banks* of equal size. One of these banks is *active* and contains all the allocated blocks, and the other is entirely unused, or *dormant*. Rather than maintaining an available space list, the allocated cells are packed contiguously, one after the other at the fron of the current bank. When the current bank is full, we *stop* and determine which blocks in the current bank are reachable or *alive*. For each such live block we find, we *copy* it from the active bank to the dormant bank (and mark it so it is not copied again). The copying process packs these live blocks one after next, so that memory is compacted in the process, and hence there is no fragmentation. Once all the reachable blocks have been copied, the roles of the two banks are swapped, and then control is returned to the program.

Because storage is always compacted, there is no need to maintain an available space list. Free space consists of one large contiguous chunk in the current bank. Another nice feature of this method is that it only accesses reachable blocks, that is, it does not need to touch the garbage. (In mark-and-sweep the sweeping process needs to run through all of memory.) However, one shortcoming of the method is that only half of the available memory is usable at any given time, and hence memory utilization cannot exceed one half.

The trickiest issue in stop-and-copy is how to deal with pointers. When a block is copied from one bank to the other, there may be pointers to this object, which would need to be redirected. In order to handle this, whenever a block is copied, we store a *forwarding link* in the first word of the old block, which points to the new location of the block. Then, whenever we detect a pointer in the current object that is pointing into the bank of memory that is about to become dormant, we redirect this link by accessing the forwarding link. An example is shown in the figure below. The forwarding links are shown as broken lines.

Which is better, mark-and-sweep or stop-and-copy? There is no consensus as to which is best in all circumstances. The stop-and-copy method seems to be popular in systems where it is easy to determine which words are pointers and which are not (as in Java). In languages such as C++ where a pointer can be cast to an integer and then back to a pointer, it may be very hard to determine what really is a pointer and what is not, and so mark-and-sweep is more popular here. Stop-and-copy suffers from lots of data movement. To save time needed for copying long-lived objects (say those that have survived 3 or more garbage collections), we may declare them to be *immortal* and assign them to a special area of memory that is never garbage collected (unless we are really in dire need of space).

## Supplemental Lecture 7: Range Trees

**Range Queries:** The objective of *range searching* is to count or report the set of points of some point set that lie within a given shape. The most well-known instance of range searching

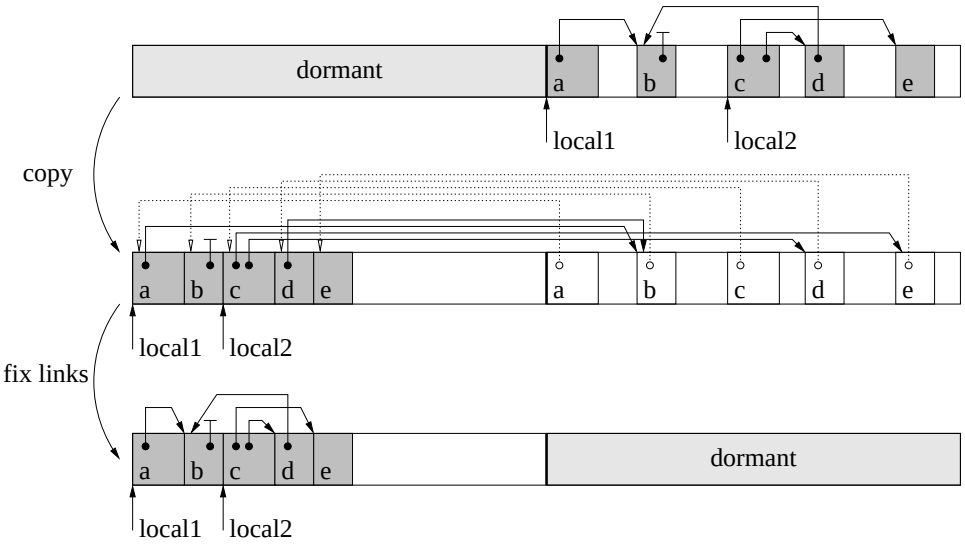



Fig. 149: Stop-and-copy and pointer redirection.

