CMSC 420
Data Structures[1]

David M. Mount
Department of Computer Science
University of Maryland
Fall 2019

[1] Copyright, David M. Mount, 2019, 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: Course Introduction and Background

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 walk-through, 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 operations that are used to store, access, and manipulate these entities and the formal meaning of these operations,

Representation: the manner in which these entities are represented concretely in a computer's memory, and

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 not always 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: *Review material. Please be sure you are familiar 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]

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.

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 operation. 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.

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

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

(This function was just made up as an illustration.) Unless we say otherwise, assume that logarithms are taken base 2. When the value n is small, we do not worry too much about

[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.

this function since it will not be too large, but as n increases in size, we will have to worry about the running time. Observe that as n grows larger, the size of n^3 is much larger than n^2, which is much larger than $n \log n$ (note that $0 < \log n < n$ whenever $n > 1$) which is much larger than $\sqrt{n}$. Thus the n^3 term dominates for large n. Also note that the leading factor 13 is a constant. Such constant factors can be affected by the machine speed, or compiler, so we may ignore it (as long as it is relatively small). We could summarize this function succinctly by saying that the running time grows "roughly on the order of n^3", and this is written notationally as $T(n) = O(n^3)$. Informally, the statement $T(n) = O(n^3)$ means, "when you ignore constant multiplicative factors, and consider the leading (i.e. fastest growing) term, you get n^3". This intuition can be made more formal, however. We refer you back to your algorithms course for how to do this.

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 α 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: Van Emde Boas trees.)

$T(n) = O(\log n)$: 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.)

$T(n) = O(n^p)$: (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

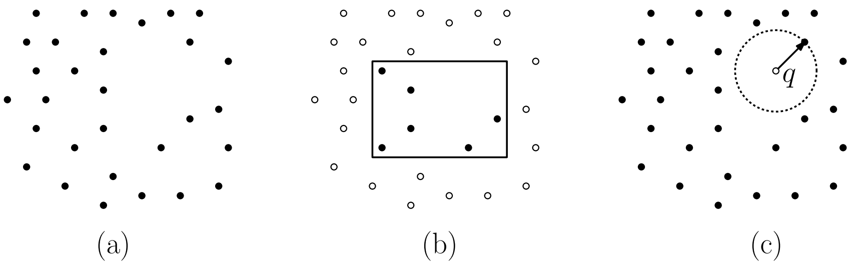


Fig. 1: (a) A point set, (b) orthogonal range search query, and (c) nearest-neighbor query.

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.)

$T(n) = O(n^2)$: *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). This is called *polynomial* time. Practical if k is not too large. (Example: Matrix multiplication.)

$T(n) = O(2^n), O(n^n), O(n!)$: *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

Note: In addition to the material presented here, in class we also presented and discussed a Fun Challenge Problem. Please check out the PowerPoint version of the slides for information about this. This material will *not* be covered on exams, but it might be the topic of a homework assignment.

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 ith 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 ith 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)?

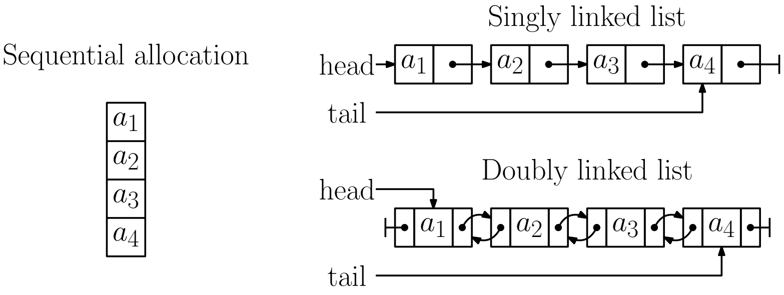


Fig. 2: Common types of list allocation.

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*.

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*.

Deque: This data structure is a combination 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.

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 8 elements, then when an attempt is made to insert a 9th element, we allocate an array of size 16, copy the existing 8 elements to this new array, and then add the new element. 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.

You might wonder, why do we double the array size? Why not, instead, just allocate an array with 100 additional elements? Alternatively, why not be more aggressive and square the size of the array (jumping from 8 to 64 elements)?

If you have no additional knowledge regarding the access sequence, there is a good reason why increasing the size by a constant factor is the "right" thing to do. (Doubling itself is not essential. You could increase the size by another factor, such as 1.5 or 3.0, but the increase should be by a constant factor.)

This reason is related to the notion of *amortization*, which we introduced in the previous lecture. Remember that amortization means that, rather than reporting the cost of single operation, the cost of maintaining the data structure is averaged over a long sequence of operations. In our case, many operations are very "cheap", involving just a adding or removing one element from the end of the array. But, when reallocation is performed, the cost of that one operation is very high. But in order to get to a reallocation, we must have performed a significant number of "cheap" insertions, where no reallocation was needed. Thus, the average cost is low.

For example, in Fig. 3, we have a stack array with capacity of 8, which initially contains 4 elements. The next four `push` operations are all cheap, each taking just a constant amount of time, but the fifth `push` causes us to allocate an array of size 16. The total work for this last operation is 17 (16 to pay for the allocation of the new array, and 1 more for the actual `push` operation). In an amortized analysis, we will "pretend" that each of the cheap operations takes 5 time units, rather than the actual 1 unit. We'll call this the *adjusted cost.* This is just an accounting trick. We save the additional 4 units, which we call *work tokens* in a bank account. Note that when we add the final element, we have enough in our bank account to pay for the reallocation. Thus, we can say that the *amortized cost* of each operation is 5 units, that is, a constant per operation. (You will notice that the sum of the amortized costs of 25 in our figure is a bit higher than the sum of the actual costs of 21. This is because we will need the extra 4 work tokens to pay for the next reallocation, when we jump from 16 to 32.)

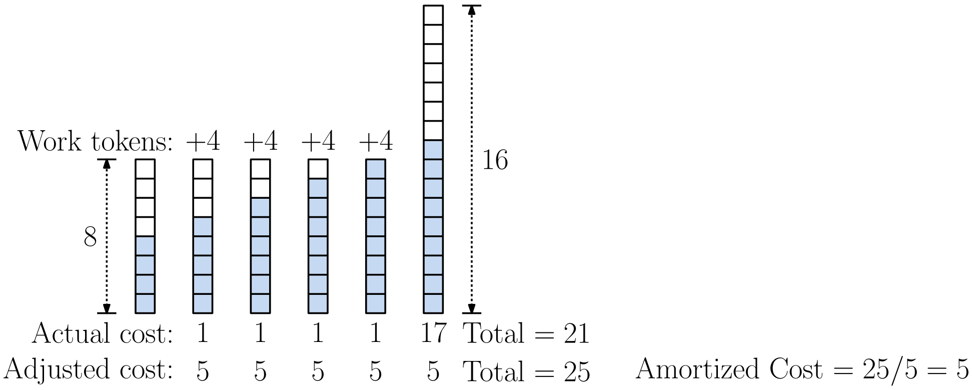

Work tokens: +4 +4 +4 +4
8
16
Actual cost: 1 1 1 1 17 Total = 21
Adjusted cost: 5 5 5 5 5 Total = 25
Amortized Cost = 25/5 = 5

Fig. 3: Doubling reallocation.

The following theorem makes this intuition formal.

Theorem: When doubling reallocation is used for stack/queue/deque operations, the amortized cost of each operation is $O(1)$.

Proof: Let us do the proof for stacks, since the generalization to the other structures is straightforward. Let us also assume that the initial allocation is of constant size (e.g., we always start with capacity for 8 elements). The initialization takes $O(1)$ time.

We will use a charging argument to show that the amortized cost is constant per operation. In particular, we will "amortize" the cost of reallocation among the `push` operations that came just before it.

Let n denote the current size of the array allocated for the stack. Each time we do a `push` operation, we perform the operation and put 4 *work tokens* in a bank account. The operation itself takes only a constant time, and the 4 work tokens will be saved up for later. Now, suppose that the latest `push` operation causes us to run out of space in the array. We allocate a new array of size $2n$, which must be initialized and elements copied to it. Let's say that the actual cost of performing this work is $2n$. We want to pay for this work from our bank account. Have we accumulated enough funds to do so? Well, the last time we reallocated we went from an array of size $n/2$ to an array of size n. In order to overflow this array, we must have performed at least $n - n/2 = n/2$ additional `push` operations. Since each `push` allows us to place 4 tokens in our bank account, we have accumulated at least $4(n/2) = 2n$ tokens. Thus, we have enough to pay for the cost of reallocation.

Would this work if instead we had added 100 additional elements? The answer is no. If this list was really large (say, millions), the reallocations are expensive (on the order of millions), but the accumulated work tokens would be much smaller (on the order of hundreds). So, our bank account would be way too small to pay for the actual work done (and both our accountant and us would wind up in math prison!)

On the other hand, what if we increased by a much larger amount, say squaring the current array size. The good news is that we would definitely have enough tokens to pay for the reallocation, but this is wasteful. For example, if our final insertion caused our array to go from 1,000 entries to 1,001, the doubling scheme would generate allocate 2,000 entries, while squaring would result in 1,000,000 entries!

As mentioned above, it is not necessary to double. The above proof can be modified to show that any scheme that increases the array size by a constant factor c, where $c > 1$ will achieve an $O(1)$ amortized cost. The bigger you set c, the small the amortized cost (but the more space is wasted).

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.

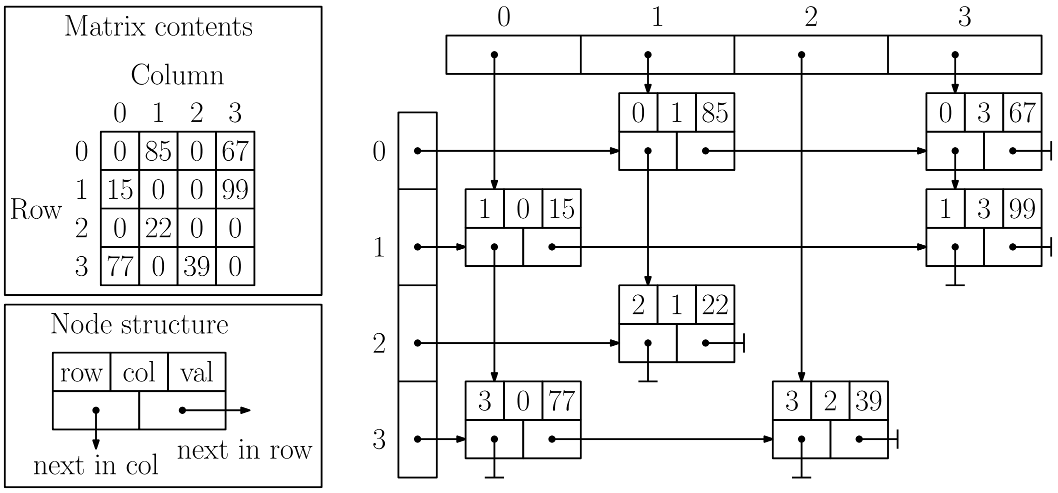
The physicist knows that most stars are so far apart from each other that (due to the inverse square law of gravity), only a small number of matrices are significant, and all the others could be set to zero. For example, $n = 10,000$ but a star typically exerts a significant gravitational pull on only 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. 4). 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.

Lecture 3: Rooted Trees and Binary Trees

Tree Definition and Notation: Trees and their variants are among the most fundamental data structures. A tree is a special class of graph.[3] The most general form of a tree, called a *free*

[3] Recall from your previous courses that a *graph* $G = (V, E)$ consists of a finite set of *nodes* V and a finite set of edges E. Each *edge* is a pair of nodes. In an *undirected graph*, the edge pairs are unordered, and in a *directed graph*,



Row	0	1	2	3
0	0	85	0	67
1	15	0	0	99
2	0	22	0	0
3	77	0	39	0

Fig. 4: Sparse matrix representation using a multilist structure.

tree, is simply a connected, undirected graph that has no cycles (see Fig. 5(a)). An example of a free tree is the minimum cost spanning tree (MST) of a graph.

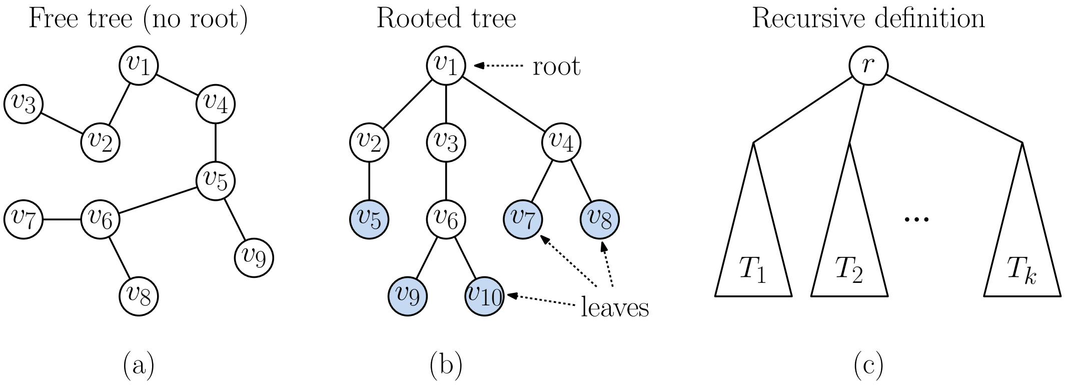


Fig. 5: Trees: (a) free tree, (b) rooted tree, (c) recursive definition.

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. 5(b)). All the others are called *internal nodes*.

A rooted tree can be defined formally as follows. First, a single node is a rooted tree. Second, given any set $\{T_1, \ldots, T_k\}$ of one or more rooted trees, joining these trees together under a common root node r is also a rooted tree (see Fig. 5(c)).

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.

There is a lot of notation involving trees. Most terms are easily understood from the family-tree analogy. Each non-leaf node has one or more *children*, and except for the root, every

the edge pairs are ordered. An undirected graph is *connected* if there is path between any pair of nodes.

node has a single *parent*. The *degree* of a node is the number children it has. Two nodes that share the same parent are *siblings* of each other. Each node of the tree can be viewed as the root of a *subtree*, consisting of this node an all of its descendants. (For example, referring to Fig. 5(b), v_7 and v_8 are the children of v_4. Nodes v_2, v_3, and v_4 are siblings, and they share v_1 as their common parent. Node v_6 has degree 2, and it is the root of a 3-node subtree consisting of v_6, v_9 and v_{10}.)

Although we have not specified a direction to the edges, it is natural to do so for rooted trees. When the edges are directed away from the root, the tree is called an *arborescence* or *out-tree* (see Fig. 6(a)). When they are directed in towards the root, the tree is called an *anti-arborescence* or *in-tree* (see Fig. 6(c)).

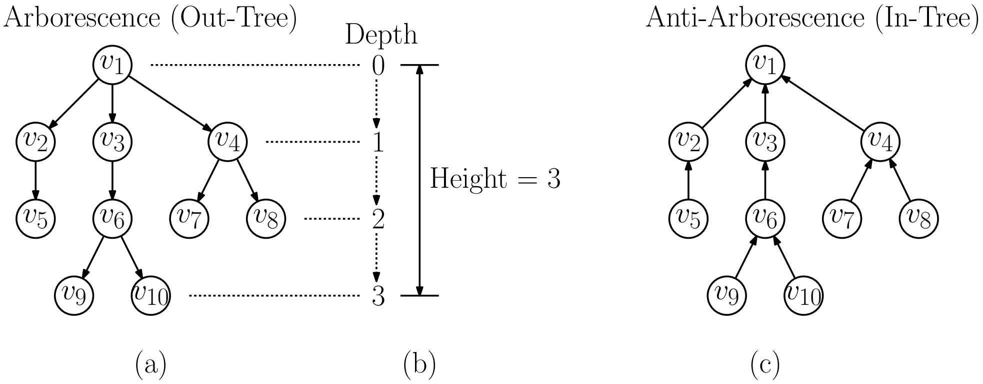


Fig. 6: More notation involving trees.

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. 6(b)). For example, the tree of Fig. 5(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. When the ordering among a node's is significant, it is called an *ordered tree.*

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 out-tree. (In-trees are easier to represent, since each node can just store a single pointer to its parent.) Out-trees (arborescences) are tricky because the number of children a node has is not fixed.

We will present a widely-used representation, which has the feature that all nodes have the same size, irrespective of the number of children the node has. This representation works for ordered trees (where the siblings are ordered), but of course by ignoring the order information it can be applied to unordered trees as well. In addition to storing whatever data about the node that is pertinent to the application, each node stores two references (pointers), one to the node's first child and the other to its next sibling (see Fig. 7(a)). Let us call these `firstChild` and `nextSibling`, respectively. Fig. 7(b) illustrates how the tree in Fig. 6(a) would be represented using this technique. This is minimal representation. In practice, we

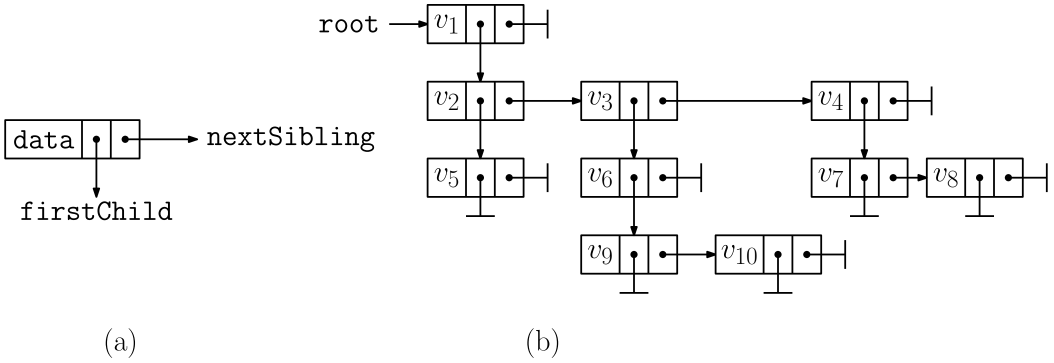


Fig. 7: Standard (binary) representation of rooted trees.

may wish to add additional information. For example, each node could also include a reference to its parent.

It is interesting to observe that this representation is itself a binary tree (defined below).

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. 8(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.)

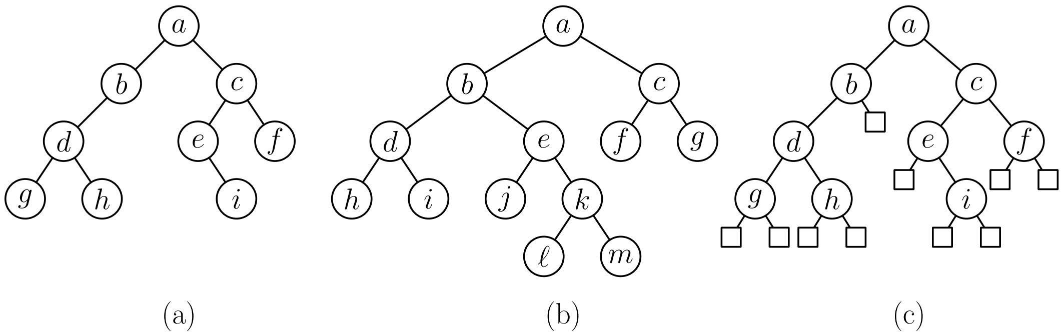


Fig. 8: Binary trees: (a) standard definition, (b) full binary tree, (c) extended binary tree.

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. 8(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. 8(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. 8(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 BinaryTreeNode<E> {
    private E                    entry;      // this node's data
    private BinaryTreeNode<E>    left;       // left child reference
    private BinaryTreeNode<E>    right;      // right child reference
    // ... remaining details omitted
}
```

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

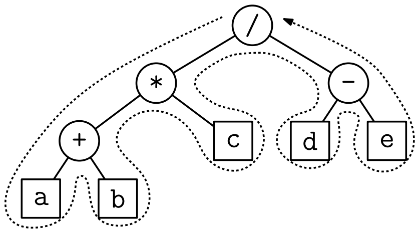
Binary trees come up in many applications. One that we will see a lot of this semester is for representing ordered sets of objects, a *binary search tree*. Another is an *expression tree*, which is used in compiler design in representing a parsed arithmetic exception (see Fig. 9).

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. 9 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. 9: 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(BinaryTreeNode 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. 9), then such a reconstruction is possible. Think about this.

Extended Binary Trees: Let us explore a few basic combinatorial facts regarding extended binary trees. Consider an extended binary tree having n internal nodes. Can we predict how many external nodes there will be? The answer is yes, and the number is $n + 1$. If you draw a few extended trees, you can convince yourself of this. It is also easy to see this by incrementally replacing an arbitrary external node with a triple consisting of an internal node and two external children. Let's provide a formal proof by induction. This sort of induction is so common on binary trees, that it is worth going through this simple proof to see how such proofs work in general.

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

Proof: (by induction on the size of the tree) Let $x(n)$ denote the number of external nodes in a binary tree of n nodes. We want to show that for all $n \geq 0$, $x(n) = n + 1$.

The basis case is trivial. An extended tree with zero internal nodes has a single external node, so $x(0) = 1$, which agrees with our formula.

Now let us consider the case of $n \geq 1$. The induction hypothesis states that, for all $n' < n$, $x(n') = n' + 1$. Let n_L and n_R denote the number of internal 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 external nodes in the left and right subtrees are $x(n_L) = n_L + 1$ and $x(n_R) = n_R + 1$. Putting this together, we find that the total number of external nodes 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. Since there are n internal nodes and $n + 1$ external nodes, the total number is $2n + 1$.

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 `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. 10(a), we show a threaded version of the tree in Fig. 8(b)).

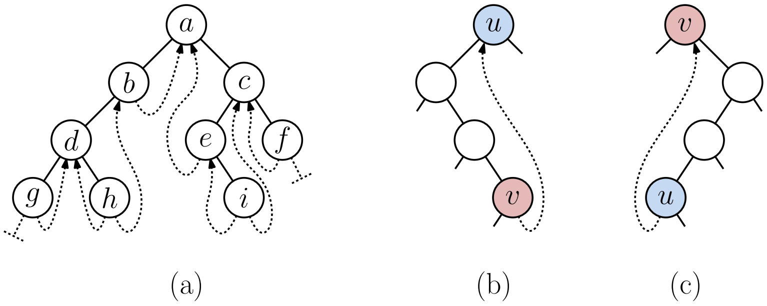


Fig. 10: A Threaded Tree.

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. 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. 10(b)). Otherwise, we go the node's right child, and then traverse left-child links until reaching the bottom of the tree, that is a threaded link (see Fig. 10(c)).

Inorder Successor in a Threaded Tree

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

For example, in Fig. 10(b), if we start at d, the thread takes us directly to a, which is d's inorder successor. In Fig. 10(c), if we start at a, then we follow the right-child link to b, and then follow left-links until arriving at d, which is the inorder successor.

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. 11). (We leave these as exercises involving geometric series.)

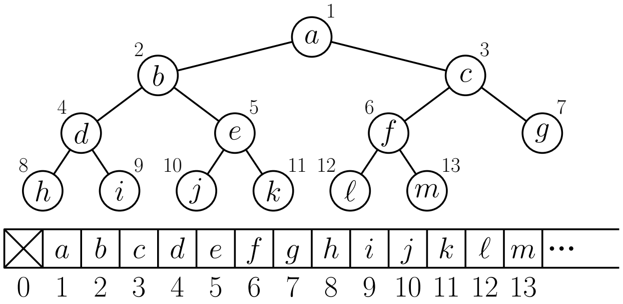


Fig. 11: A complete binary tree.

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 also 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: Binary Search Trees

Searching: Searching is among the most fundamental problems in data structure design. We are storing 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 each key value occurs at most once in the data structure, and 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`. (In a Java implementation, this can be handled by defining a class with two generic types, one for the key and one for the value.) We will also assume that key values can be compared using the usual comparison operators, such as `<`, `==`, `>=`. In actual implementation, it is assumed that the `Key` class supports a function for comparing keys. For example, in Java's various map classes, it is assumed that the `Key` class implements the `Comparator` interface. This means that there is a function `compare`(x_1, x_2), which returns a negative integer, zero, or a positive integer depending on whether the x_1 is less than, equal to, or greater than x_2, respectively.

The Dictionary ADT: Perhaps the most basic example of a search data structure is the dictionary. A *dictionary* is an ADT that supports the operations of insertion, deletion, and finding. There are a number of additional operations that one may like to have supported, but these are the core operations.

`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.

Other operations that might like to see in a dictionary include iterating the entries, answering range queries (that is, reporting or counting all objects between some minimum and maximum key values), returning the kth smallest key value, and computing set operations such as union and intersection.

There are three common methods for storing dictionaries: sorted arrays, hash tables, and binary search trees. We discuss two of these below. Hash tables 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 $O(n)$ 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 $n = 1,000,000$, $\log_2 n$ 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, such that an inorder traversal visits the nodes in increasing key order. In particular, if x is the key stored in the root 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. 12(a)). (Recall that we assume that keys are distinct, so no other key can be equal to x.)

Defining such an object in an object-oriented language like Java typically involves two class definitions. The main class is for the dictionary itself, and the other is for the individual

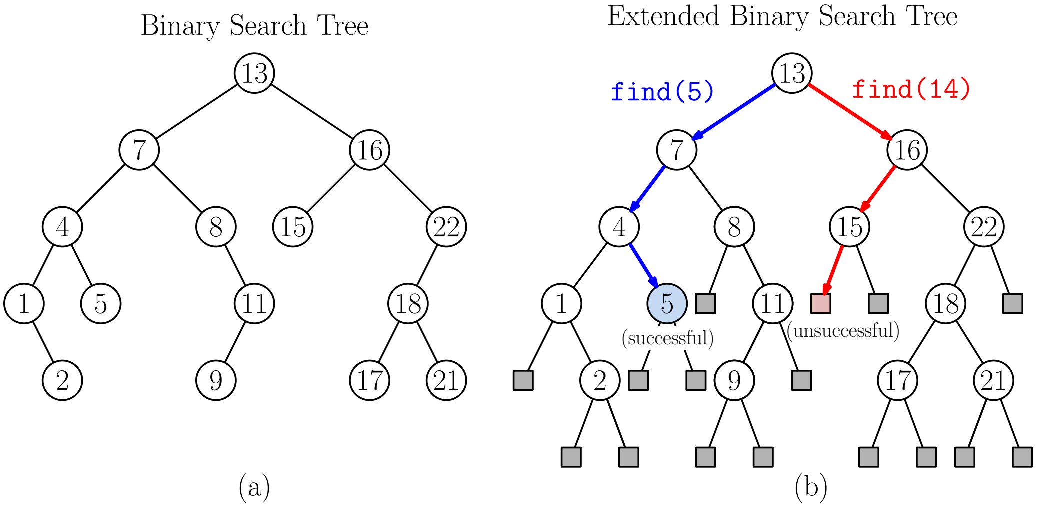


Fig. 12: Binary Search Tree.

nodes of the tree. We will call these `BinarySearchTree` and `BinaryNode`, respectively. The `BinarySearchTree` class has all the public functions and stores a reference to the root node of the tree. But most of the hard work is done by the methods associated with the `BinaryNode` class.

A node in the binary search tree would typically store the key, the value, and left and right pointers. It may also store additional information, such as parent pointers. When presenting our code examples, we will not be concerned with the value component, and will just focus on the key and the other pointers.

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 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. 12(b)) or we fall out of the tree (see Fig. 12(b)).

Note that if we think of the tree as an *extended tree*, then an unsuccessful search terminates at an external node. Each external node represents the unsuccessful searches for keys that lie between its inorder predecessor and inorder successor. (For example, in Fig. 12(b), the external node where the search for 14 ends represents all the searches for keys that are larger than 13 and smaller than 15.) One argument in favor of using extended trees is that the external node provides this additional information (as opposed to a simple `null` pointer).

A natural way to handle this would be to make the search procedure a recursive member function of the `BinaryNode` class. The initial call is made from the `find()` method associated with the `BinarySearchTree` class, which invokes `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. While most tree-based algorithms are best expressed recursively, this one is easy enough to do iteratively, and shown in the following code block. If you really wanted the best in performance, you would likely prefer this iterative form.

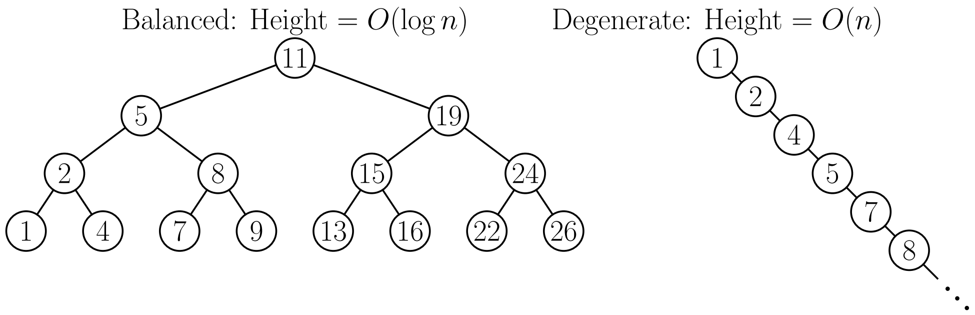
What is the running time of the search algorithm? Well, it depends on the key you are

Recursive Binary Tree Search

```
Value find(Key x, BinaryNode 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
}
```

Iterative Binary Tree Search

```
Value find(Key x) {
    BinaryNode p = root;                  // start at the root
    while (p != null) {                   // proceed until we fall out of the tree
        if (x < p.key) p = p.left;        // x is smaller? ...search left
        else if (x > p.key) p = p.right;  // x is larger? ...search right
        else return p.value;              // successful search
    }
    return null;                          // unsuccessful search
}
```


Balanced: Height $= O(\log n)$

Degenerate: Height $= O(n)$

Fig. 13: Balanced and degenerate binary trees.

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. 13 right) or as large as $O(n)$ for the case of a *degenerate tree* (see Fig. 13 left). However, we shall see that if the keys are inserted in random order, the expected height of the tree is just $O(\log n)$.

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. 14, we fall out of the tree at the left child of node 15, and we insert the new node with key 14 here.)

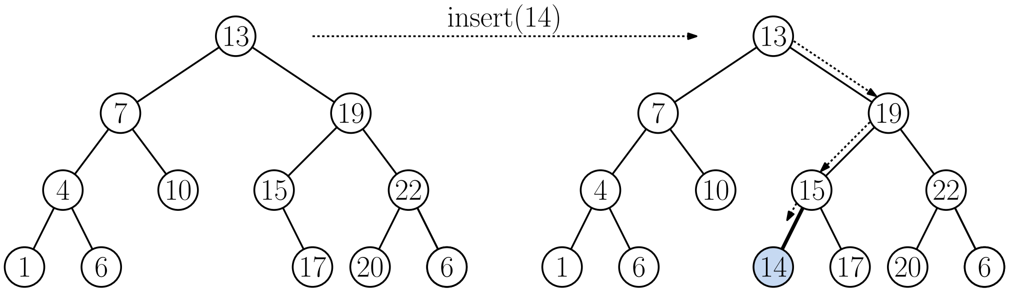


Fig. 14: Binary tree insertion.

The insertion procedure is shown in the code fragment below. There is one technical difficulty with implementing this in Java (or generally, any language that uses pass-by-value in function calls). When we create the new node, we want to "reach up" and modify one of the pointer fields 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. There are a number of ways to fix this issue (including coding the procedure iteratively or explicitly passing in a reference to the parent node). Instead, we will employ a coding trick to get around this. In particular, the insertion function will return a reference to the modified subtree after insertion, and we store this value in the appropriate child pointer for the parent.

The initial call from the `BinarySearchTree` class is `root = insert(x, v, root)`. We assume that there is a constructor for the `BinaryNode`, which is given the key, value, and the initial values of the left and right child pointers.

Recursive Binary Tree Insertion

```
BinaryNode insert(Key x, Value v, BinaryNode p) {
    if (p == null)                           // fell out of the tree?
        p = new BinaryNode(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!)
}
```

A Closer Look at the Trick: To better understand how our coding trick works, see Fig. 15 to insert 14. We first search for 14 in the tree, falling out of the tree at the left child of node 15, that is, when the local variable p refers to node 15. Let's call this local variable p_2. We generate a call `p2.left = insert(14, v, p2.left)`.

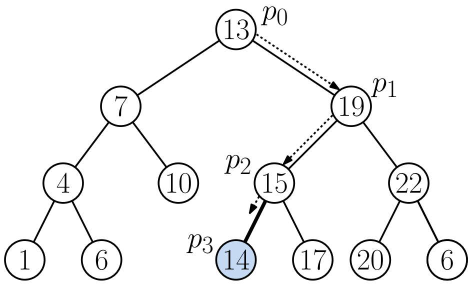


Fig. 15: Child link update in insertion.

Since node 15 has no left child, the next recursive call discovers right away that its local p (which has the value `p2.left`) value is `null`, and so we create the new node with key value 14, and assign it to the current local variable p. Let's call this p_3. The last line of the recursive procedure returns p_3 to the calling procedure at node 15, at the statement `p2.right = insert(14, v, p2.right)`. Since the `insert` function returns the pointer p_3 to the new node, we effectively perform the action `p2.right = p3`, which links the new node into the tree as desired. Voila!

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. 16(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. 16(b)).

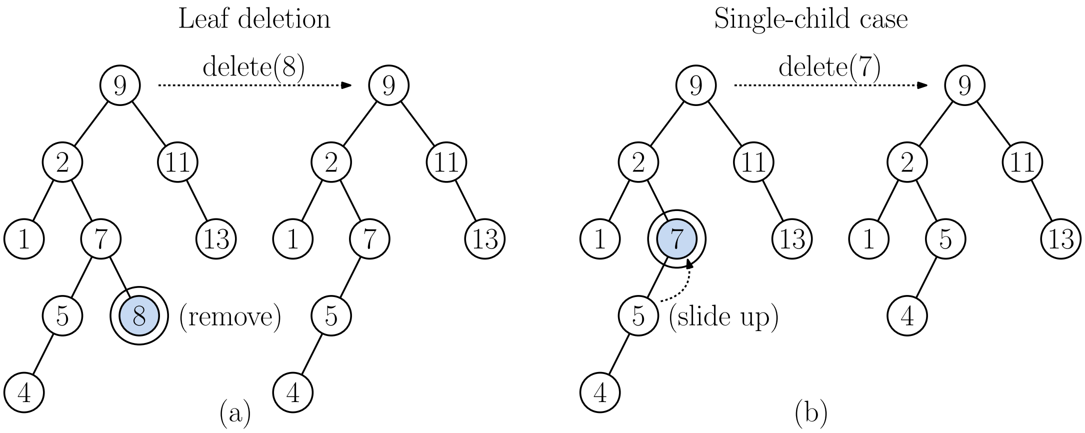


Fig. 16: Deletion: (a) Leaf and (b) single-child case.

The hardest case is when the deleted node that has two children. Let p denote the node to be deleted (see Fig. 17(a)):

- Find the node r that is p's inorder successor in the tree (see Fig. 17(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 to p (see Fig. 17(c)).
- Delete node r (see Fig. 17(d)). (Because r's key immediately follows p's key, this replacement maintains the sorted order of keys.)

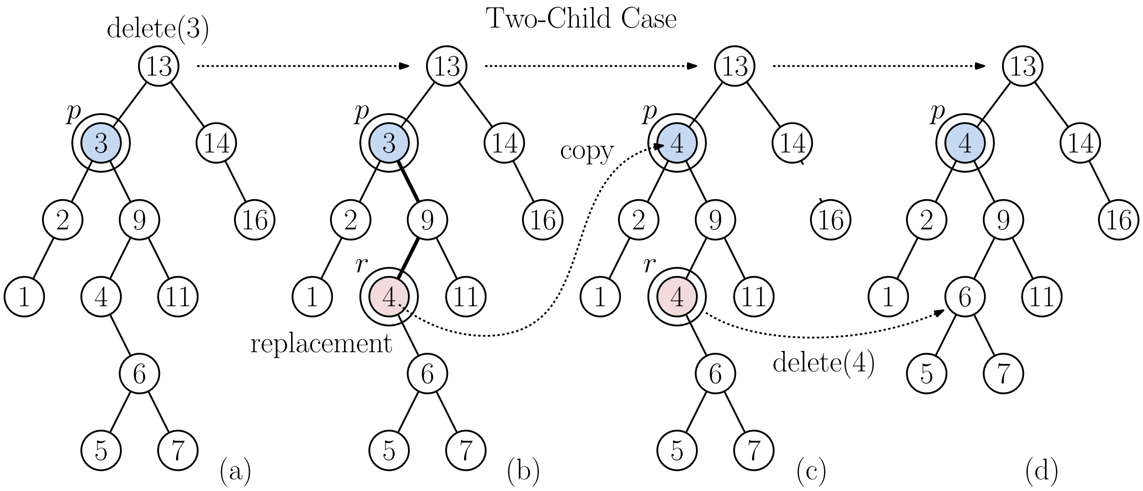


Fig. 17: Deletion: Two-child case.

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.

Replacement Node for the Two-child Case

```
BinaryNode findReplacement(BinaryNode p) {           // find p's replacement node
    BinaryNode r = p.right;                          // start in p's right subtree
    while (r.left != null) r = r.left;               // go to the leftmost node
    return r;
}
```

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

Binary Tree Deletion

```
BinaryNode delete(Key x, BinaryNode 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) return p.right;     // return replacement node
            else                return 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;
}
```

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 not hard to see that in the worst case, if we insert keys in either strictly increasing or strictly decreasing order, then the resulting tree will be completely degenerate, and have height $n - 1$. We will show that, if keys are inserted in random order, then the expected depth of any node is $O(\log n)$. (We emphasize that this assumes insertions only. See below for a discussion of the situation when insertions and deletions are combined.)

Proving that the expected depth is $O(\log n)$ is not an trivial exercise. The proof involves setting up a fairly complicated recurrence and solving it. (If you have ever seen the complete analysis of QuickSort, the two recurrences are very similar.) Instead, we will produce a "quick and dirty" proof, which hopefully will convince you that the assertion is reasonable, if not fully convincing. In particular, rather than 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 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 ith key. The only way that the leftmost node changes is when the ith 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.

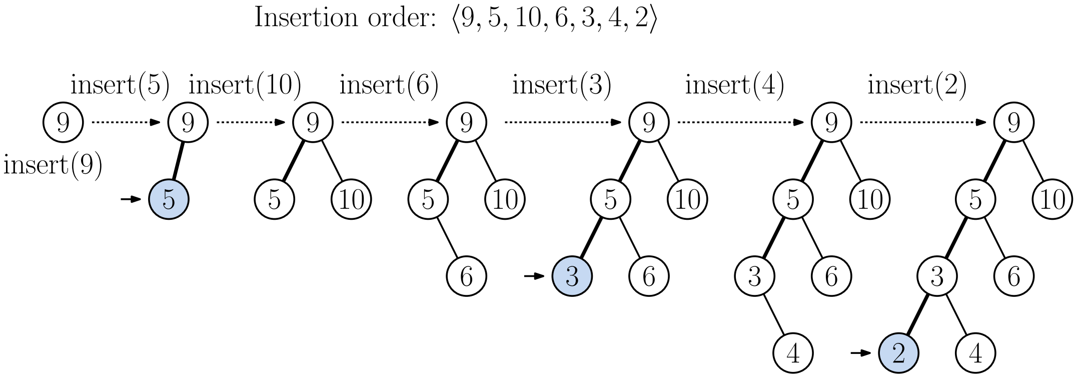


Fig. 18: Length of the leftmost chain.

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 ith 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.)

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 ith. 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, this analysis breaks down if we are doing 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]

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 bias can be eliminated by selecting the replacement node (randomly) as 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).

Lecture 5: 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 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. 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. The most natural relaxation of this condition is expressed in the following invariant:

AVL balance condition: For every node in the tree, the absolute difference in the heights of its left and right subtrees is at most 1.

For any node v of the tree, let height(v) denote the height of the subtree rooted at v (shown in blue in Fig. 19(a)). It will be convenient to define the height of an empty tree (that is, a `null` pointer) to be -1. Define the *balance factor* of v, denoted balance(v) to be

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

[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.

(see Fig. 19(b)). The AVL balance condition is equivalent to the requirement that balance$(v) \in \{-1, 0, +1\}$ for all nodes v of the tree. (Thus, Fig. 19(b) is an AVL tree, but the tree of Fig. 19(c) is not because node 10 has a balance factor of +2.)

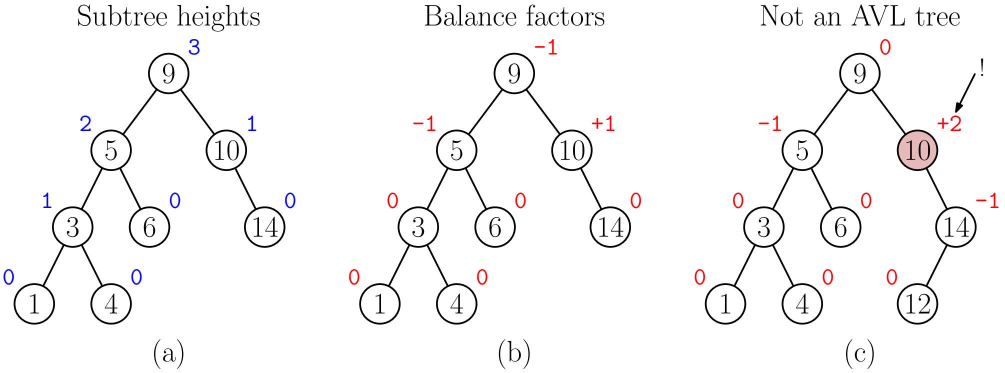


Fig. 19: AVL-tree balance condition.

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 will arise in the analysis. Recall that for $h \geq 0$, the hth *Fibonacci number*, denoted F_h is defined by the following recurrence:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_h &= F_{h-1} + F_{h-2}, \qquad \text{for } h \geq 2. \end{aligned}$$

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*.

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

Proof: Let $N(h)$ denote the minimum number of nodes in any AVL tree of height h. We will generate a recurrence for $N(h)$ as follows. 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 $N(1) = 2$.

For $n \geq 2$, 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 $h_R = h-2$. Counting the root node plus the numbers of nodes in the left and right subtrees we obtain the following recurrence for the total number of nodes:

$$\begin{aligned} N(0) &= 1 \\ N(1) &= 2 \\ N(h) &= 1 + N(h_L) + N(h_R) \ = \ 1 + N(h-1) + N(h-2). \end{aligned}$$

Height:	0	1	2	3	4	5
# Nodes:	1	2	4	7	12	20
# Leaves	1	1	2	3	5	8

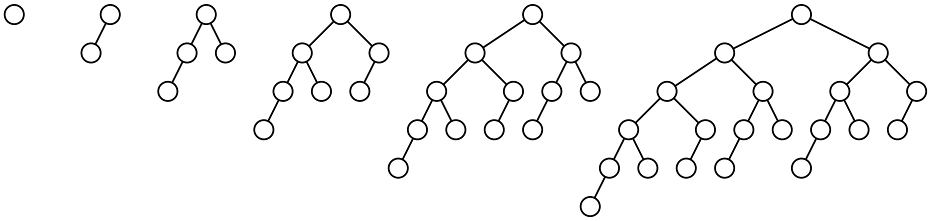


Fig. 20: Most imbalanced AVL trees of various heights.

The resulting trees are shown in Fig. 20. Observe that while the number of nodes does not follow the Fibonacci sequence exactly, the number of leaves does follow the Fibonacci sequence.

While $N(h)$ is not quite the same as the Fibonacci sequence, by an induction argument[5] we can show that for large h, there is a constant c such that

$$N(h) \;\geq\; c\varphi^h, \quad \text{where } \varphi = \left(\frac{1+\sqrt{5}}{2}\right)^h.$$

Theorem: 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 $1.44 \lg n$.)

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 the relationship between two nodes, 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. 21(a) and (b)).

We have intentionally labeled the elements of Fig. 21 to emphasize the fact that the inorder properties of the tree are preserved. That is subtree A comes before node b comes before subtree C, and so on. The code fragment below shows how to apply a right and left rotations to a node p. As has been our practice, we return a pointer to the modified subtree (in order to modify the child link pointing into this subtree).

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.

[5] Here is a sketch of a proof. Let us conjecture that $N(h) \approx \varphi^h$ for some constant φ. Since we are proving a lower bound, there is no harm in ignoring the $+1$ in the recurrence for $N(h)$. Substituting our conjectured value for $N(h)$ into the above recurrence, we find the φ satisfies $\varphi^h = \varphi^{h-1} + \varphi^{h-2}$. Removing the common factor of φ^{h-2}, we have $\varphi^2 = \varphi + 1$, that is, $\varphi^2 - \varphi - 1 = 0$. By applying the quadratic formula, we conclude that $\varphi = (1+\sqrt{5})/2$.

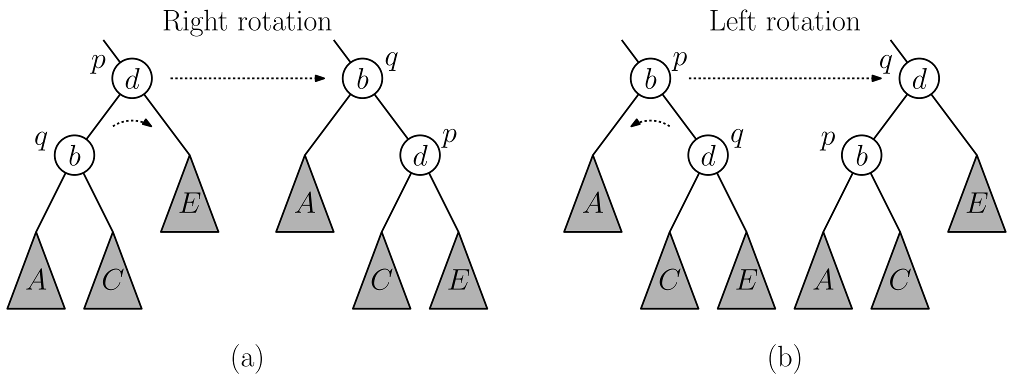


Fig. 21: (Single) Rotations. (Triangles denote subtrees, which may be null.)

Binary-Tree Rotations