is *orthogonal range searching*, where the shape is an axis-aligned rectangle. In our previous lecture, we discussed the use of kd-trees for answering orthogonal range queries, where we showed that if the tree is balanced, then the running time is close to $O(\sqrt{n})$ to count the points in the range. Generally, if there are $k$ points in the range, then we can report them in total time $O(k + \sqrt{n})$. (The modification is that whenever we find a point or subtree that lies within the range, we traverse the subtree and add all it points to the output.)

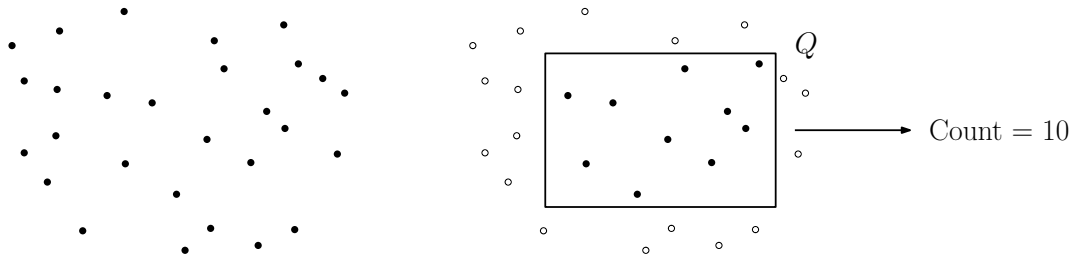



Fig. 150: 2-Dimensional orthogonal range counting query.

**Range Tree Overview:** In this lecture we will present a faster solution for answering these queries. We will consider the problem in 2-dimensional space, but generalizations to higher dimensions are straightforward. The data structures is called a *range tree*. Unlike kd-trees, which are general purpose and can be applied to many different types of queries, range trees are designed especially for orthogonal range queries. We will show that a range tree can answer orthogonal counting range queries in the plane in time $O(\log^2 n)$. (Recall that $\log^2 n$ means $(\log n)^2$). If there are $k$ points in the range it can also report these points in $O(k + \log^2 n)$ time. It uses $O(n \log n)$ space, which is somewhat larger than the $O(n)$ space used by kd-trees. (There are actually two versions of range trees. We will present the simpler version. There is a significantly more complex version that can answer queries in $O(k + \log n)$ time, thus shaving off a log factor in the running time.) Range trees can be generalized to higher dimensions. In any constant dimension $d$, it answers orthogonal range queries in $O(\log^d n)$ time.

**Layering:** Range trees are of additional interest because they illustrate a more general technique used in data structure design, called *layering*. Suppose that you want to design a data structure that answers queries based on multiple criteria, all of which must be satisfied. (For example, find all medical records where the patient was between ages $a_{\text{lo}}$ and $a_{\text{hi}}$, whose weight is between $w_{\text{lo}}$ and $w_{\text{hi}}$, and whose blood pressure is between $b_{\text{lo}}$ and $b_{\text{hi}}$.) Suppose we had data structures for answering each type of query individually. How can we use these to build a data structure for answering finding the entries that satisfy all three. This can be done by "layering" the individual data structures. (Which we will describe below.) This is exactly how range trees work—they layer multiple 1-dimensional range trees to answer multi-dimensional range queries. These are called *multi-layer search trees*.

**Extended Binary Search Trees:** Recall that an binary tree can be *extended* by replacing each null pointer with an new type of node, called an *external node*. The original nodes are called *internal nodes*. In such a tree, every internal node has exactly two children. Extended trees are often used when implementing binary search trees.

When extended trees are used in the context of binary search trees, the two types of nodes have distinct functions.