```
BinaryNode rotateRight(BinaryNode p) {  // right rotation at p
    BinaryNode q = p.left;
    p.left = q.right;
    q.right = p;
    return q;
}

BinaryNode rotateLeft(BinaryNode p) {   ... symmetrical ... }
```

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. 22). 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. 22(a)) followed by a right rotation to `p` (labeled d in Fig. 22(a)). The right-left rotation is symmetrical (see Fig. 22(b)).

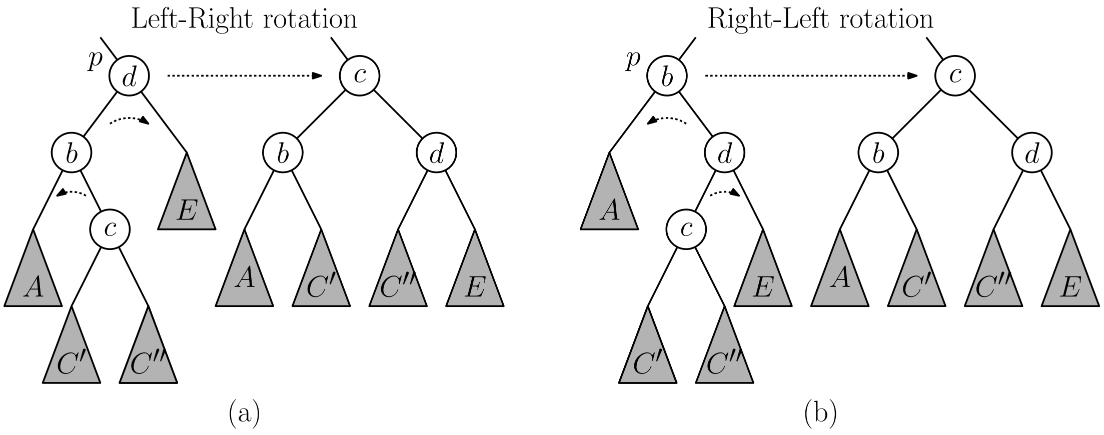


Fig. 22: Double rotations (`rotateLeftRight(p)` and `RotateRightLeft(p)`).

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 ±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. 23, 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.)

Let's first consider the case where the insertion took place in the the subtree A (see Fig. 23(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. 23(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.

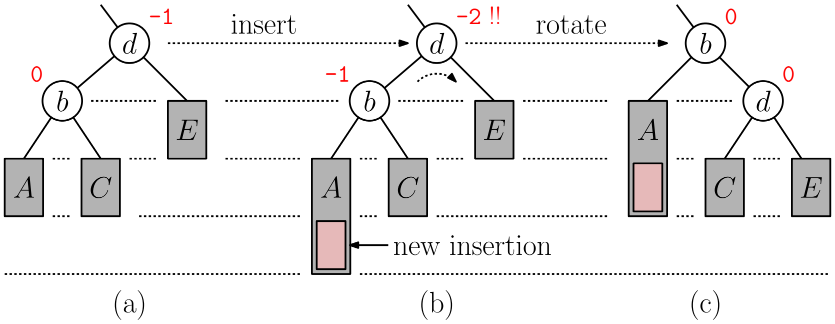


Fig. 23: Restoring balance after insertion through a single rotation.

Next, us consider the case where the insertion occurs within subtree C (see Fig. 24(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 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. 24(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.

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.

Insertion Implementation: The entire insertion procedure for AVL trees is shown in the following code fragment. It starts with a few utilities. We assume that we store the height of each node in a field p.height, which contains the height of the subtree rooted at this node. We

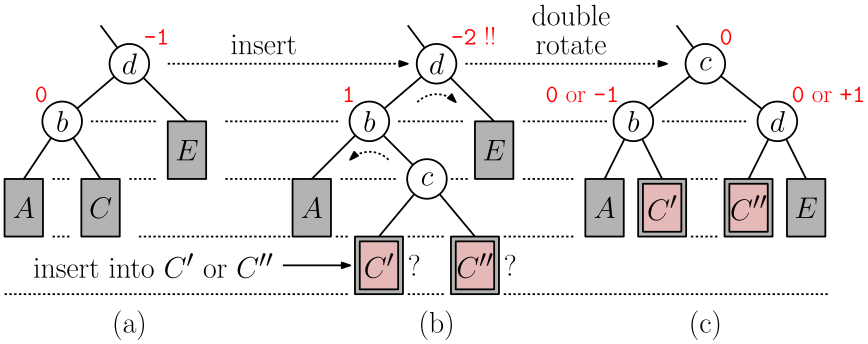


Fig. 24: Restoring balance after insertion through a double rotation.

define a utility function `height(p)`, which returns `p.height` if p is non-null and -1 otherwise. Based on this we provide a utility function `updateHeight`, which is used for updating the height's of nodes (assuming that their children's heights are properly computed). We also provide a utility for computing balance factors and the rotation functions. We omit half of the rotation functions since they are symmetrical, just with left and right swapped.

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 starts by applying the deletion algorithm for standard (unbalanced) binary search trees. Recall that this breaks into three cases, leaf, single child, and two children. This part of the deletion process is identical to the standard case. The only change is that (as in insertion) we restore balance to the tree by invoking the function `rebalance(p)` just prior to returning from each node. Even though we design this piece of code to work in the case of insertion, it can be shown that it works just as well for deletion.

In Fig. 25, we illustrate a deletion of a node (from the subtree E) which can be remedied by a single rotation. This happens because the left subtree (rooted at b) is too heavy following the deletion, and the left child of the left subtree (A) is at least as heavy as the right child (C). (The "?" in our figures illustrate places where the subtree's height is not fully determined. For this reason, some of the balance factors are listed as "x or y" to indicate the possible options.)

In Fig. 26, we illustrate an instance where a double-rotation is needed. In this case, the left subtree (rooted at b) is too heavy following the deletion, but the right child of the left subtree (C) is strictly heavier than the left child (A).

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, insertion could result in a cascade of $O(\log n)$ rebalancing operations.

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

AvlNode rotateRight(AvlNode p)          // right single rotation
{
    AvlNode 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
}

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

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

AvlNode insert(Key x, Value v, AvlNode p) {
    if (p == null) {                    // fell out of tree; create new node
        p = new AvlNode(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);                // rebalance if needed
}

AvlNode rebalance(AvlNode p) {
    if (p == null) return p;            // null - nothing to do
    if (balanceFactor(p) < -1) {        // too heavy on the left?
        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) { // too heavy on the right?
        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
}
```

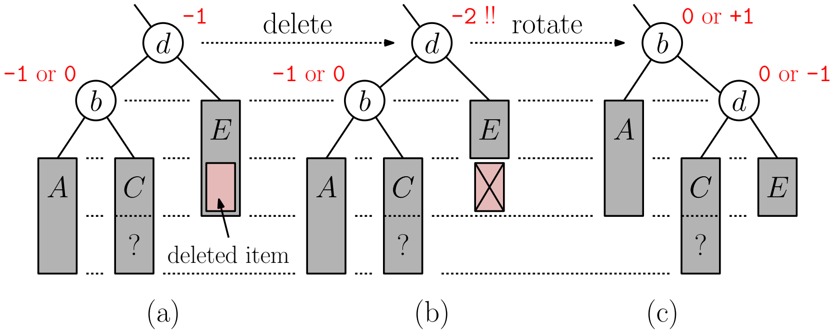


Fig. 25: Restoring balance after deletion with single rotation.

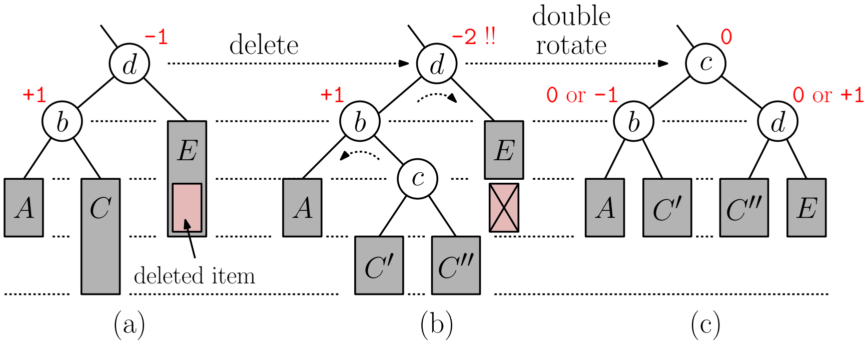


Fig. 26: Restoring balance after deletion with double rotation.

Lecture 6: 2-3, Red-black, and AA trees

"A rose by any other name": In today's lecture, we consider three closely related search trees. All three have the property that they support find, insert, and delete in time $O(\log n)$ for a tree with n nodes. Although the definitions appear at first glance to be different, they are essentially equivalent or very slight variants of each other. These are *2-3 trees*, *red-black trees*, and *AA trees*. Together, they show that the same idea can be viewed from many different perspectives.

2-3 Trees: An "ideal" binary search tree has n nodes and height roughly $\lg n$. (More precisely, the ideal would be $\lfloor \lg n \rfloor$, where we recall our convention that "lg" means logarithm base 2.) However, it is not possible to efficiently maintain a tree subject to such rigid requirements. AVL trees relax the height restriction by allowing the two subtrees associated with each node to be of *similar heights*.

Another way to relax the requirements is to say that a node may have either two or three children (see Fig. 27(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 requirement that for all $a \in A$, $c \in C$, and $e \in E$, we have

$$a < b < c < d < e,$$

(The concept of an *inorder traversal* of such a tree can be generalized, but it involves visiting each 3-node twice, once to visit the first key and again to visit the second key.) These are called *2-nodes* and *3-nodes*, respectively.

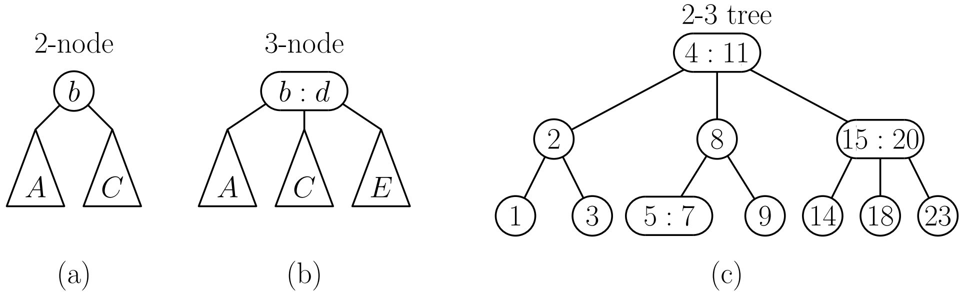


Fig. 27: (a) 2-node, (b) 3-node, and (c) a 2-3 tree.

A *2-3 tree* is defined recursively as follows. It is either:

- empty (i.e., `null`), or
- its root is a 2-node, and its two subtrees are each 2-3 trees of equal height, or
- its root is a 3-node, and its three subtrees are each 2-3 trees of equal height.

As we did with AVL trees, we define the height of an empty tree to be -1. (For an example, Fig. 27(c) shows a 2-3 tree of height 2.)

Since all the leaves are at the same level, and the sparsest possible 2-3 tree is a complete binary tree. We have the following direct consequence.

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

Since our foray into 2-3 trees will be brief, we will not bother to present an implementation. (Later this semester, we will discuss B-trees in detail, and a 2-3 tree is a special case of a B-tree.)

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.

In the descriptions that follow, it will be convenient to temporarily allow for the existence of "invalid" *1-nodes* and *4-nodes*. As soon as one of these exceptional nodes comes into existence, we will need to 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 AVL 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. 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 e with E and F as subtrees. We then *promote* the middle key *d* by inserting it (recursively) into the parent node (see Fig. 28(a)). If the parent is a 2-node, this can be accommodated without problem. On the other hand, if the parent is a 3-node, this creates a 4-node, and the splitting process continues recursively until either (a) we arrive at a node that does not overflow, or (b) we reach the root. When the root node overflows, the promoted key becomes the new root node, which must be a 2-node (see Fig. 28(b)). It is easy to see that this process preserves the 2-3 tree structural properties.

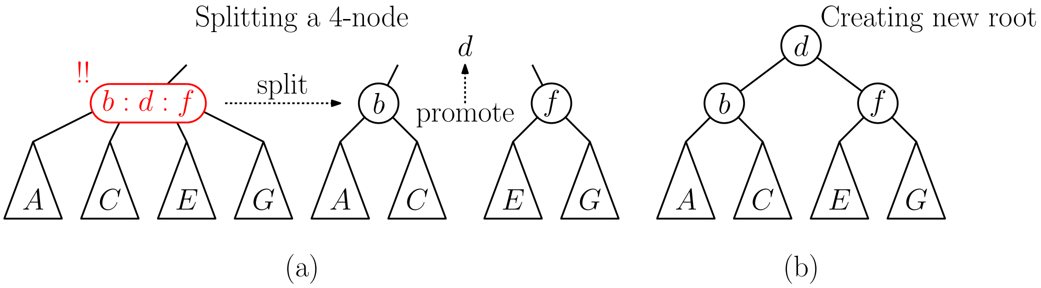


Fig. 28: 2-3 tree insertion: (a) splitting a 4-node into two 2-nodes and (b) creating a new root.

In the figure below, we present an example of the result of inserting key 6 into a 2-3 tree, which required two splits to resolve.

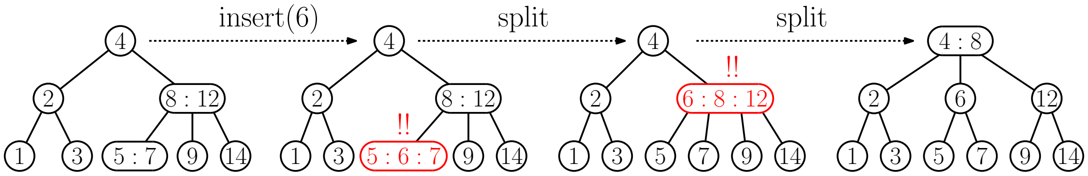


Fig. 29: 2-3 tree insertion involving two splits.

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.

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: 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. 30(a)). Then, we adopt the leftmost key and subtree from this sibling, resulting in two 2-nodes. (A convenient mnemonic is the equation $1 + 3 = 2 + 2$.)

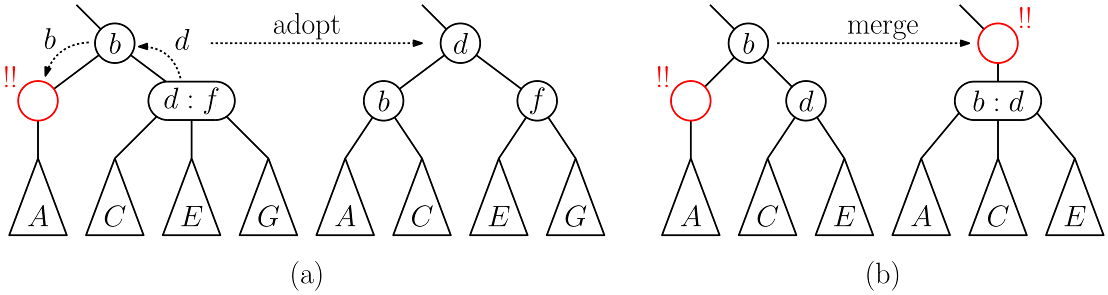


Fig. 30: 2-3 tree deletion: (a) adopting a key/subtree from a sibling and (b) merging two nodes.

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. 30(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. 31. 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.)

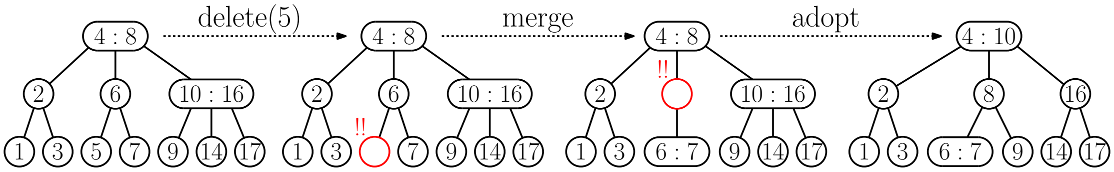
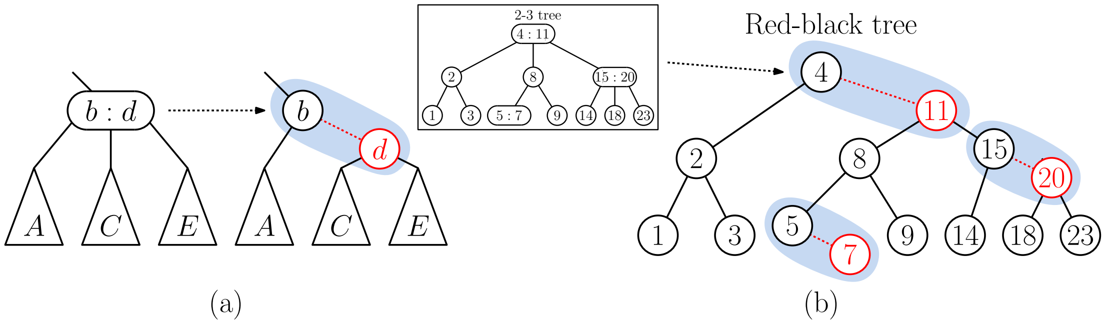


Fig. 31: 2-3 tree deletion involving a merge and an adoption.

Red-Black trees: While a 2-3 tree provides an interesting alternative to AVL trees, the fact that it is *not* a binary tree is a bit annoying. As we saw earlier in the semester, there are ways of representing arbitrary trees as binary trees. This suggests the idea of encoding a 2-3 tree as a 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 that works, however. 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. 32(a) below. The 2-3 tree shown in Fig. 27(c) above would be represented in the manner shown in Fig. 32(b).



(a) (b)

Fig. 32: Representing the 2-3 tree of Fig. 27 as an equivalent binary tree.

If we 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. It is easy to see that the resulting binary tree satisfies the following properties:

- Each node is either red or black.
- The root is black.
- All `null` pointers are labeled as black. (This is just a convenient convention.)
- If a node is red, then both its children are black.
- Every path from a given node to any of its `null` descendants contains the same number of black nodes.

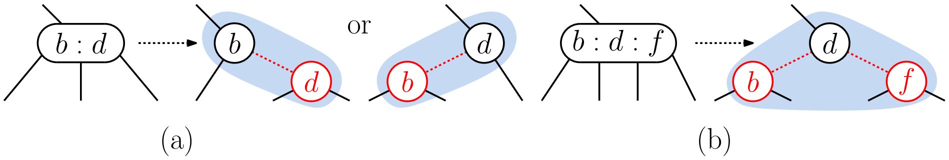
A binary search tree that satisfies these conditions is called a *red-black tree*. Because 2-3 trees have $O(\log n)$ height, the following is an immediate consequence:

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

It is easy to see that any the transformation that we described above for 2-3 trees always results in a valid red-black tree. Thus, we have:

Lemma: Every 2-3 tree corresponds to a red-black tree.

However, the converse is not true. 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. 33(a)). More seriously, the red-black condition allows for the sort of structure in Fig. 33(b), which clearly does not correspond to a node of a 2-3 tree.



(a) (b)

Fig. 33: Color combinations allowed by the red-black tree rules.

It is interesting to observe that this three-node combination can be seen as a way of modeling a node with four children. Indeed, there is a generalization of the 2-3 tree, called a *2-3-4 tree*, which allows 2-, 3-, and 4-nodes. Red-black trees as defined above correspond 1–1 with 2-3-4 trees. Red-black trees are the basis of `TreeMap` class in the `java.util` package. The principle drawback of red-black trees is that they are rather complicated to implement. For this reason, we will introduce a simpler variant of the red-black tree below, called an AA tree.

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[α] tree).

Anderson's idea was to allow the conversion described above between 2-3 trees and red-black trees, and forbid the other red-black combinations. In particular, each red node can arise only as the *right child* of a black node. (The other rules of red-black trees are the same.) 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. While AA-trees are simpler to code, they are a bit slower than red-black trees in practice.

The implementation of the AA tree has the following two noteworthy features:

We do not use `null` pointers: Instead, we create a special *sentinel node*, called `nil` (see Fig. 34(a)), and every `null` pointer is replaced with a pointer to `nil`. (Although the tree may have many `null` pointers, there is only one `nil` node allocated, with potentially many incoming pointers.) This node is considered to be black.

Why do this? Observe that `nil.left == nil` and `nil.right == nil`. This simplifies the code because we can always de-reference a pointer, without having to check first whether it is `null`.

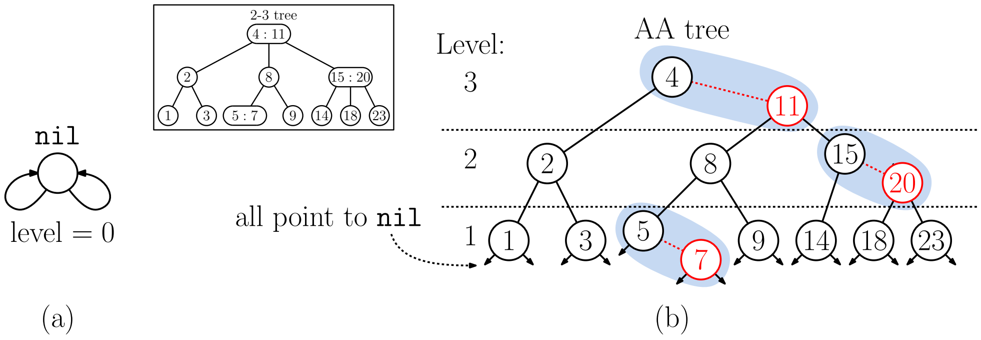


Fig. 34: AA trees: (a) the `nil` sentinel node, (b) the AA tree for the 2-3 tree of Fig. 27.

We do not store node colors: Instead, each node p stores a *level number*, denoted `p.level` (see Fig. 34(b)). Intuitively, the level number encodes the level of the associated node in the 2-3 tree. Formally, `nil` node is at level 0, and if `q` is a black child of some node `p`, then `p.level` = `q.level` + 1, and if `q` is a red child of `p`, then they have the same level numbers.

We do not need to store node colors, because we can determine whether a node is red by testing that its level number is the same as its parent.

It is surprising that our representation does not actually assign color to the nodes! It is not needed because color information is implicitly encoded in the level numbers. For example, if `p.right.level == p.level`, then we can infer that `p.right` is a red node (assuming it is not `nil`).

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 rotation operations go under the special names `skew` and `split`. They are defined as follows:

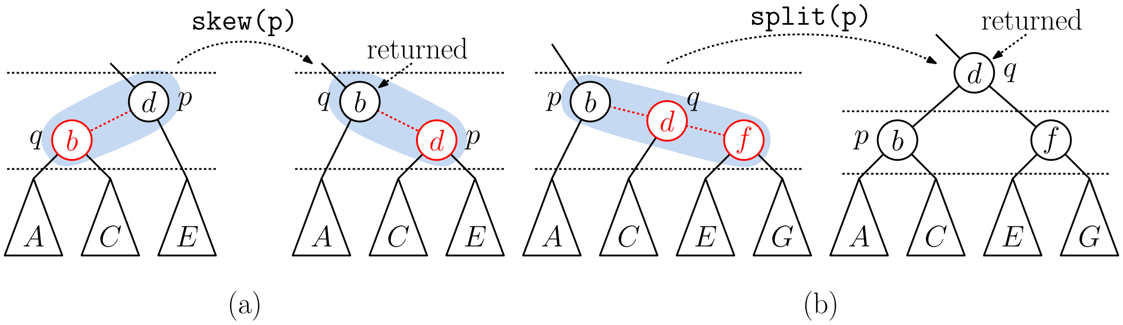


Fig. 35: AA restructuring opertions (a) skew and (b) split. (Afterwards q may be red or black.)

`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. 35(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 , 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. 35(b)).

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. 36(a)).

The first problem might arise is that this newly inserted red node is a left child, which is not allowed (see Fig. 36(b)). Letting p denote the node's parent, this is easily remedied by performing `skew(p)` (see Fig. 36(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. 36(d)). Note that the split operation moves the middle node of the chain up to the next level of the tree. The problems

AA tree skew and split utilities

```
AANode skew(AANode 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.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
}
```

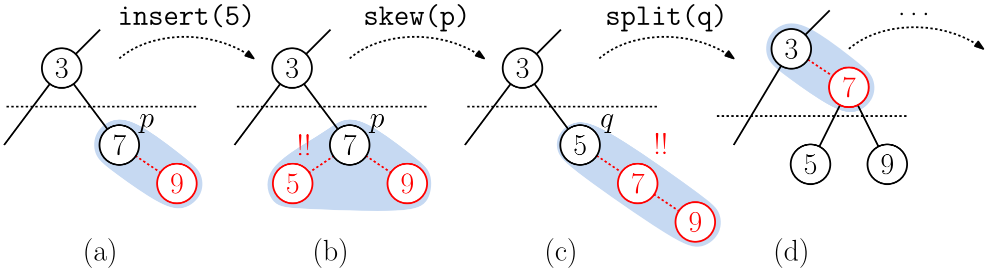


Fig. 36: AA insertion: (a) Initial tree, (b) after insertion, (c) after skewing, (d) after splitting.

that we just experienced may occur with this promoted node, and so the skewing/splitting process generally propagates up the tree to the root.

The insertion function is provided in the following code block. In spite of this lengthy above explanation of how restructuring is performed, the entire restructuring process is handled very elegantly by the statement "`return split(skew(p))`". (This 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. 37 (mimicking the 2-3 tree of Fig. 29).

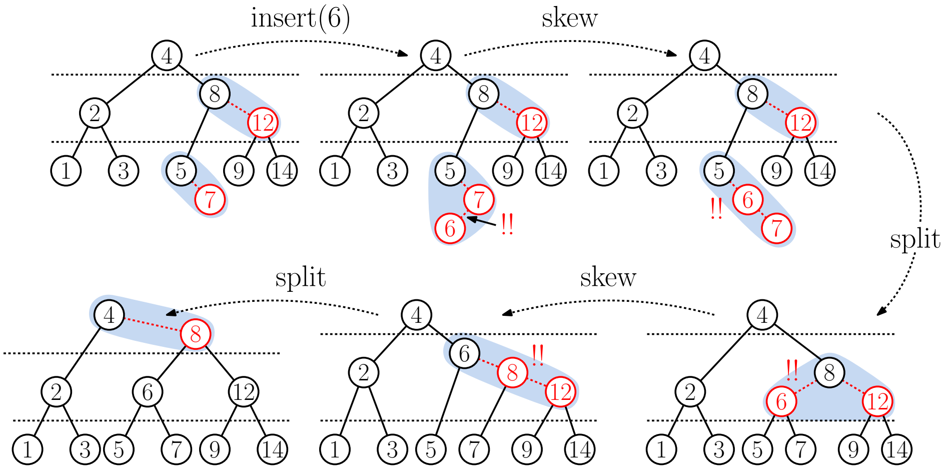


Fig. 37: Example of AA-tree insertion.

AA-tree deletion: As usual deletion is more complex than insertion. If this is not a leaf node, we find a suitable replacement node (it's inorder successor). 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 to its 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.

AA-Tree update level utility

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

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 to p, one to `p.right`, and one to `p.right.right` (see Fig. 38). 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. 38). Whew! These splits may not be needed, but remember that the split function only modifies the tree if needed. The restructuring function, called `fixupAfterDelete`, is presented in the following code fragment.

AA-Tree Deletion Utility

```
AANode fixupAfterDelete(AANode p) {
    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;
}
```

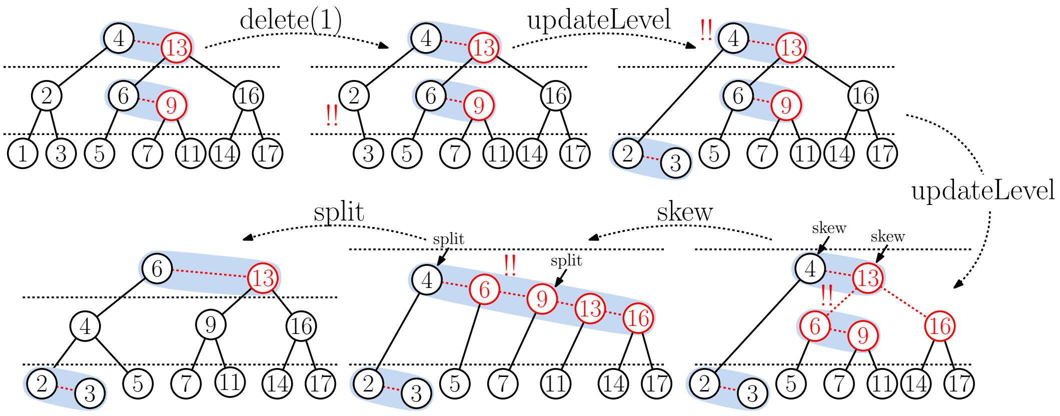


Fig. 38: Example of AA-tree deletion.

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 `fixupAfterDelete` to restructure the tree. All the operations (find, insert, and delete) take time proportional to the height of the tree, that is, $O(\log n)$.

AA Tree Deletion

```
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 fixupAfterDelete(p);             // fix structure after deletion
    }
}
```

Lecture 7: Randomized Search Structures: Treaps and Skip Lists

Randomized Data Structures: A common design techlque 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.

This may seem perplexing at first. Shouldn't an intelligent, clever algorithm designer be able to make better decisions than a simple random number generator? The issue is that a deterministic decision-making process may be susceptible to systematic biases, which in turn can result in unbalanced data structures. Randomness creates a layer of "independence," which can alleviate these systematic biases.

In this lecture, we will consider two famous randomized data structures, which were invented at nearly the same time. The first is a randomized version of a binary tree, called a *treap*. This data structure's name is a portmanteau (combination) of "tree" and "heap." It was developed by Raimund Seidel and Cecilia Aragon in 1989. (Remarkably, 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.)

The other data structure is the *skip list*, which is a randomized version of a linked list where links can point to entries that are separated by a significant distance. This was invented by Bill Pugh (a professor at UMD!).

Because the treaps and skiplists are randomized data structure, running times depend 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 protest, since this allows for rare instances where the performance 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.

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. 39(b)). From your earlier courses you should know a data structure that has this very property—such a tree is generally called *heap*.

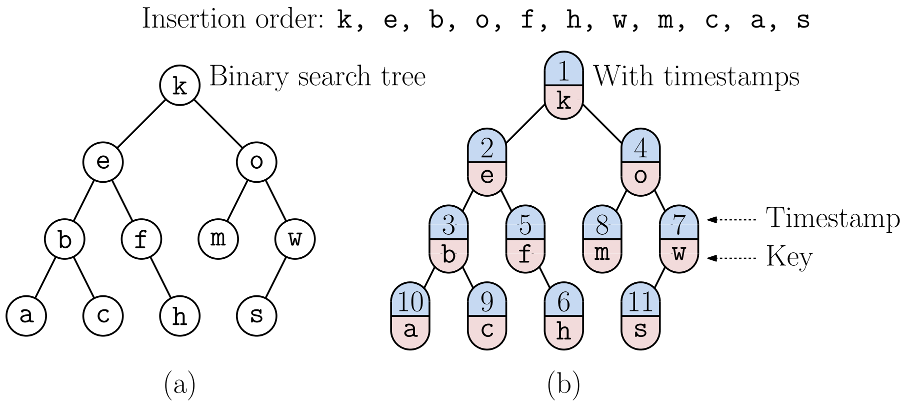


Fig. 39: (a) A binary search tree and (b) associating insertion timestamps with each node.

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 (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 will be sufficiently rare that they cannot significantly affect the tree's structure. The next question is whether we can maintain this structure efficiently. The answer is "yes", and it is remarkably easy.

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 walk 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 rotation (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 so simple, that we will leave as an exercise.

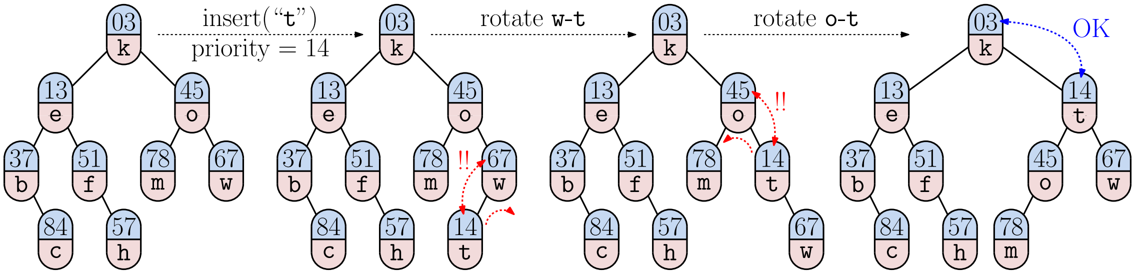


Fig. 40: Treap insertion.

Treap Deletion: 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.

There is, however, a cute trick for performing deletions. We first locate the node in the tree and then set its priority to ∞ (see Fig. 41). We then apply rotations to sift it down the tree to the leaf level, where we can easily unlink it from the tree.

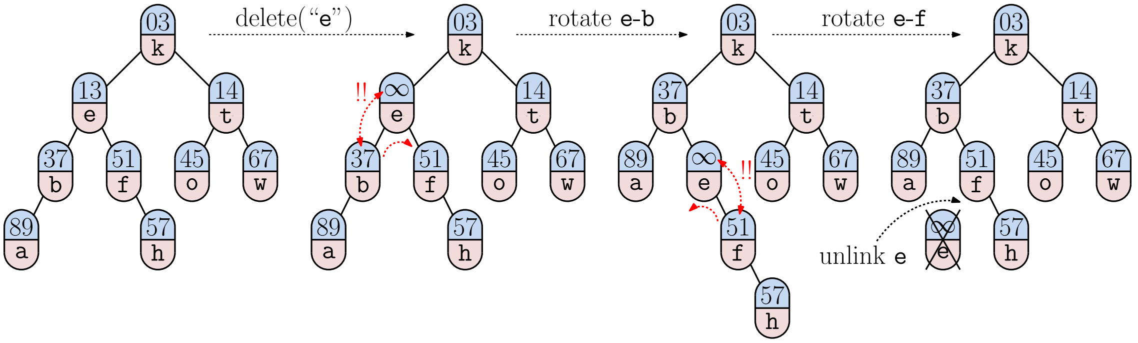


Fig. 41: Treap deletion.

The treap is particularly easy to implement because we never have to worry about adjusting the priority field. For this reason, treaps are among the fastest data tree-based dictionary structures.

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 key value. 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. 42). 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 an *"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`.

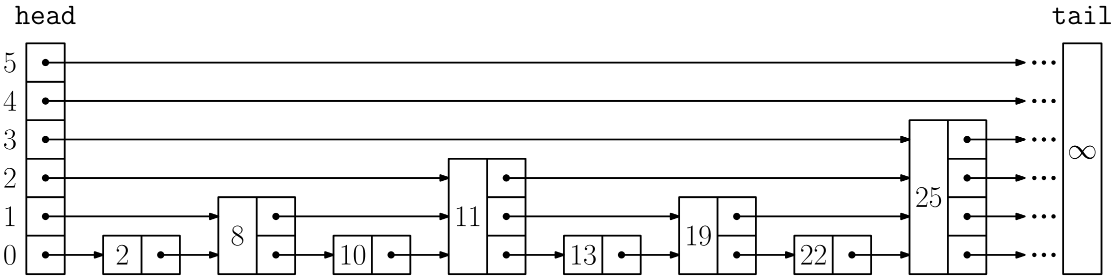


Fig. 42: The "ideal" skip list.

To search for a key 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 ∞, 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. 43).

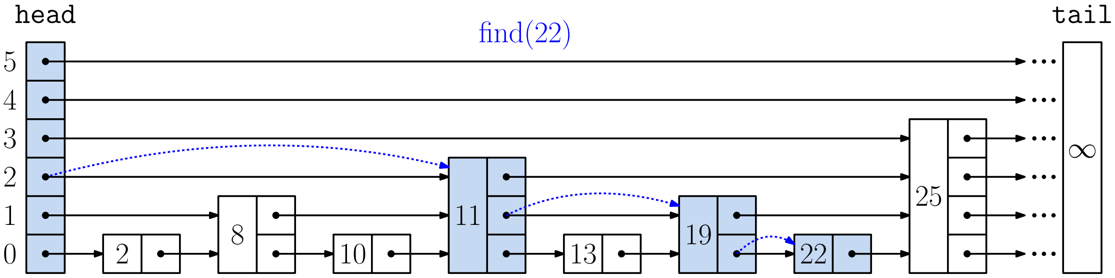


Fig. 43: Searching the ideal skip list.

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. 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 $O(\log n)$.

Randomized Skip Lists: Unfortunately, like a perfectly balanced binary tree, the ideal skip list is too pure to be able to use for 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*?

Suppose that we have built the skip list up to some level i, and we want to extend this to level $i+1$. Imagine of 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 level k will be $n/2^k$, which means that the expected number of nodes at level $\lceil \lg n \rceil$ is a constant. Fig. 44 shows what such a *randomized skip list*, or simply *skip list*, will look like.

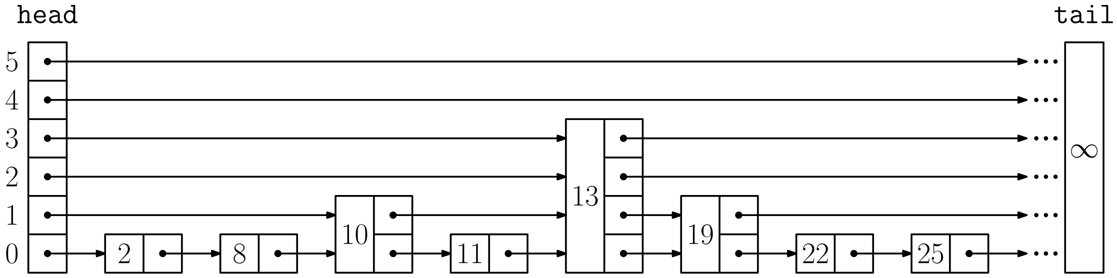


Fig. 44: A (randomized) skip list.

Space Analysis: Unlike binary search trees whose nodes are all of the same size, the nodes of a skip list have variable sizes. If we assume that the maximum number of levels of a skip is $O(\log n)$, then in the worst case every node contributes to every level, and the skip list would have total storage of $O(n \log n)$. In the best case (from the perspective of storage), every node contributes only to the lowest level, and the total storage would be $O(n)$. Note that either of these cases is *extremely* unlikely.

To describe the expected case, observe that in expectation, exactly half of the nodes from one level are promoted to the next. Thus in expectation, all n nodes contribute to level 0, $n/2$ contribute to level 1, $n/4$ contribute to level 2, and generally $n/2^i$ contribute to level i. In summary in expectation, the total storage (for pointers) is:

$$\sum_{i=0}^{h-1} \frac{n}{2^i} \;=\; n \sum_{i=0}^{h-1} \frac{1}{2^i} \;=\; n\left(2 - \frac{1}{2^i}\right) \;\leq\; n\left(2 - \frac{1}{2^m}\right) \;\leq\; 2n.$$

(Here we have made use of the formula for the geometric series, $\sum_{i=0}^{m-1} (\frac{1}{2})^i = 2 - (\frac{1}{2})^m$.) Thus, the expected storage just for the pointers is $O(n)$. Storing the keys themselves takes just $O(n)$ storage. Finally, the head and tail nodes take $O(\log n)$ storage each, but $\log n$ is dominated by n asymptotically, so we can ignore their contribution.

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.

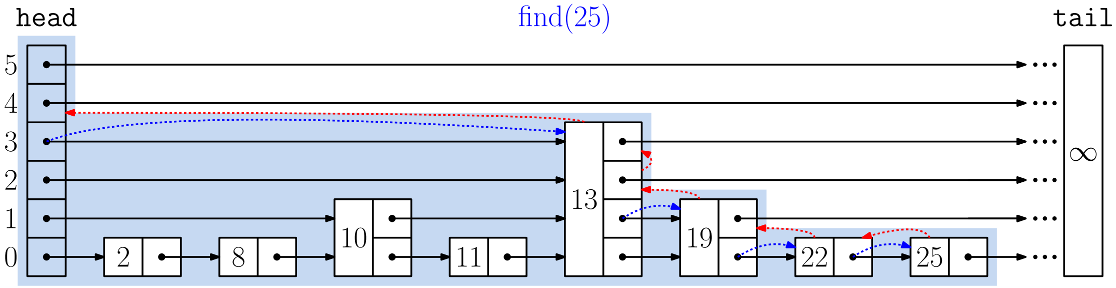


Fig. 45: The search path (blue) to $x = 25$ and the reverse search path (red).

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. 45, 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, and $E(6)$ would be the expected number of steps at all the levels.) Whenever we arrive at some node of level i, the probability that it contributes to the next higher level is exactly p. With the remaining probability $1-p$ we stay at the same level. Counting the current node we are visiting (+1), we can express $E(i)$ by the recurrence:

$$E(i) \;=\; 1 + pE(i-1) + (1-p)E(i).$$

With a bit of algebra, we have:

$$E(i) \;=\; \frac{1}{p} + E(i-1).$$

By expansion, it is easy to verify that $E(i) = \frac{i}{p}$. Since $i \leq O(\log n)$, and by our assumption that p is a constant, it follows that the expected search time is $O(\log n)$.

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. 46).

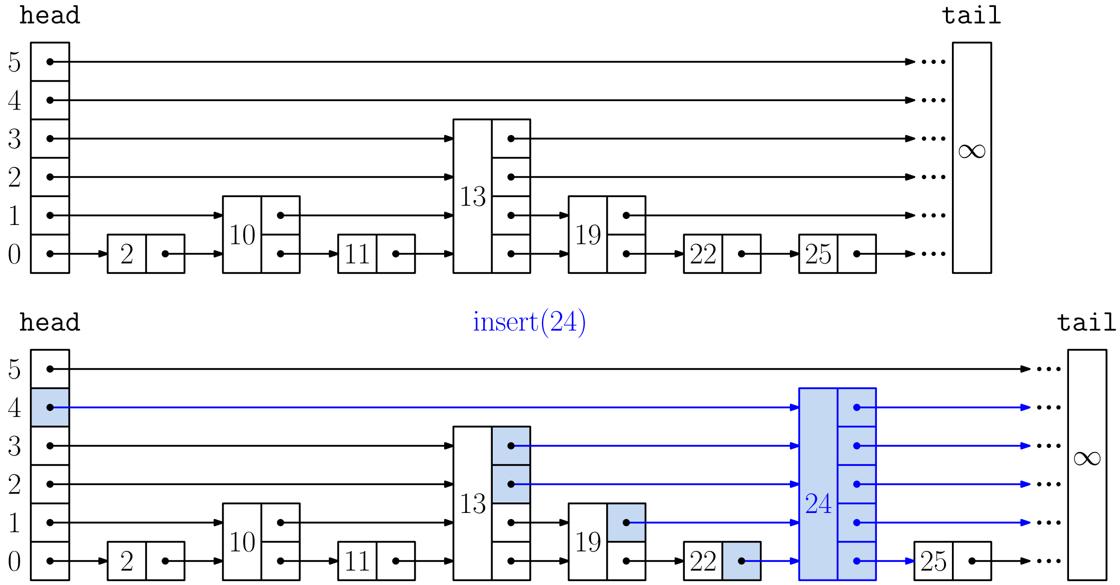


Fig. 46: Inserting a new key 24.