**Internal:** Internal nodes each store a key (no value). They are used to direct the search towards the external nodes that actually store the data. The convention we will use is that if an internal node stores a key $x$, then the left subtree contains keys strictly less than $x$ and the right subtree stores keys that are greater than or equal to $x$ (see Fig. 151(a)).

**External:** External nodes contain the actual data (keys and values) that constitute the dictionary's contents (see the square nodes in Fig. 151(b)).

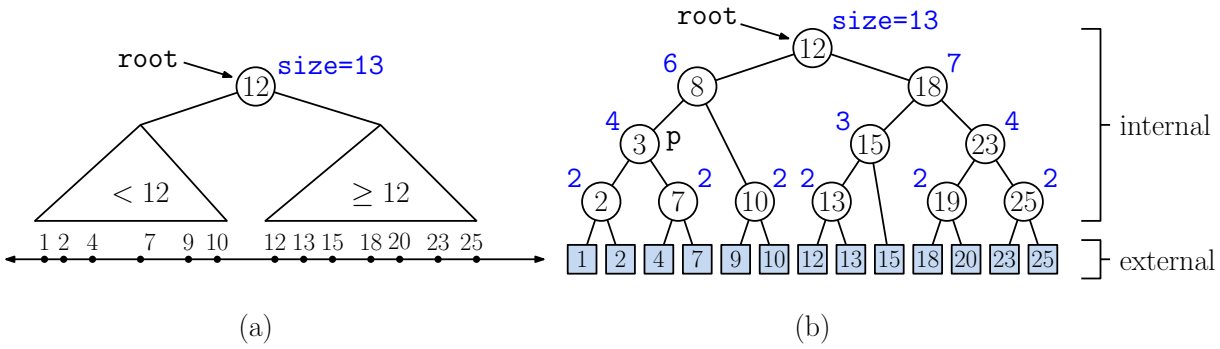



Fig. 151: Extended binary search tree (with sizes).

It is important to note that the *key values stored in the internal nodes are not part of the dictionary*. They are merely guideposts to lead us to the external nodes where the data is stored. For example, in Fig. 151(b), an internal node stores the key 8, but no leaf has this key. Therefore, 8 is *not* in the dictionary. The operation `find(8)` would return `null`.

Why are extended binary search trees useful? In practice, keys are small (e.g., a student ID number) and values are large (e.g., your major, year, college, transcript, and financial records). By storing only the keys, we keep the internal nodes small, and much of the tree can be maintained resident in memory. The values themselves take up much more space and can be stored on disk. This way, the search is fast, and we need only pay the heavy price of a disk access when retrieving the actual data item we want.

Why would we store keys like 8 in internal nodes when there is no external node with this key? This happens as a result of deletions. As we insert new data items, we need to create new internal nodes, and naturally we use the keys themselves when labeling the internal nodes. When an item is deleted from the tree, the associated external node is deleted, but the internal node with this key may remain.

When performing range counting, we want to store size information with each internal node. For each node `p`, we let `p.size` denote the number of *external nodes* descended from `p` (see Fig. 151(b)). (These nodes represent actual data items in the dictionary. We don't really care about the number of internal nodes. But, as observed in an earlier lecture, the number of internal nodes is always one less than the number of external nodes.)

**1-dimensional Range Tree:** Before discussing 2-dimensional range trees, let us first consider what a 1-dimensional range tree would look like. Given a set $S$ of scalar (say, real-valued) keys, we wish to preprocess these points so that given a 1-dimensional interval $Q = [\text{lo}, \text{hi}]$ along the $x$-axis, we can count (or report) all the points that lie in this interval (see Fig. 152). There are a number of simple solutions to this, but we will consider a tree-based method, because it readily generalizes to higher dimensions.

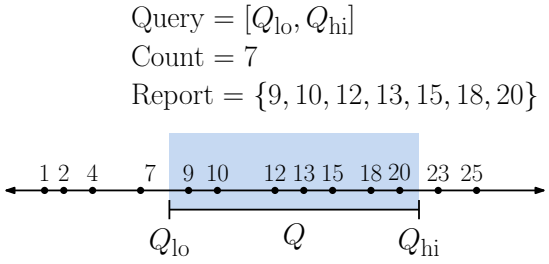



Fig. 152: 1-dimensional range query.