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. 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.

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 8: 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 $\Omega(n)$.

If deletions are also random and symmetric deletion is used (that is, selecting the replacement key randomly as the inorder successor or inorder predecessor), all dictionary operations have expected running time of $O(\log n)$. (If deletions are not symmetric, the height of the tree can degenerate to $\Omega(\sqrt{n})$ over time.)

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. This is a very well known "classic" data structure, but AVL trees are neither the simplest nor fastest in practice.

2-3 Trees: These trees are perfectly balanced trees (all leaf nodes at the same depth), where each node is allowed to have either two or three children. Rebalancing is performed by splitting and merging nodes. Height is guaranteed to be $O(\log n)$, and all dictionary operations run in $O(\log n)$ time. There is some space wastage, because each node allocates space for a three children, but there may only be two.

Red-Black and AA Trees: These are both binary-tree implementations of the 2-3 tree. (More accurately, the standard red-black tree implements the 2-3-4 tree.) Height is guaranteed to be $O(\log n)$, and all dictionary operations run in $O(\log n)$ time.

Red-black trees are popular because they don't suffer the space wastage of 2-3 trees and they run very fast in practice. Unfortunately, the number of special cases is high, and we presented the AA tree, which is a bit slower but easier to code.

Treap and Skip Lists: These are randomized structures, which means that they rely on a random-number generator to determine their structures. The treap uses random priorities and a heap-based ordering of priorities. The skip list generalizes concept of a linked list by layering multiple linked-lists, where roughly half of the nodes of one level are chosen at random to be promoted to the next.

In both cases, all dictionary operations run in $O(\log n)$ time in expectation (over random choices). These structures are easy to implement and very fast in practice. The randomized nature of the data structure is troubling to some, but the probability of poor performance is very low. (Note that, unlike standard binary search trees, the expected-case behavior depends only on the random number generation, not on the actions of the user.)

At this point you might wonder, is that everything? Far from it! While these represent the very best known ordered-dictionary structures, there are many other operations that a user might want the data structure to support, and there are a number of variants on these ideas and specialized data structures for particular operations. Here are a few examples:

Order-statistic queries: Given an integer k, where $1 \leq k \leq n$, find the kth smallest element in the set.

Range queries: Given two keys x_0 and x_1, compute the sum (or any associative operation) on all the keys of the dictionary that lie between x_0 and x_1.

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 $> x$. 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.

Expected-Case Optimal Search Tree: Suppose that not all searches are equally likely. For example, a small number of keys are much more likely to be sought than the others. Intuitively, we should place these items much closer to the root, so that searches for them are faster. What is the best way to do this?

All of the aforementioned search structures can be adapted to solve the first three problems (order-statistic, range, and split/merge) in the same time as the other standard dictionary operations. In some cases, additional information needs to be stored in the nodes of the tree tree. (We will leave these as exercises.)

The problem of expected-case optimality is a different story, however. The structures we have seen do not address this question. Note that this is a static problem, in the sense that we want to build a single tree, that will perform the best over a long series of find operations. The problem can be stated more formally as follows. Suppose that the tree stores keys $\{x_1, \ldots, x_n\}$, and 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 $E(T)$?

There does exist an efficient algorithm for computing the optimum binary-search tree. (It is a nice 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*. 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. This is really quite clever, when you consider the fact that the tree 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 $O(\log n)$. This type of 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 red-black trees, treaps, and skip lists. 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. 47. 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

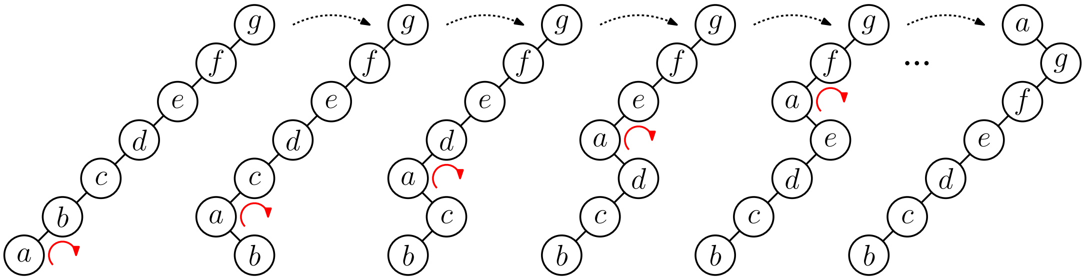


Fig. 47: Single rotations up to the root—the tree is still poorly balanced.

and apply rotations at each of these nodes. For example, in Fig. 48, 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?)

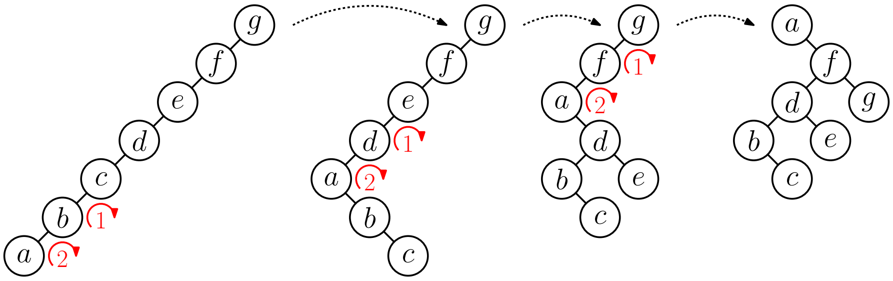


Fig. 48: Two-at-a-time rotations up to the root—the tree height is cut almost in half.

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. 49(a)), and continue up the tree.
  - **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. 49(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. 49(a)), and are now done.
- If p is the root of T, we are done.

A full example is shown in Fig. 50. 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.

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

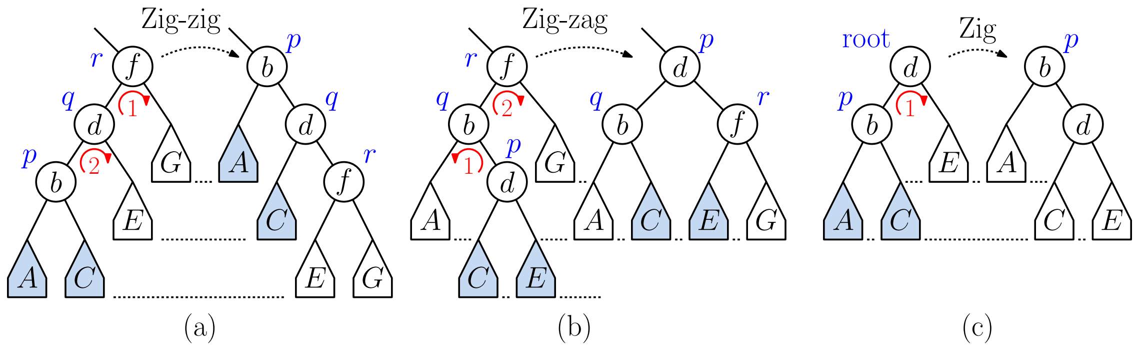


Fig. 49: Splaying cases: (a) Zig-Zig, (b) Zig-Zag and (c) Zig.

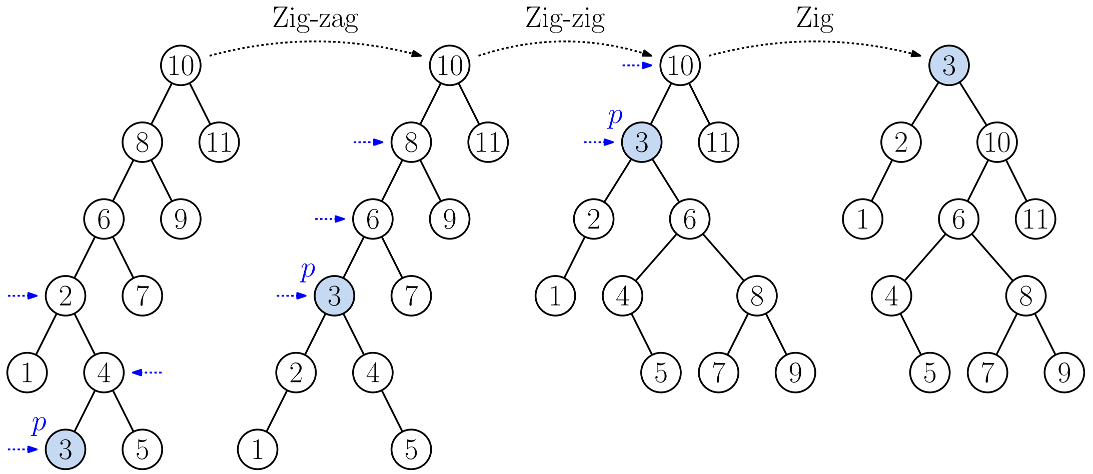


Fig. 50: The full splay operation on p.

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. 49(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. 49(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.

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?

`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. (Why bother moving x to the root? 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 nodes nearest to the root.)

`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. 51). The remaining nodes are hung off as the left subtree L of this node.

`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. 52). 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.

Otherwise, both subtrees are nonempty, and we next find an appropriate replacement node. To do this, we perform `R.splay(x)`. (It might be smarter to flip a coin and do `L.splay(x)` half the time for the sake of symmetry. This will not affect the theoretical analysis of the tree's performance.) Let's call the resulting subtree R'. Since x is not in R and all the keys in this subtree are greater than x, this will bring the smallest key y of R to the root. This implies that y is x's inorder successor from the original tree. It

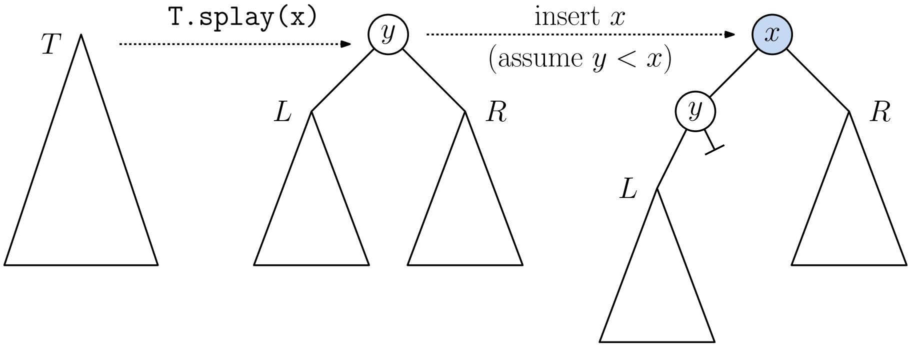


Fig. 51: Splay-tree insertion of x.

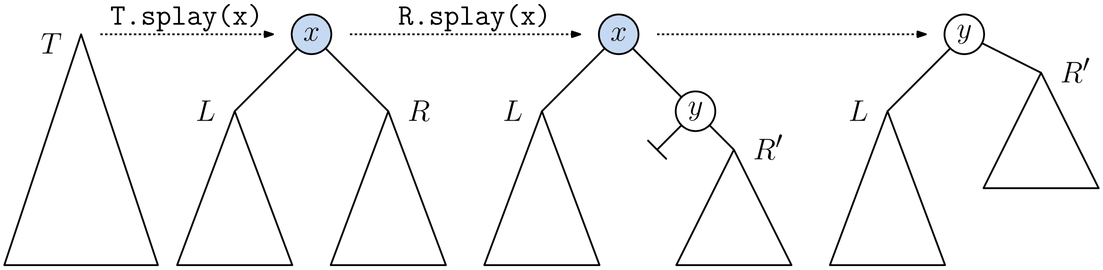


Fig. 52: Splay-tree deletion of x.

follows that y can have no left child in R'. To complete the operation, we simply link L as the left child of R'.

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 $\text{size}(p)$ 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

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

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

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

(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 $\mathrm{rank}(x)$ as $r(x)$ and $\mathrm{size}(x)$ and $s(x)$. Suppose that we perform a zig-zig rotation involving three nodes x, y, and z, as shown in Fig. 49(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.

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 $O(n\log n)$.)

Lecture 9: B-Trees

B-trees: 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, an entire *block* (or *page*) is input at once. So it makes sense to design the tree so that each node of the tree essentially occupies one entire block. This idea applies more generally to modern memory hierarchies, where memory is divided into various levels of caches. The data structure we will discuss today is appropriate whenever memory can be accessed efficiently in blocks.

An alternative strategy is to use *multiway search trees*, where each node is chosen so that it fits coincides with a memory block. 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.)

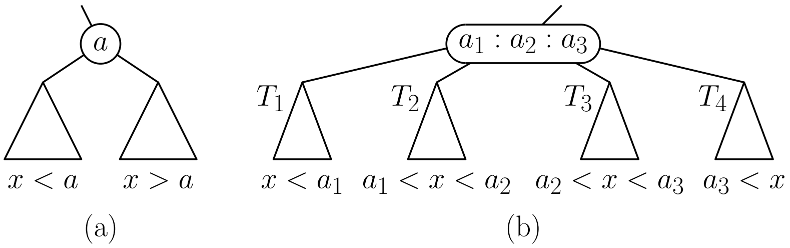


Fig. 53: Binary and 4-ary search tree nodes.

B-trees are multiway search trees, in which we achieve balance by constraining the "width" of each node. We have already introduced this concept in our discussion of 2-3 trees. In this lecture, we will consider how to generalize this to nodes of arbitrary width.

B-trees were first introduced way back in 1970 by Rudolf Bayer and Edward McCreight. They have proven to be very popular, but with popularity comes variety. 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:

- 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.

A B-tree of order 5 is shown in Fig. 54.

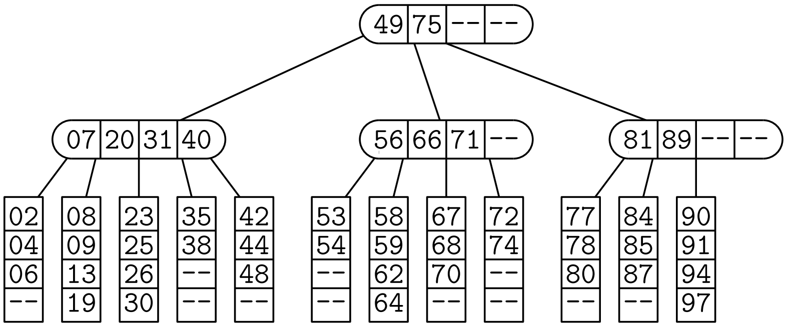


Fig. 54: B-tree of order 3, also known as a 2-3 tree.

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 depth of the tree is 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 $(\lg n)/\gamma$, where $\gamma = \lg(m/2)$.

Proof: To avoid messy floor-ceiling arithmetic, let's just assume that m is even. Let $N(h)$ denote the number of nodes in the skinniest possible order-m B-tree of height h. The root has at least two children, each of them has at least $m/2$ children. Therefore, there are at least two nodes at depth 1, $2(m/2)$ nodes at depth 2, $2(m/2)^2$ nodes at depth 3, $2(m/2)^3$ nodes at depth 4, and in general, there are at least $2(m/2)^{k-1}$ nodes at depth k. Thus, the total number of nodes in an entire tree of height h is at least

$$N(h) \;=\; \sum_{i=1}^{h} 2\left(\frac{m}{2}\right)^{i-1} \;=\; 2\sum_{i=0}^{h-1}\left(\frac{m}{2}\right)^{i}.$$

This is a geometric series of the form $\sum_i c^i$, where $c = m/2$, and by standard formulas we have $N(h) = 2(c^h - 1)/(c - 1)$. Assuming that m is relatively large, we may ignore the -1 in the numerator and denominator to yield $N(h) \approx 2c^h/c = 2c^{h-1} = 2(m/2)^{h-1}$. Each node contains at least $m/2 - 1$ keys. Again, assuming that m is large, we can approximate this as $m/2$. So the number of keys is at least $(m/2)2(m/2)^{h-1} = 2(m/2)^h$. By our hypothesis, the tree has n keys, and thus (recalling that "lg" means log base 2) we infer that

$$\begin{aligned} n \;\geq\; N(h) \;\geq\; 2\left(\frac{m}{2}\right)^h \quad &\Rightarrow\quad \left(\frac{m}{2}\right)^h \;\leq\; \frac{n}{2} \\ &\Rightarrow\quad h \lg\frac{m}{2} \;\leq\; \lg\frac{n}{2} \\ &\Rightarrow\quad h \;\leq\; \left(\lg\frac{n}{2}\right) \Big/ \lg\frac{m}{2} \\ &\Rightarrow\quad h \;\leq\; (\lg n) \Big/ \lg\frac{m}{2}. \end{aligned}$$

In the case where $m = 100$, the above result 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 depth roughly 5.

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. In this case, we are storing integers as the elements. We place twice as many elements in each leaf node as in each internal node, since we do not need the storage for child pointers.

B-Tree Node

```
final int M = ...                   // order of the B-tree

class BTreeNode {
    int         nChildren;          // number of children (from M/2 to M)
    BTreeNode   child[M];           // children pointers
    Key         key[M-1];           // keys
    Value       value[M-1];         // values
}
```

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, 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.

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).

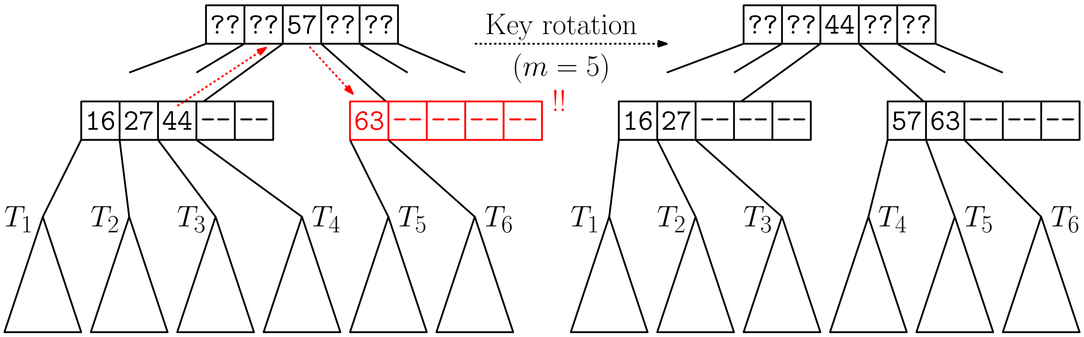


Fig. 55: Key rotation for a B-tree of order $m = 5$.

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. 55, 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, we prefer this operation whenever it is available.

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. 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 $m \geq 2$. 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 $m \geq 1$.

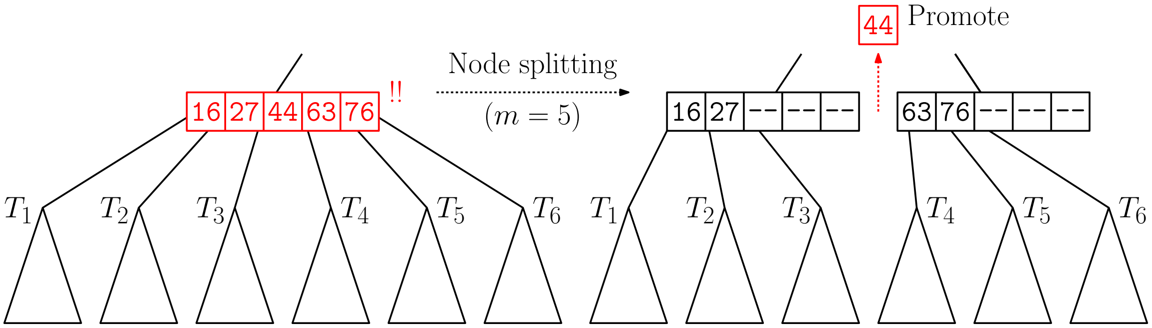


Fig. 56: Node splitting for a B-tree of order $m = 5$.

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. 56). 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. This node 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. 57). 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.

Since the parent has lost a key and a child, the merging process may propagate to the parent node.

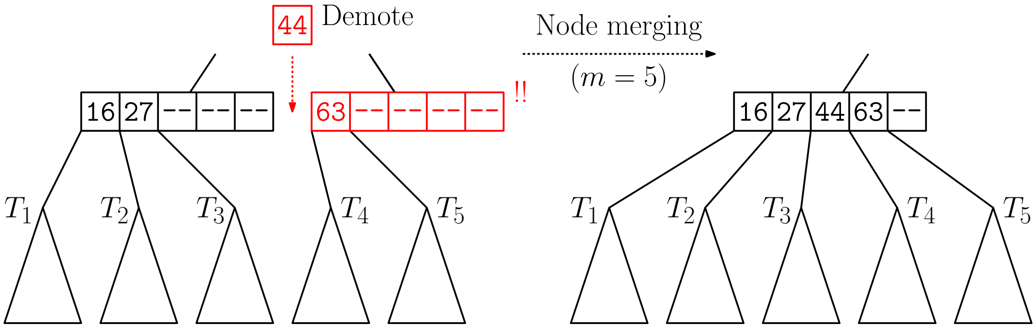


Fig. 57: Node merging for a B-tree of order $m = 5$.

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.