Let us begin by storing all the points of our data set in the external nodes (leaves) of any balanced binary search tree sorted by $x$-coordinates. Each node $p$ in this tree is implicitly associated with a subset $S(p) \subseteq S$ of elements of $S$ that are in the leaves descended from $p$. (For example $S(\text{root}) = S$, and for the internal node `p` labeled 3 in Fig. 151(b), $S(p) = \{1, 2, 4, 7\}$.) Observe that `p.size` is equal to the size of $S(p)$.

**Relevant and Canonical Nodes:** Let us introduce a few definitions before continuing. Given the interval $Q = [Q_{\text{lo}}, Q_{\text{hi}}]$, we say that a node $p$ is *relevant* to the query if $S(p) \subseteq Q$. That is, all the descendents of $p$ lie within the interval. If $p$ is relevant, then clearly all of the nodes descended from $p$ are also relevant.

A relevant node $p$ is *canonical* if $p$ is relevant, but its parent it not. The canonical nodes are the roots of the maximal subtrees that are contained within $Q$ (see Fig. 153(a)).

For each canonical node $p$, the subset $S(p)$ is called a *canonical subset*. Because of the hierarchical structure of the tree, it is easy to see that the canonical subsets are disjoint from each other, and they cover the interval $Q$. In other words, the subset of points of $S$ lying within the interval $Q$ is equal to the disjoint union of the canonical subsets. Thus, solving a range counting query reduces to finding the canonical nodes for the query range, and returning the sum of their sizes.

**Finding the Canonical Nodes:** We claim that the canonical subsets corresponding to any range can be identified in $O(\log n)$ time from this structure. Intuitively, given any interval $[Q_{\text{lo}}, Q_{\text{hi}}]$,

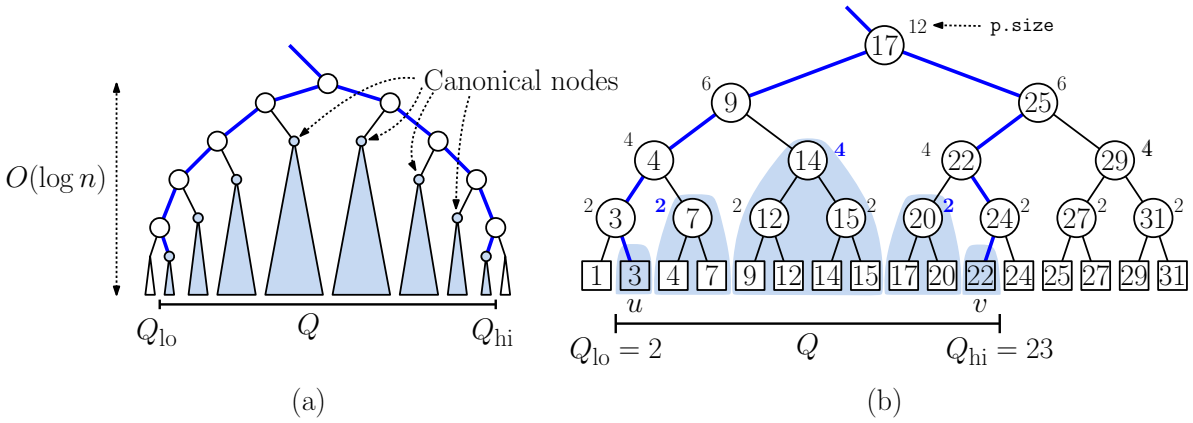



Fig. 153: The canonical subtrees for 1-dimensional range query.