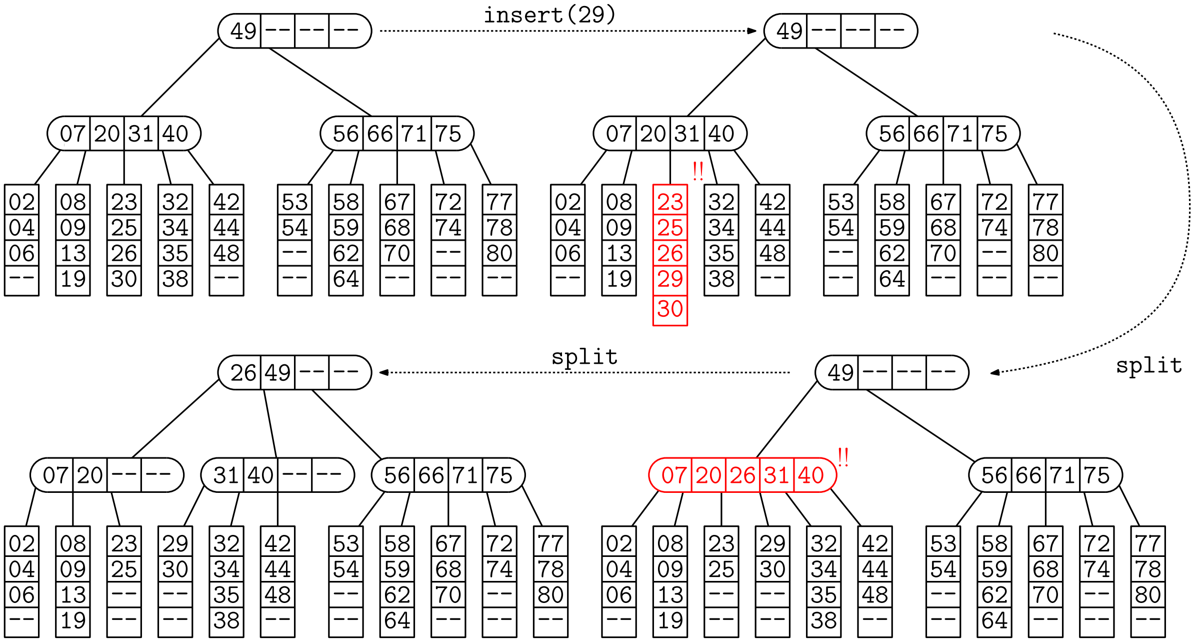


Fig. 58: Insertion of key 29 $(m = 5)$.

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. 58). 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. 59).

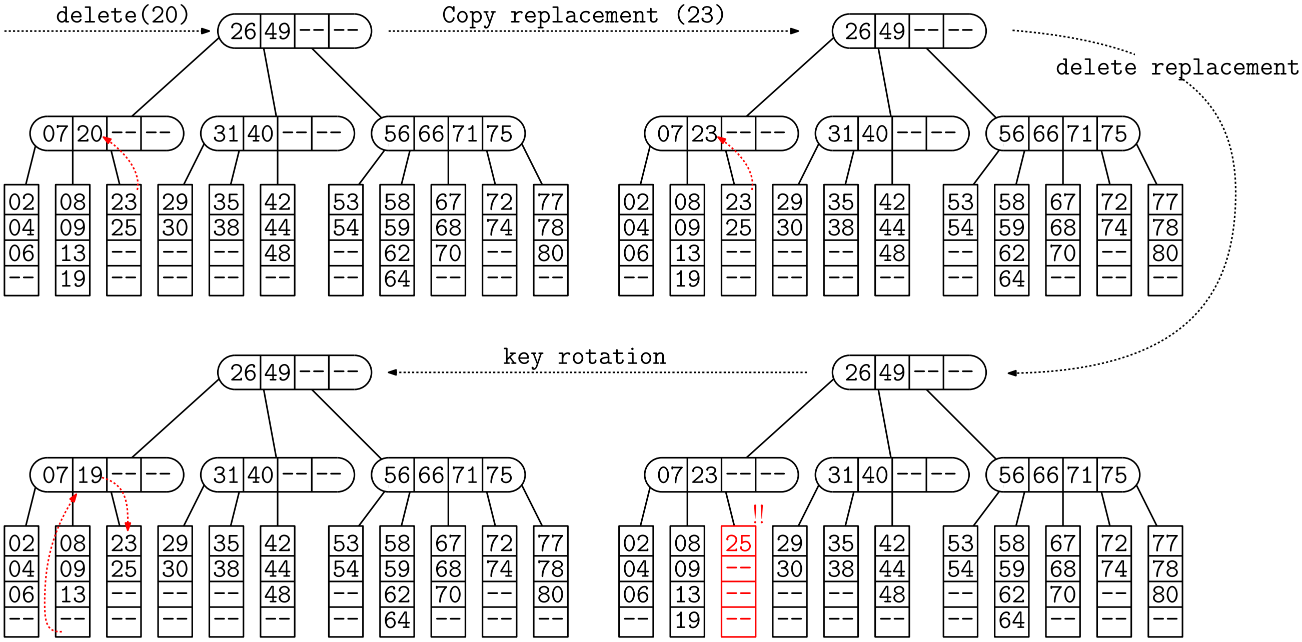


Fig. 59: Deletion of key 23 ($m = 5$).

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. 60).

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:
  - 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.

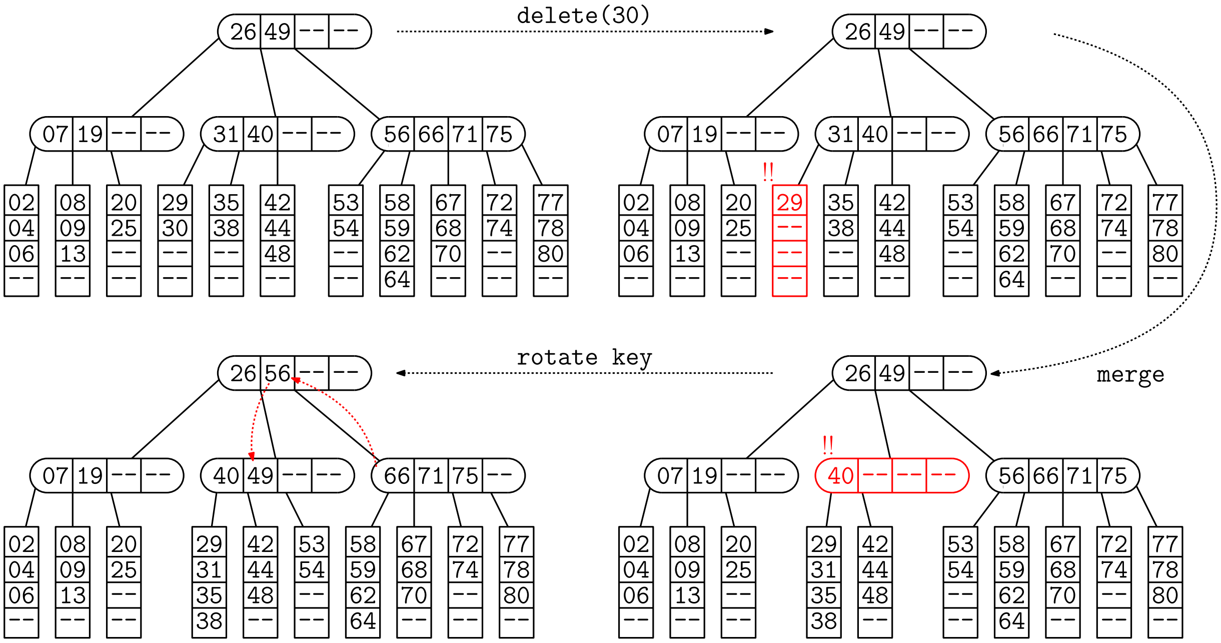


Fig. 60: Deletion of key 30 ($m = 5$).

  - 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 $a_{i-1} < x \leq a_i$.)

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 reaching $x_{\max}$.

Lecture 10: Hashing - Basic Concepts and Hash Functions

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*. The price we pay is that we cannot perform dictionary operations based on search order, such as range queries (finding the keys x such that $x_1 \leq x \leq x_2$) or nearest-neighbor queries (find the key closest to a given key x).

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. We select a *hash function* $h(x)$, which is an easily computable function that maps a key x to a "virtually random" 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. Of course, 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.

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. Note that hashing is not usually used for continuous data, such as floating point values, because similar keys 3.14159 and 3.14158 may be mapped to entirely different locations.

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:

- It should be *efficiently computable*, say in constant time and using simple arithmetic operations.
- It should produce *few collisions.* Two additional aspects of a hash function implied by this are:
  - It should be a function of *every bit* of the key (otherwise keys that differ only in these bits will collide)
  - It break up (scatter) naturally occuring *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. For example, if the c_i's are characters of a string, we might convert each to an integer (e.g., using Java's function `Character.getNumericValue(c[i])`, which returns the character's Unicode value as an integer) and then for some fixed value p (which you as the hash function designer pick) compute the polynomial

$$h(x_0, \ldots, x_n) \;=\; \left(\sum_{i=0}^{k-1} c_i p^i \right) \bmod m$$

For example, if $k = 4$ and $p = 37$, the associated polynomial would be $c_0 + c_1 37 + c_2 37^2 + c^3 37^3$.

You might wonder whether we can efficiently compute high-order polynomial functions. A useful algorithm for computing polynomials is called *Horner's rule*. The idea is to compute the polynomial through nested multiplications. To see how it works, observe that the above polynomial could be expressed equivalently as

$$c_0 + c_1 37 + c_2 37^2 + c^3 37^3 \;=\; ((c_3 \cdot 37 + c_2) \cdot 37 + c_1) \cdot 37 + c_0.$$

Using this idea, the polynomial hash function could be expressed in Java as

Polynomial hash function with Horner's rule

```
public int hash(String c, int m) {  // polynomial hash of a string
    final int P = 37;               // replace this with whatever you like
    int hashValue = 0;
    for (int i = c.length()-1; i >= 0; i--) { // Horner's rule
        hashValue = P * hashValue + Character.getNumericValue(c.charAt(i));
    }
    return hashValue % m;           // take the final result mod m
}
```

Randomization and Universal Hashing: Any deterministic hashing scheme runs the risk that we may (very rarely) come across a set keys that behaves badly for this choice. As we have seen before, one way to evade attacks by a clever adversary is to employ *randomization*. Any given hash function might be bad for a particular set of keys. So, it would seem that we can never declare that any one hash function to be "ideal."

One way to approach this conundrum is to flip the question on its head. Rather than trying to determine the chances that a fixed hash function works for a random set of keys, let us instead fix two keys x and y, say, and then select our hash function h at random out of large bag of possible hash functions. Then we ask the question, what is the probability that $h(x) = h(y)$, given that the choice of h is random. Since there are m table entries, a probability of $1/m$ would be the best we could hope for.

This gives rise to the notion of *universal hashing*. A hashing scheme is said to be *universal* if the hash function is selected randomly from a large class of functions, and the probability of a collision between any two fixed keys is $1/m$.

There are many different universal hash functions. Let's consider one simple one (which was first proposed by the inventors of universal hashing, Carter and Wegman). First, let p be any prime number that is chosen to be larger than any input key x to the hash function. Next, select two integers a and b at random where

$$a \in \{1, 2, \ldots, p-1\} \qquad \text{and} \qquad b \in \{0, 1, \ldots, p-1\}.$$

(Note that $a \neq 0$.) Finally, consider the following linear hash function, which depends on the choice of a and b.

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

As a and b vary, this defines a *family* of functions. Let H_p denote the class of hash functions that arise by considering all possible choices of a and b (subject to the above restrictions). The following theorem shows that H_p is a universal hashing system by showing that the probability that two fixed keys collide is $1/m$. The proof is not terribly deep, but it involves some nontrivial modular arithmetic. We present it for the sake of completeness.

Theorem: Consider any two integers x and y, where $0 \leq y < x < p$. Let $h_{a,b}$ be a hash function chosen uniformly at random from H_p. Then the probability that $h_{a,b}(x) = h_{a,b}(y)$ is at most $1/m$.

Proof: (Optional) Let us select a and b randomly as specified above and let $h = h_{a,b}$. Observe that $h(x) = h(y)$ if and only if the two values $(ax + b) \bmod p$ and $(ay + b) \bmod p$ differ from each other by a multiple of m. This is equivalent to saying that there exists an integer i, where $|i| \leq (p-1)/m$, such that:

$$(ax + b) \bmod p = (ay + b) \bmod p + i \cdot m.$$

Because p is prime, we can express this equivalently as

$$ax + b \equiv (ay + b) + i \cdot m \pmod{p},$$

where $0 \leq i \leq (p-1)/m$. Subtracting, we have

$$a(x - y) \equiv i \cdot m \pmod{p}.$$

Since $y < x$, their difference $x - y$ is nonzero and (since p is prime) $x - y$ has an inverse modulo p. That is, there exists a number q such that $(x-y) \cdot q \equiv 1 \pmod{p}$. Multiplying both sides by q, we have

$$a \equiv i \cdot m \cdot q \pmod{p}$$

By definition of our hashing system, are $p-1$ possible choices for a. By varying i in the allowed range, there are $\lfloor (p-1)/m \rfloor$ possible nonzero values for the right-hand side. Thus, the probability of collision is

$$\frac{\lfloor (p-1)/m \rfloor}{p-1} \;\leq\; \frac{(p-1)/m}{p-1} \;=\; \frac{1}{m},$$

as desired.

Like the other randomized structures we have seen this year, universal hash functions are both simple and provide good guarantees on the expected-case performance of hashing systems. We will pick this topic up in our next lecture, focusing on methods for collision resolution, under the assumption that our hashing function has a low probability of collisions.

Lecture 11: Hashing - Handling Collisions

Hashing: In the previous lecture we introduced the concept of hashing as a method for implementing the dictionary abstract data structure, supporting `insert()`, `delete()` and `find()`. Recall that we have a table of given size m, called the *table size*. We select an easily computable *hash function* $h(x)$, which is designed to scatter the keys in a virtually random manner to indices in the range `[0..m-1]`. We then store x (and its associated value) in index $h(x)$ in the table.

In the previous lecture we 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. 61).
- `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.

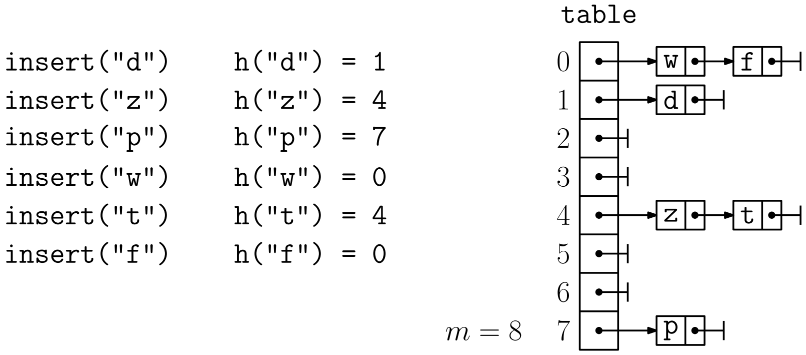


Fig. 61: Collision resolution by separate chaining.

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 λ.

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 λ 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. Assuming that we use universal hashing (see the previous lecture), this uniformity assumption is very reasonable, since the user cannot predict which random hash function will be used. 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. Before discussing them, we should discuss the issue of controlling the load factor.

Controlling the Load Factor and Rehashing: Recall that the load factor of a hashing scheme is $\lambda = n/m$, and the expected running time of hashing operations using separate chaining is $O(1 + \lambda)$. We will see below that other popular collision-resolution methods have running times that grow as $O(\lambda/(1-\lambda))$. Clearly, we would like λ to be small and in fact strictly smaller than 1. Making λ 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. 62). 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.)

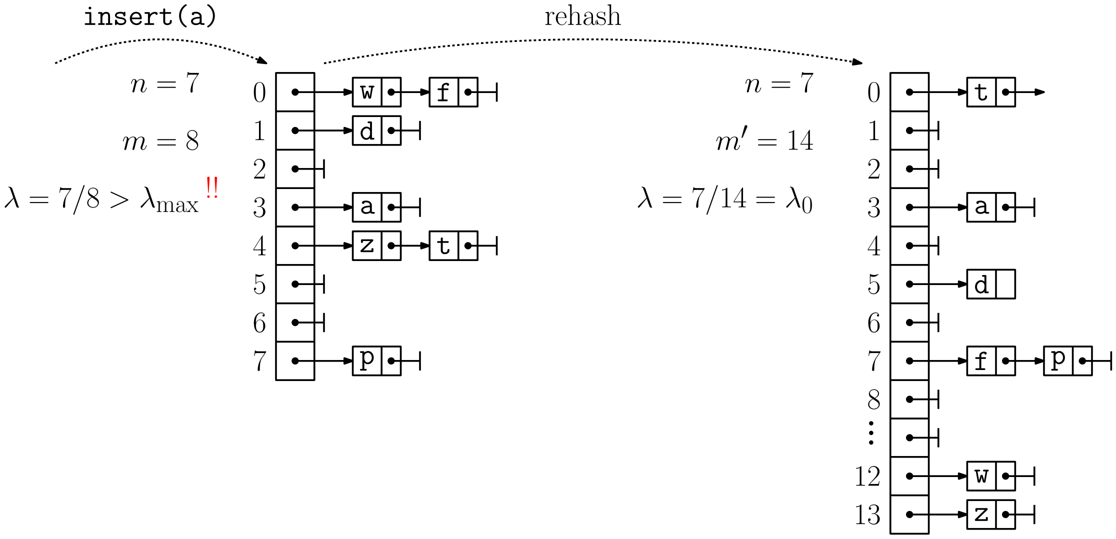


Fig. 62: Controlling the load factor by rehashing, where $\lambda_{\min} = 0.25$, $\lambda_{\max} = 0.75$, and $\lambda_0 = 0.5$.

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.

Amortized Analysis of Rehashing: 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 observe that 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 750 keys against the 375 (cheap) insertions.

Hopefully, this idea will sound familiar to you. In an earlier lecture, we discussed the idea of doubling an array to store a stack. We showed there that by doubling the storage each time the stack overflowed, the amortized cost of each operation is just $O(1)$. There was no magic to doubling. Increasing the storage by any constant factor works, and the same analysis applies here as well. Each time we rehash, we are either increasing or decreasing the hash-table size by a constant factor. Assuming that the hash operations themselves take constant time, we can "charge" the expensive rehashing time to the inexpensive insertions or deletions that led up to the present state of affairs.

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 $2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min})$ *work 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 work 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 work 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 work 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 work 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. 62). 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.

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 63. 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]`.

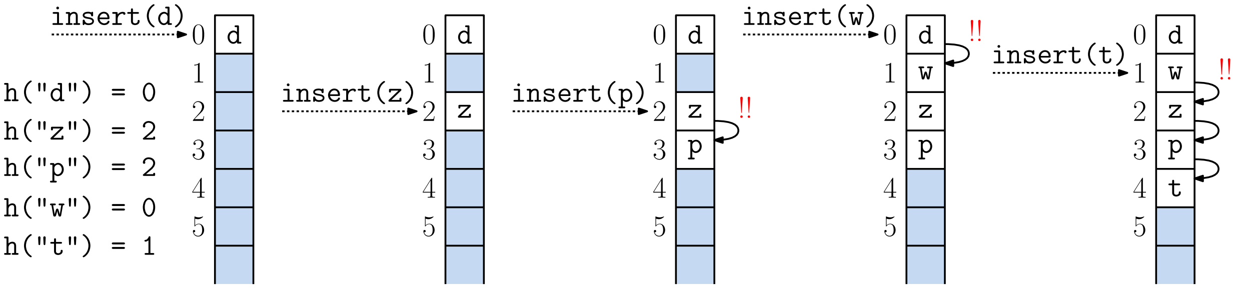


Fig. 63: Linear probing.

This phenomenon is called *secondary clustering*. Primary clustering happens when multiple keys hash to the same location. Secondary clustering happens when keys hash to different locations, but the collision-resolution has resulted in new collisions. Note that secondary clustering cannot occur with separate chaining, because the lists for separate hash locations are kept separate from each other. But in open addressing, secondary clustering is a significant phenomenon. As the load factor approaches 1, secondary 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 secondary clustering is a major shortcoming, and the methods given below do significantly better in this regard.

Quadratic Probing: To avoid secondary 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 $f(i) = i^2$.

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`.

Experience shows that this succeeds in breaking up the secondary clusters that arise from linear probing, but this simple procedure conceals a rather knotty problem. Unlike linear probing, which is guaranteed to try every entry in your table, quadratic probing bounces

Find Operation with Quadratic Probing

```
Value find(Key x) {                     // find x
    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                // next position
        c = c % m                       // wrap around if needed
    }
    return table[c].value               // return associated value (or null if empty)
}
```

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! A more realistic example is when $m = 105$. In this case,

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:

$$i^2 \equiv j^2 \;\Leftrightarrow\; i^2 - j^2 \equiv 0 \;\Leftrightarrow\; (i-j)(i+j) \equiv 0 \pmod{m}$$

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 quadratic probing will succeed in probing every table entry before repeating an entry.
- If the table size m is a power of two, and the increment is chosen to be $\frac{1}{2}(i^2 + i)$ (thus, you probe locations $h(x)$, $h(x) + 1$, $h(x) + 3$, $h(x) + 6$, and so on) then you will succeed in probing every table entry before repeating an entry.

Double Hashing: Both linear probing and quadratic probing have their shortcomings (secondary clustering for the first and short cycles for the second). 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:

$$h(x) + g(x), \quad h(x) + 2g(x), \quad h(x) + 3g(x), \quad \ldots$$

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.

Fig. 64 provides an illustration of how the various open-addressing probing methods work.

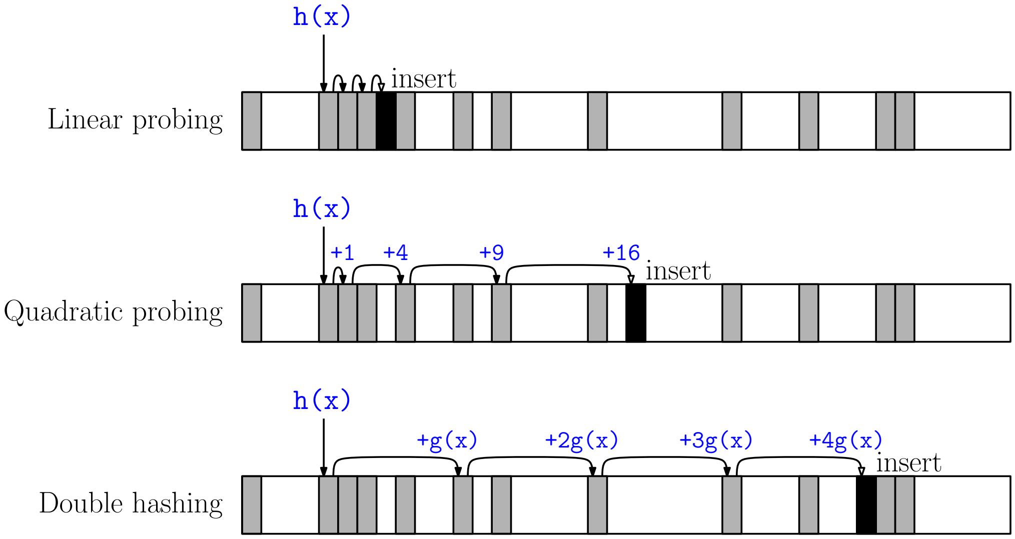


Fig. 64: Various open-addressing systems. (Shaded squares are occupied and the black square indicates where the key is inserted.)

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

$$S_{\mathrm{DH}} = \frac{1}{\lambda}\ln\frac{1}{1-\lambda} \qquad U_{\mathrm{DH}} = \frac{1}{1-\lambda}.$$

To get some feeling for what these quantities mean, consider the following table:

λ	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. 65. 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.

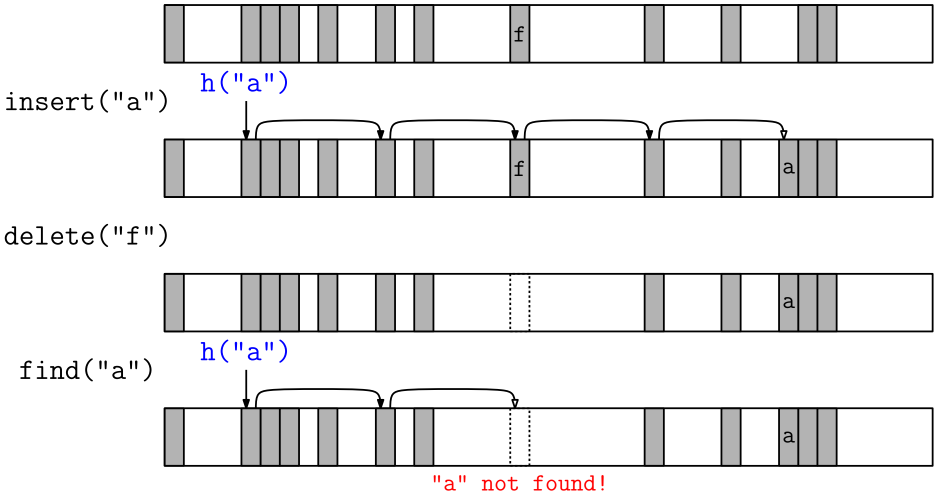


Fig. 65: The problem with deletion in open addressing systems.

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).

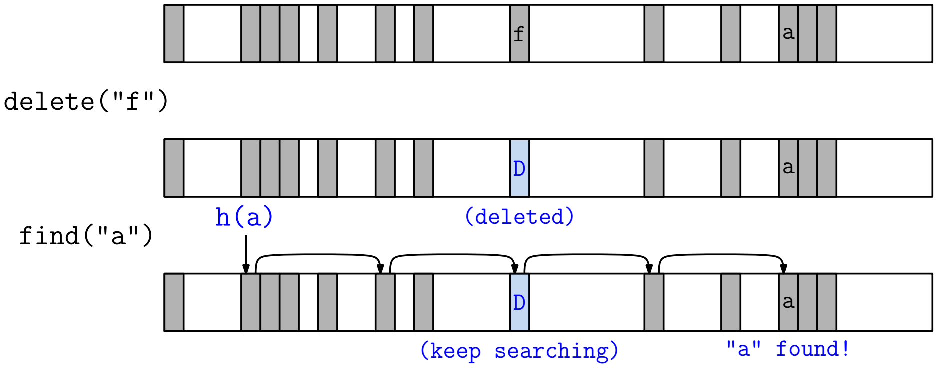


Fig. 66: Deleting in open-addressing by using special *empty* entry.

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 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.

Lecture 12: Extended and Scapegoat Trees

Overview: Today's lecture will focus on two concepts, extended binary search trees and scapegoat trees. (The material on the SG-Tree, which was discussed in class is only covered in the lecture slides.)

Extended Binary Search Trees: Recall from an earlier lecture the concept of an *extended binary tree*, that is, a binary tree whose nodes have either two children or zero children. The former are called *internal nodes* and the latter are called *external nodes*. An example is shown in Fig. 67(a).

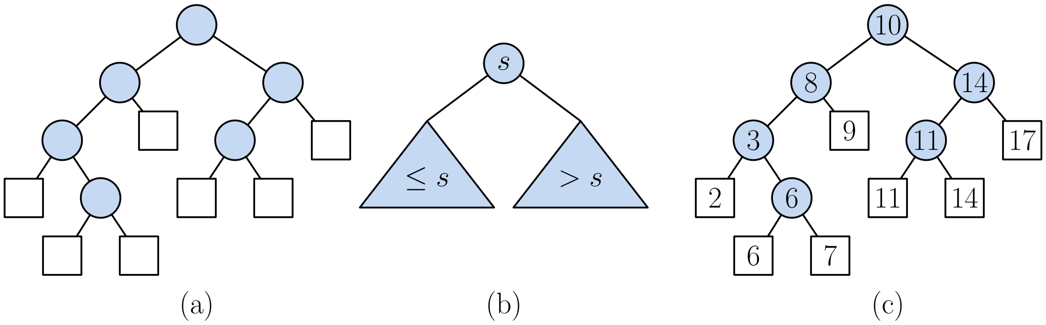


Fig. 67: (a) Extended binary tree, (b) extended binary search tree structure, and (c) extended binary search tree containing the keys $\{2, 6, 7, 9, 11, 14, 17\}$.

As we saw in our discussion of B+ trees in an earlier lecture, it is often useful to employ extended trees in the context of search trees. While B+ trees are multiway trees, we will explore this in the context of binary search trees. The idea is to store all the key-value pairs in just the external nodes. The internal nodes are merely an index structure whose purpose is to allow us to rapidly identify an external node of interest.

More formally, each internal node stores a key s, called a *splitter*, with the property that all external nodes whose key value x is at most s reside in s's left subtree and all external nodes whose key value is strictly greater than s reside within s's right subtree (see Fig. 67(b)). An example of an extended binary search tree is shown in Fig. 67(c).

It is important to note that the tree's *contents* consist stored in the external nodes, not the internal nodes. For example, in Fig. 67(c), the tree's contents are $\{2, 6, 7, 9, 11, 14, 17\}$. The splitter values 3, 8, and 10 appear in internal nodes, but they are not counted among the tree's contents.

Motivation - Multi-dimensional trees: In the context of binary search trees, the advantage of this extended-tree approach is not very obvious. The useful of distinguishing data from splitters becomes more evident when we consider search structures in a multi-dimensional context of *partition trees*.

For example, consider the hierarchical decomposition of space shown in Fig. 68 (left). In this case, the splitters correspond to lines in the plane. Each such line could be expressed as its equation (e.g., $y = ax + b$). The points lying on one side of the line are stored in the left subtree and the points lying on the other are stored in the right subtree. Continuing in this manner, we obtain a data structure called a *binary space partition tree*, or *BSP tree* for short. The key-value pairs in this case are points and whatever additional data we wish to decorate each point with. In this case, it is easy to see that there is a fundamental difference between splitters (line equations) and data (points). Before exploring extended trees in the context of multi-dimensional space, it will be useful to consider them in the simpler 1-dimensional context, which we are familiar with.

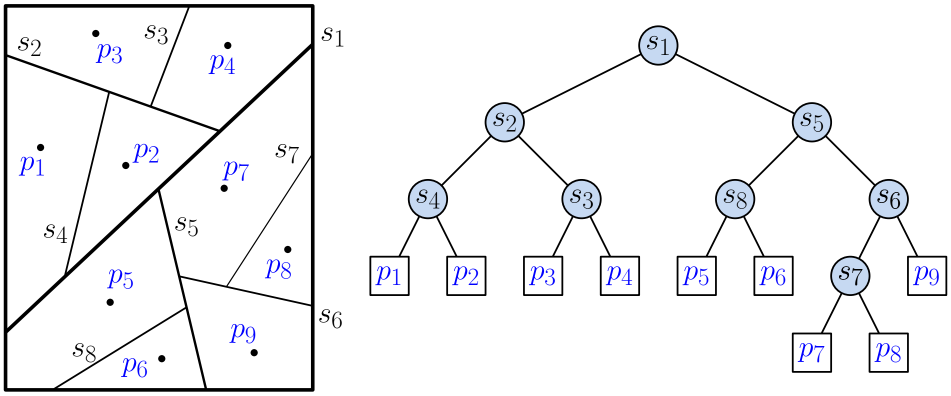


Fig. 68: Binary space partition tree, (a) decomposition of space and (b) the tree structure.

Dictionary Operations on Extended Trees: The dictionary operations that we defined for standard (unbalanced) binary search trees are readily generalized to extended binary search trees.

`find(Key x, Node p)`: The initial call is `find(x, root)`. The procedure operates recursively. If $\texttt{x} \leq \texttt{p.key}$ we recurse on the left subtree, and otherwise we recurse on the right subtree. On arriving to an external node `p`, we test whether $\texttt{x} = \texttt{p.key}$. If so, we report success and otherwise we report failure.

Note that if we encounter an internal node whose key value is equal to `x`, we cannot report success, since the key values in the internal nodes are not reflective of the tree's contents. They are merely an aid to finding the key in the external nodes. For example, on the tree shown in Fig. 67(c), `find(10)` returns false, even though there is an internal node containing `10`. (In this case, the search ends at the external node containing `9`.)

`insert(Key x, Value v, Node p)`: This function returns a reference to the root of the updated subtree where `x` is inserted. The initial call is `root = insert(x, v, root)`.

The procedure operates recursively. We use the same process as in `find` to locate an external node `p`. If there is no such node because the tree is empty, we create a single external node containing `x`, which we return. Otherwise, we check whether `x` = `p.key`, and if so we signal a duplicate-key error. If neither of these happens, we create a new external node containing `x`, and an internal node to split between `x` and `p.key`.

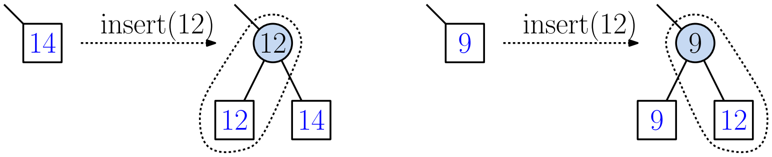


Fig. 69: Inserting a new external node into an extended binary search tree. Note that the internal node is assigned the smaller of the two key values.

More formally, let $y \leftarrow$ `p.key`. Following our convention that the left subtree contains key values that less than or equal to the splitter and the right subtree is strictly greater, the splitter can be any value s such that $\min(x, y) \leq s < \max(x, y)$. We will assume the simple convention of setting $s = \min(x, y)$. We first create a new internal node containing s and a new external node containing key x and value v. Between the two external nodes x and y, we make the smaller the left child of s and the larger is its right child (see Fig. 69). Finally, we return a reference to the internal node containing s, which is stored in the child link of the parent of `p`. (An example is shown in Fig. 70.)

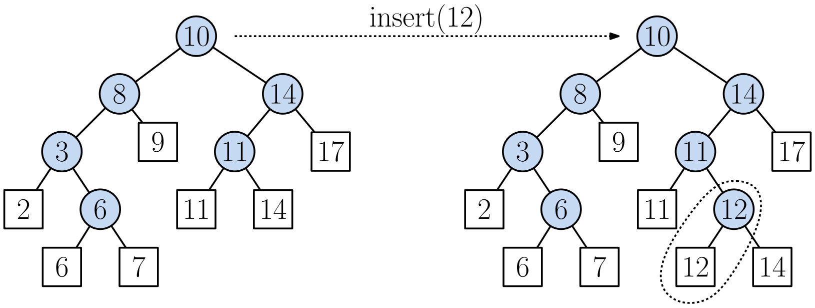


Fig. 70: Inserting a key `12` into an extended binary search tree. A search for `12` leads to the external node `14`. Two nodes are created, one external node containing `12` and one internal node containing the minimum of `12` and `14`.

`delete(Key x, Node p)`: This function returns a reference to the root of the updated subtree from which `x` is deleted. The initial call is `root = delete(x, root)`.

The procedure operates recursively. We use the same process as in `find` to locate the external node `p` that contains `x`. If `x` is not found, we signal an nonexistent-key error. Otherwise, if this external node is the root of the tree, we remove it and return the value `null`. If neither of these occurs, we delete the external node and its internal node parent. We return a reference to the other child of the parent (see Fig. 71).

As with standard (unbalanced) binary search trees, all operations take time proportional to the height of the tree. The height of the tree is (up to a constant additive term) the same in expectation for the extended tree as for the standard tree, namely $O(\log n)$ if n keys are inserted in random order.

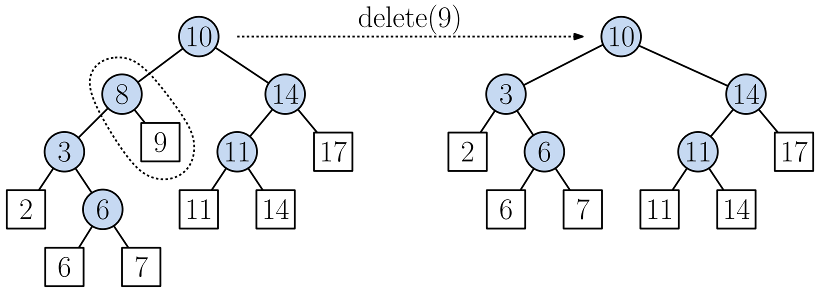


Fig. 71: Deleting a key 9 from an extended binary search tree. After finding the external node containing 9, we remove it and its parent, and link the other child of the parent into the grandparent.

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).

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 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. (One such example is the binary space partition tree shown in Fig. 68.) As we shall see, the scapegoat tree achieves good balance by "rebuilding" subtrees that exhibit poor balance. The trick behind scapegoat trees is figuring out which subtrees to rebuild and when to do this.

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. (Additional information, such as parent pointers may be added to simplify coding, however, but these are not needed.) Nonetheless, it height will always be $O(\log n)$. (Note that this is not the case for splay trees, whose height can grow to as high as $O(n)$.) 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, 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*,[6] and we completely rebuild the subtree rooted at this node so that it is perfectly balanced.

Deletion:

[6] A "scapegoat" is 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.

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

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 external node containing the deleted key is too small, then trigger a rebuilding operation." However, 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 k keys, this operation can be performed in $O(k)$ time. Suppose that p is a pointer to the node of the scapegoat tree whose subtree is to be rebuilt. We begin be performing an inorder traversal of p's subtree, copying the keys to an array $A[0, ..., k-1]$. Because we use an inorder traversal, the elements of A are in ascending sorted order.

To create the new subtree, we will define a procedure that extracts the median element of the array as the root, and then recursively rebuilds the subarrays to the left and right of the median, and then makes the resulting subtrees the left and right children of the median node. More formally, let us define a function `buildSubtree(A, i, k)`, which returns a reference to a balanced subtree containing the k-element subarray of A whose first element is $A[i]$, that is, $A[i, \ldots, i+k-1]$. Pseudocode is given in the code block below.

Building a Balanced Tree from an Array

```
BinaryNode buildSubtree(Key[] A, int i, int k) {
    if (k == 0) return null;                      // empty array
    else {
        int m = ceiling(k/2);                     // root is the median
        BinaryNode p = new BinaryNode(A[i+m]);    // A[i+m] is root
        p.left = buildSubtree(A, i, m);           // A[i..m-1] in left subtree
        p.right = buildSubtree(A, i+m+1, k-m-1);  // A[i+m+1..i+k-1] in right
        return p;                                 // return root of the subtree
    }
}
```

Ignoring the recursive calls, we spend $O(1)$ time within 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 number of keys in the tree. The second, denoted by m, is an upper bound on the size of the tree. This latter value plays a role in deciding when the rebalance the tree when deletions are performed. In particular, whenever we insert a key, we increment m, but whenever we delete a key we do not decrement m. Thus, $m \geq n$ and 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 will 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.

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

`delete(Key x)`: This operates exactly the same as deletion in a standard binary search tree. After the deletion, we decrement n, but we do not change m. As mentioned above, if $m > 2n$, we rebuild the entire tree, and set $m \leftarrow n$.

`insert(Key x, Value v)`: The begins exactly as insertion does for a standard binary search tree. But, 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.) Increment both n and m. 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. Let `u` be the current node that is visited, and let `u.child` be the child of `u` that lies on the search path.
- Let `size(u)` denote the *size* of the subtree rooted at `u`, that is, the number of nodes in this subtree. If

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

then rebuild the subtree rooted at `u` (e.g., using the method described above).

The fact that a child has over 2/3 of the nodes of the entire subtree intuitively means that this subtree has roughly speaking 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." An example is shown in Fig. 72.

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. It follows that if p is at depth k in the tree, then

$$\text{size}(p) \geq \left(\frac{2}{3}\right)^k n.$$

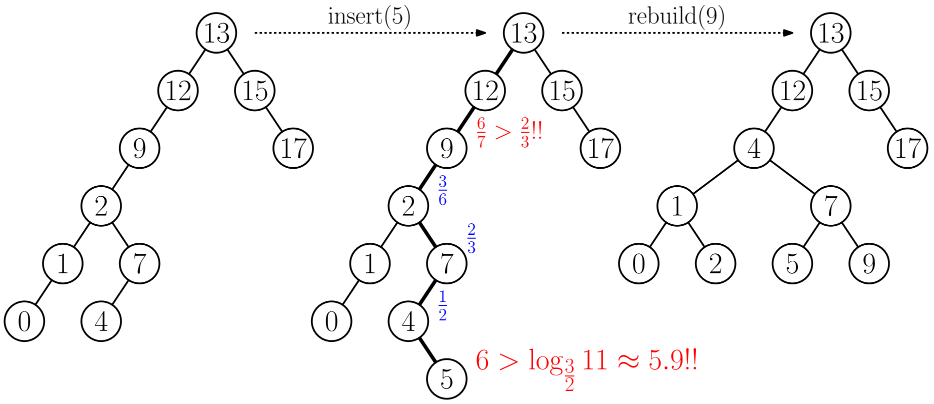


Fig. 72: Inserting a key into a scapegoat tree, which triggers a rebuilding event. The node containing 9 is the first scapegoat candidate encountered while backtracking up the search path and is rebuilt.

We know that size$(p) \geq 1$ (since the subtree contains p itself, if nothing else), so it follows that $1 \geq (2/3)^k n$. With some simple manipulations, we have

$$\left(\frac{3}{2}\right)^k \leq n,$$

which implies that $k \leq \log_{3/2} n$. 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.

How to Compute Subtree Sizes? 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?

Unfortunately, there is no clever way to do this efficiently (say in $O(\log n)$ time). 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 `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 accounted for in the cost of the rebuilding process, and hence it essentially comes for free!

By the way, there is an alternative method for computing sizes. This is to store the size value of each node explicity 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 k dictionary operations (find, insert, and delete) to a scapegoat tree can be performed in time $O(k \log k)$.

Proof: (Sketch)

- 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 a tree due to deletions, at least half the entries since the last full rebuild must have been deleted. By token-based analyses (recall stacks and rehashing from earlier lectures), it follows that the $O(n)$ cost of rebuilding the entire tree can be amortized against the time spent processing the deletions.
- Insert: This is analyzed by a potential argument. Intuitively, after any subtree of size k is rebuilt it takes $O(k)$ insertions to force this subtree to be rebuilt again. Charge the rebuilding time against these "cheap" insertions.

Corollary: The amortized complexity of the scapegoat tree with at most n nodes is $O(\log n)$.

Lecture 13: 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 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. 73(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. 73(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. 73(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. 73(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.

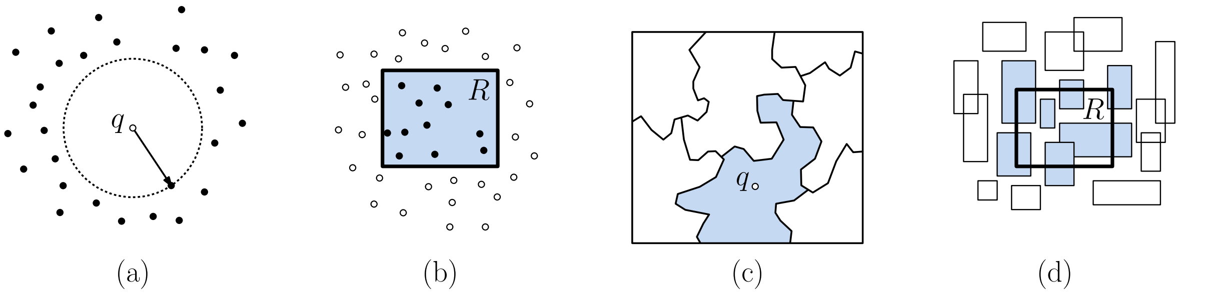


Fig. 73: Common geometric queries: (a) nearest-neighbor searching, (b) range searching, (c) point location, (d) intersection searching.

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 ith 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 `x.get(i)` to get the ith coordinate of a point x, we will often be lazy in our code samples, and just write `x[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:

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

The resulting subdivision is shown in Fig. 74(a) and the tree structure is shown in (b).