we search the tree to find the leftmost leaf $u$ whose key is greater than or equal to $Q_{\text{lo}}$ and the rightmost leaf $v$ whose key is less than or equal to $Q_{\text{hi}}$. Clearly all the leaves between $u$ and $v$ (including $u$ and $v$) constitute the points that lie within the range. Since these two paths are of length at most $O(\log n)$, there are at most $O(2 \log n)$ such trees possible, which is $O(\log n)$. To form the canonical subsets, we take the subsets of all the *maximal subtrees* lying between $u$ and $v$. In Fig. 153(b), we illustrate this for the interval $[2, 23]$

There are a few different ways to map this intuition into an algorithm. Our approach will be modeled after the approach used for range searching in kd-trees. We will maintain for each node a *cell* $C$, which in this 1-dimensional case is just an interval $[C_{\text{lo}}, C_{\text{hi}}]$. As with kd-trees, the cell for node $p$ contains all the points in $S(T)$.

The arguments to the procedure are the current node, the range $Q$, and the current cell. Let $C_0 = [-\infty, +\infty]$ be the initial cell for the root node (or generally, any interval large enough to contain all the points). The initial call is `range1D(root, Q, C0)`.

Let $p.x$ denote the key associated with the current node $p$, and let $C = [x_0, x_1]$ denote the current cell for node $p$. We assume that given two ranges $A$ and $B$, we have utility functions `A.contains(B)` which determined whether interval $A$ contains interval $B$, and there is a similar function `A.contains(x)` that determines whether $A$ contains a single point $x$.

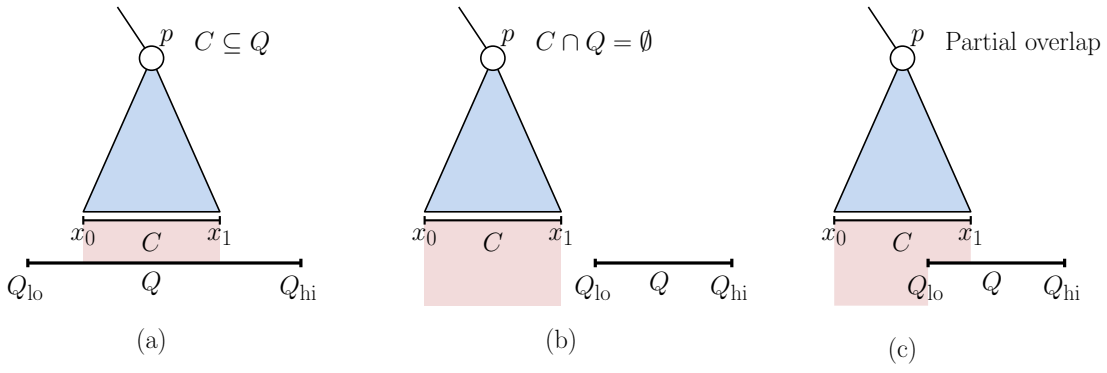



Fig. 154: Range search cases: (a) $Q$ contains $p$'s cell, (b) $Q$ is disjoint from $p$'s cell, (c) $Q$ partially overlaps $p$'s cell.

The processing considers the following cases. If we arrive at an external node, we check

whether the point stored in this external node is contained in $Q$, and if so, we we count it. Otherwise we are at an internal node. First, if $Q$ contains the entire cell $C$, then all the points of this subtree lie within $Q$ and we add `p.size` to the count (see Fig. 154(a)). Next, if $Q$ is entirely disjoint from $C$ (that is, $Q \cap C = \emptyset$), then none of $p$'s descendants can contribute to the query and we return 0. Otherwise, $Q$ and $C$ partially overlap, and we recursively invoke the search procedure on each subtree (see Fig. 154(c)). When we invoke the procedure on the left subtree, we trim the cell to the left part $[x_0, p.x]$ and when we invoke the procedure on the right subtree, we trim the cell to the right part $[p.x, x_1]$. The recursive helper function `range1D(p, Q, C)` is shown in the code block below.

1-Dimensional Range Counting Query

```
int range1Dx(Node p, Range Q, Interval C=[x0,x1]) {
    if (p.isExternal)                              // hit the leaf level?
        return (Q.contains(p.point) ? 1 : 0)       // count if point in range
    else if (Q.contains(C))                        // Q contains entire cell?
        return p.size                              // return entire subtree size
    else if (Q.isDisjointFrom(C))                  // no overlap
        return 0
    else
        return range1Dx(p.left,  Q, [x0, p.x]) + // count left side
               range1Dx(p.right, Q, [p.x, x1])   // and right side
}
```

The external nodes counted in the second line and the internal nodes for which we return `p.size` are the canonical nodes. (To extend this from counting to to reporting, we simply replace the step that counts the points in the subtree with a procedure that traverses the subtree and prints the data in the leaves. Each tree can be traversed in time proportional to the number of leaves in each subtree.) Combining the observations of this section we have the following results.

**Lemma:** Given a (balanced) 1-dimensional range tree and any query range $Q$, in $O(\log n)$ time we can compute a set of $O(\log n)$ canonical nodes $p$, such that the answer to the query is the disjoint union of the associated canonical subsets $S(p)$.

**Theorem:** 1-dimensional range counting queries can be answered in $O(\log n)$ time and range reporting queries can be answered in $O(k + \log n)$ time, where $k$ is the number of values reported.

**Range Trees:** Now let us consider how to answer range queries in 2-dimensional space. We first create 1-dimensional tree $T$ as described in the previous section sorted by the $x$-coordinate (see the left side of Fig. 155). For each internal node $p$ of $T$, recall that $S(p)$ denotes the points associated with the leaves descended from $p$. For each node $p$ of this tree we build a 1-dimensional range tree for the points of $S(p)$, but sorted on $y$-coordinates (see the right side of Fig. 155). This called the *auxiliary tree* associated with $p$. Thus, there are $n - 1$ auxiliary trees, one for each internal node of $T$.

Notice that there is a significant amount of duplication here. Each point in a leaf of the $x$-range tree arises in the sets $S(p)$ for all of its ancestors $p$ in the $x$-range tree. Since the tree is assumed to be balanced, it has height $O(\log n)$, and therefore, each of the $n$ points appears in $O(\log n)$ auxilliary trees. Thus, the sum of the sizes of all the auxiliary trees is $O(n \log n)$.

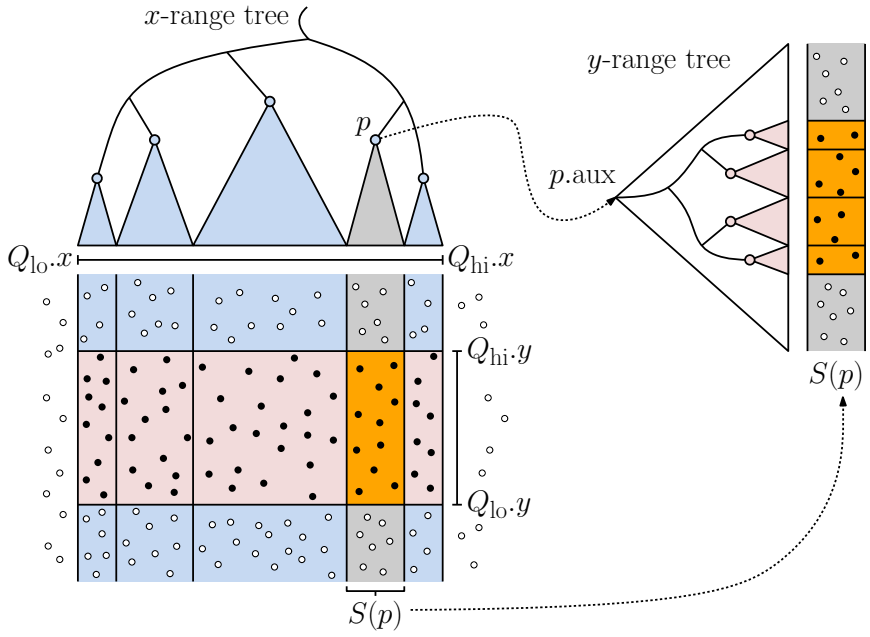



Fig. 155: 2-Dimensional Range tree.

The original tree itself contributes space of $O(n)$. Thus, we have the following total space for the 2-dimensional range tree.

**Claim:** The total size of an 2-dimensional range tree storing $n$ keys is $O(n \log n)$.

Now, when a 2-dimensional range is presented we do the following. First, we invoke a variant of the 1-dimensional range search algorithm to identify the $O(\log n)$ canonical nodes. (These are shown in blue in the left side of Fig. 155.) For each such node $p$, we know that all the points of the set lie within the $x$ portion of the range, but not necessarily in the $y$ part of the range. So, for each of the nodes $p$ of the canonical subtrees, we search the associated 1-dimensional auxiliary $y$-range and return a count of the resulting points. These counts are summed up over all the auxiliary subtrees to obtain the final answer.

The algorithm given in the code block below is almost identical the previous one, except that we make explicit reference to the $x$-coordinates in the search, and rather than adding `p.size` to the count, we invoke a 1-dimensional version of the above procedure using the $y$-coordinate instead. Let $Q.x$ denote the $x$-portion of $Q$'s range, consisting of the interval $[Q_{\text{lo}}.x, Q_{\text{hi}}.x]$. The function call `Q.contains(p.point)` is applied on both coordinates, but the call `Q.x.contains(C)` only checks the $x$-part of $Q$'s range. The procedure `range1Dy()` is the same procedure described above, except that it searches on $y$ rather than $x$.

**Analysis:** It takes $O(\log n)$ time to identify the canonical nodes in the $x$-range tree. For each of these $O(\log n)$ nodes we make a call to a 1-dimensional $y$-range tree. When we invoke this on the subtree rooted at a node $p$, the running time is $O(\log |S(p)|)$. But, $|S(p)| \leq n$, so this takes $O(\log n)$ time for each auxiliary tree search. Since we are performing $O(\log n)$ searches, each taking $O(\log n)$ time, the total search time is $O(\log^2 n)$. As above, we can replace the counting code with code in `range1Dy()` with code that traverses the tree and reports the points. This results in a total time of $O(k + \log^2 n)$, assuming $k$ points are reported.

```
int range2D(Node p, Range2D Q, Range1D C=[x0,x1]) {
    if (p.isExternal)                              // hit the leaf level?
        return (Q.contains(p.point) ? 1 : 0)       // count if point in range
    else if (Q.x.contains(C)) {                    // Q's x-range contains C
        [y0,y1] = [-infinity, +infinity]           // initial y-cell
        return range1Dy(p.aux.root, Q, [y0, y1])   // search auxiliary tree
    }
    else if (Q.x.isDisjointFrom(C))                // no overlap
        return 0
    else
        return range2D(p.left,  Q, [x0, p.x]) +    // count left side
               range2D(p.right, Q, [p.x, x1])      // and right side
}
```

Thus, each node of the 2-dimensional range tree has a pointer to a auxiliary 1-dimensional range tree. We can extend this to any number of dimensions. At the highest level the $d$-dimensional range tree consists of a 1-dimensional tree based on the first coordinate. Each of these trees has an auxiliary tree which is a $(d-1)$-dimensional range tree, based on the remaining coordinates. A straightforward generalization of the arguments presented here show that the resulting data structure requires $O(n \log^{d-1} n)$ space and can answer queries in $O(\log^d n)$ time.

**Theorem:** Given an $n$-element point set in $d$-dimensional space (for any constant $d$) orthogonal range counting queries can be answered in $O(\log^d n)$ time, and orthogonal range reporting queries can be answered in $O(k+\log^d n)$ time, where $k$ is the number of entries reported. In both cases, the total space of the data structure is $O(n \log^{d-1} n)$.