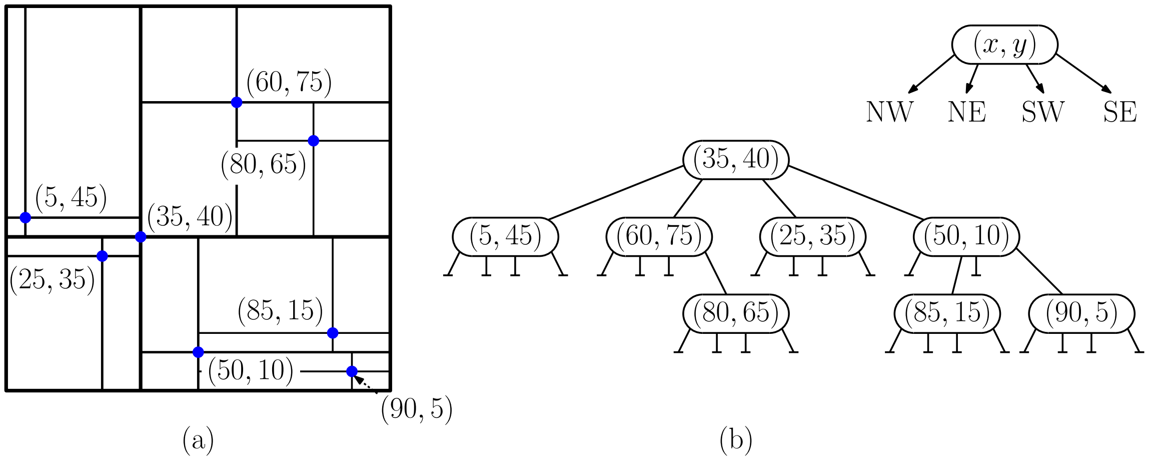


Fig. 74: Point quadtree.

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 rectangle (assuming the planar case) or a hyper-rectangle in d-dimensional space. When a new point is inserted into some node (equivalently into some cell), we will split the cell by a horizontal or vertical *splitting line*, which passes through this point. In higher dimensions, we split the cell by a $(d-1)$ dimensional hyperplane that is orthogonal to one of the coordinate axes. In any dimension, such a split can be specified by giving the *cutting axes* (which can be represented as an integer from 0 to $d-1$), and also called the *cutting value*. Following the approach used in point quadtrees, the cutting value will be taken from the coordinates of the point being stored in this node. Thus, along with its left and right child pointers, we can think of every node as storing two items, an integer cutting dimension and a point. The following code shows a possible node structure. We add a utility method to determine whether a point lies in the left subtree, that is, whether it is smaller along the cutting dimension. Of course, if it is in the right subtree, this returns false. Throughout, we will use the terms "left" and "right" to by synonymous with being smaller than or larger than the splitter along the cutting dimension, respectiely.

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

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

    boolean inLeftSubtree(Point x) {    // is x in left subtree?
        return x[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 is the cutting dimension chosen? There are a number of ways, but the most common is just to alternate 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. In general, we cycle through the various axes, setting the cutting dimension of the child to 1 plus the cutting dimension of the parent all taken modulo the dimension. An example is shown in Fig. 75 (given the same points as in the previous example). Again we show both the tree and the spatial subdivision. We will assume this method of choosing the cutting dimension in our examples, but there are better ways, for example selecting the cutting dimension based on the direction in which the points of the subtree are most widely distributed.

The tree representation is much the same as it is for quad-trees except now every node has only two children. The contents of the left child of a node contain the points x such that `x[cutDim] < point[cutDim]` and for the right child `x[cutDim] >= point[cutDim]`. (How you break ties is rather arbitrary.)

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 x being inserted, the current node p, and the cutting dimension of the newly created node. The initial call is `root = insert(x, root, 0)`.

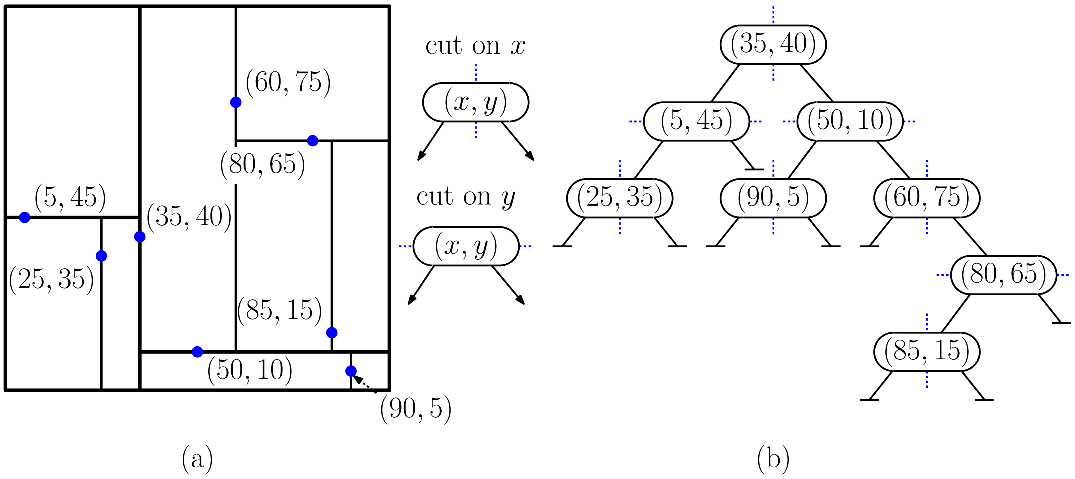


Fig. 75: Point kd-tree decomposition.

kd-tree Insertion

```
KDNode insert(Point x, KDNode p, int cutDim) {
    if (p == null) {                                // fell out of tree
        p = new KDNode(x, cutDim);                  // create new leaf
    } else if (p.point.equals(x)) {
        throw Exception("duplicate point");         // duplicate data point!
    } else if (p.inLeftSubtree(x)) {                // insert into left subtree
        p.left = insert(x, p.left, (p.cutDim + 1) % x.getDim());
    } else {                                        // insert into right subtree
        p.right = insert(x, p.right, (p.cutDim + 1) % x.getDim());
    }
    return p;
}
```

An example is shown in Fig. 76, where we insert the point $(50, 90)$ into the kd-tree of Fig. 75. 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`).

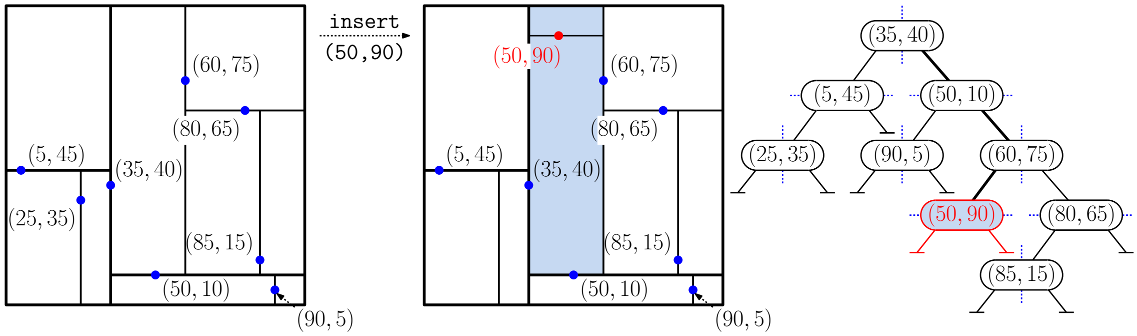


Fig. 76: Insertion into the point kd-tree of Fig. 75.

Deletion and Replacement Points: We will next discuss deletion from kd-trees. As we saw with deletion in standard bindary 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 some node `p`, and suppose further that `p.cutDim == 0`, that is, the x-coordinate. 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, cutDim)` that computes the point in the subtree rooted at p that has the smallest ith coordinate. The procedure operates recursively. When we arrive at a node p, if the cutting dimension matches i, then we know that the subtrees are ordered according the the ith coordinate. If the left subtree is nonempty, then the desired point is the minimum from this subtree. Otherwise, we return p's associated point. (There is never a need to search the right subtree. Do you see why?)

On the other hand, if p's cutting dimension differs from i then we cannot infer which subtree may contain the point with the minimum ith coordinate. So, we will have to try the right subtree, the left subtree, and the point at this node. We include a function `minAlongDim(p1, p2, i)`, which returns whichever point p_1 or p_2 is smallest along coordinate i. (The function is written so that the second point might be null, which happens when we attempt to extract the minimum point from an empty subtree.) The function is given in the code block below.

Fig. 77 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.

Find the minimum point in subtree along ith coordinate

```
Point findMin(KDNode p, int i) {                        // get min point along dim i
    if (p == null) {                                    // fell out of tree?
        return null;
    }
    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
        Point q = minAlongDim(p.point, findMin(p.left, i), i);
        return minAlongDim(q, findMin(p.right, i), i);
    }
}

Point minAlongDim(Point p1, Point p2, int i) {          // return smaller point on dim i
    if (p2 == null || p1[i] <= p2[i])                   // p1[i] is short for p1.get(i)
        return p1;
    else
        return p2;
}
```

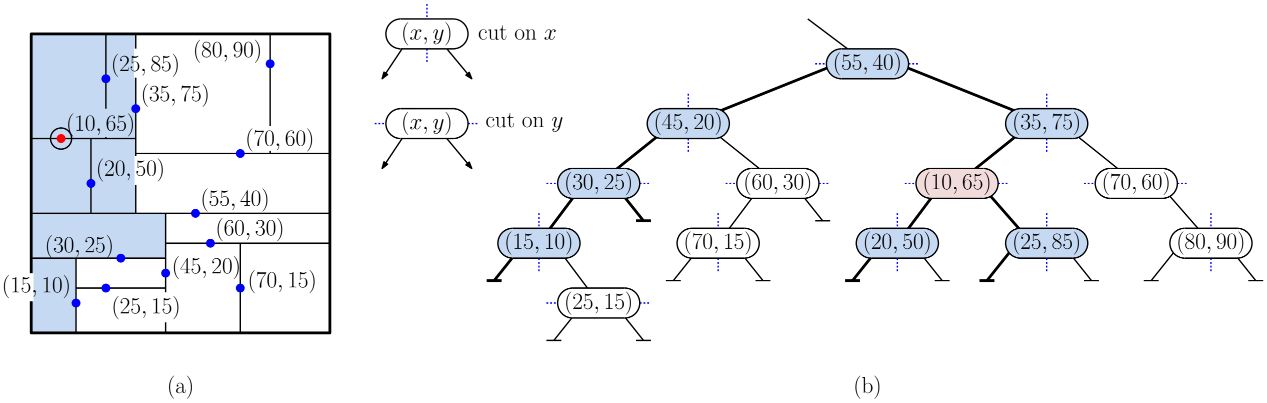


Fig. 77: Example of `findMin` when $i = 0$ (the x-coordinate) on the subtree rooted at $(55, 40)$. The function returns $(10, 65)$.

Deletion from a kd-tree: As with insertion we can model the deletion code after the deletion code 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 element. 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 for some implementations of the kd-tree, where the cutting dimension is selected by some other policy, but to keep things as clean as possible, our deletion procedure will not alter a node's cutting dimension.)

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? What does it mean to find the "smallest" element in a such a set? 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 key is the point with the smallest x-coordinate in the right subtree. We use the `findMin()` function (given 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. There is a clever trick for getting around this though. For the replacement element we will select the *minimum* (not maximum) point from the left subtree, and we move the left subtree over and becomes the new right subtree. The left child pointer is set to null. The code is given in the following code block.

An example of the operation of this deletion algorithm is presented in Fig. 78. 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, and 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, and then recursively delete $(60, 10)$ from p's left subtree. It is a leaf, so it may simply be unlinked from the tree. Finally we move this left subtree p over to become p's right subtree. (Whew!)

Recall that in the 1-dimensional case, in the 2-child case the replacement node was guaranteed to have either zero or one child. However this is not necessarily the case here. Thus we may do many 2-child deletions. As with insertion the running time is proportional to the height of the tree.

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

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

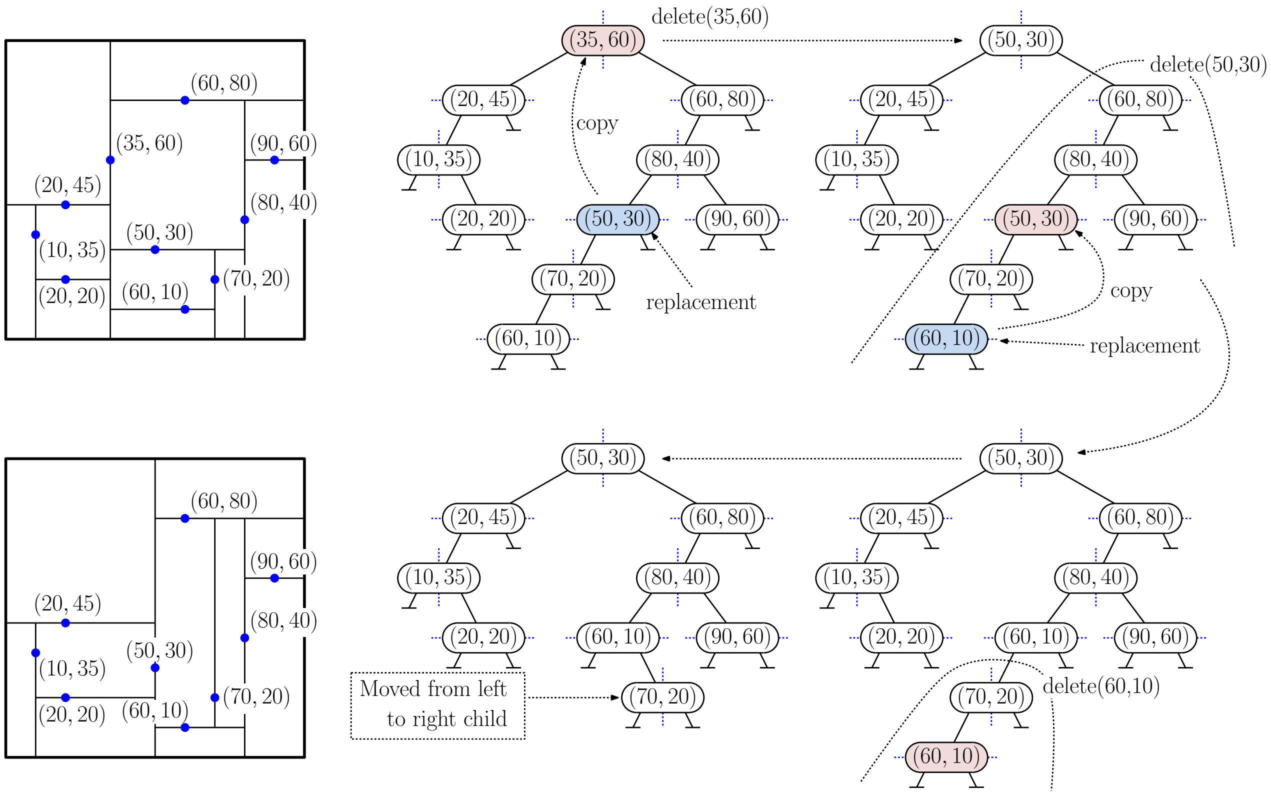


Fig. 78: Deletion from a kd-tree.

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. Because we chose replacement nodes in a biased way (always from the right subtree, if it is nonempty), it is reasonable that the same systematic bias issues that lead to $O(\sqrt{n})$ tree height over a long series of random insertions and deletions. However, to the best of my knowledge, no systematic studies have been published on this topic.

With 1-dimensional search trees we never discussed the question of building a tree in a purely static context, where the points are all given in advance. (The reason is that there is no need to build a tree. You could simply sort the points and store them in an array.) However, in multi-dimensional data sets, binary search in an array is not an option. Therefore, the question of how to build a well-balanced tree for a fixed set of points is a natural one to ask. 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.)

Lecture 14: 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 two important geometric queries using kd-trees: orthogonal range search queries and nearest-neighbor 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 point of the database is associated with the medical record of a patient. Each coordinate is the numeric value of some statistic, such as the patient's height, weight, blood pressure, HDL and LDL cholesterol numbers, etc. So, if there are 20 different numbers associated with each patient's record, each patient can be modeled as a point in a 20-dimensional space of real numbers, or $\mathbb{R}^{20}$ for short.

Suppose that as part of your study or these patients, you want to know information such as "how many patients are there with weights in the range 70–80 kilograms, heights in the range 160–170 centimeters, etc." This amounts 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. 79). This is where the name *orthogonal range searching* originates.

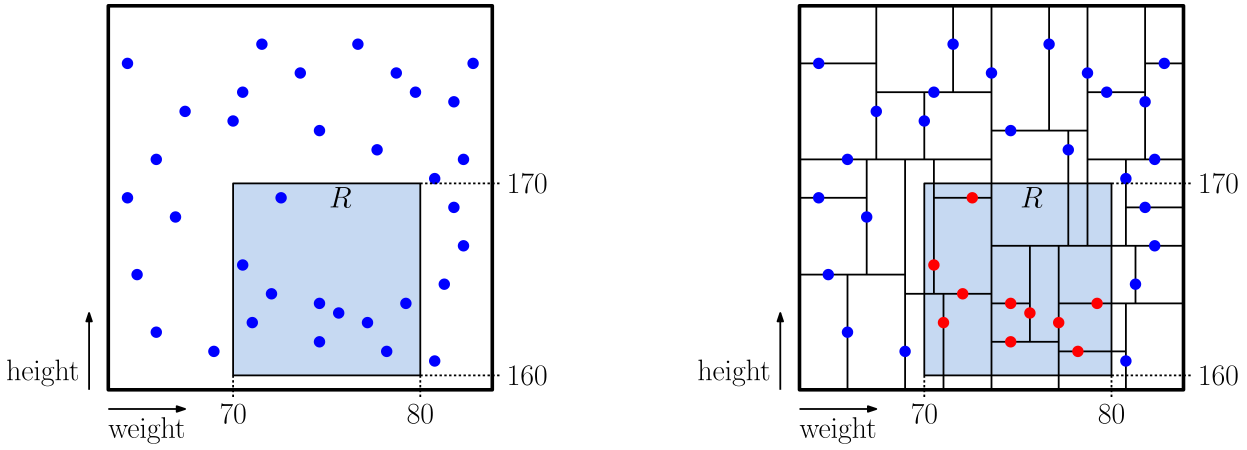


Fig. 79: Orthogonal range query.

More formally, given a set P 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 P 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 for storing a multi-dimensional rectangle, or *hyper-rectangle* for short. The private data consists of two points `low` and `high`. A point q lies within the rectangle if $\text{low}[i] \le q[i] \le \text{high}[i]$, for $0 \le i \le d-1$ (assuming Java-like indexing). In addition to a constructor, the class provides a few useful geometric primitives (illustrated in Fig. 80).

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

`boolean contains(Rectangle 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:

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

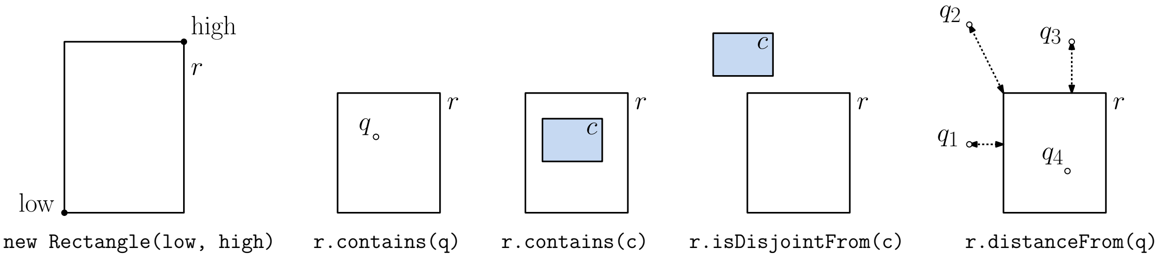


Fig. 80: An axis-parallel rectangle methods.

`boolean isDisjointFrom(Rectangle 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

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

`float distanceTo(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 ith defining interval, taking the sums of squares of these distances, and then taking the square root of this sum:

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

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.

`Rectangle 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. 80). 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].

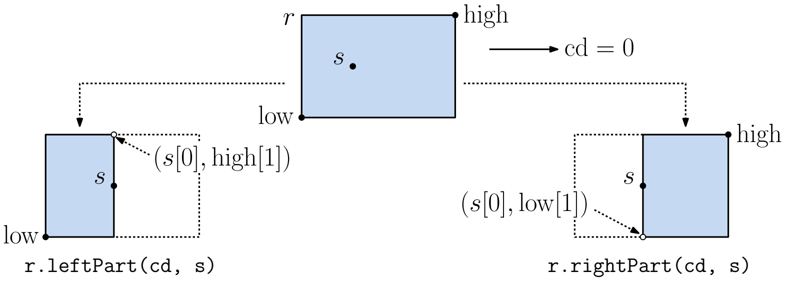


Fig. 81: The functions `leftPart` and `rightPart`.

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

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

A skelton of a simple Rectangle class

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

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

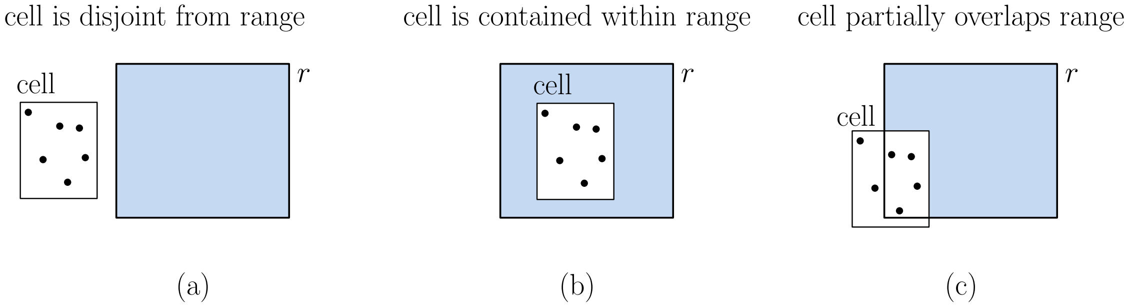
recursively. The first argument `r` is the range itself, the second argument `p` is the node currently visited, and `cell` is its associated cell. It returns a count of the number of points within `p`'s subtree that lie within `r`. The initial call is `rangeCount(r, root, boundingBox)`, where `boundingBox`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

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 is completely disjoint from the query range, we may return 0, because none of this node's points lie within the range (see Fig. 82). 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`. Finally, the range must partially overlap 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(Rectangle r, KDNode p, Rectangle cell) {
    if (p == null) return 0;            // empty subtree
    else if (r.isDisjointFrom(cell))    // no overlap with range
        return 0;
    else if (r.contains(cell))          // range contains our entire cell?
        return p.size;                  // include all points in the count
    else {                              // range partially overlaps cell
        int count = 0;
        if (r.contains(p.point))        // consider this point
            count++;
                                        // 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. 83 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 subtree rooted at h is entirely contained within the range, and any points in the subtree can be


cell is disjoint from range
cell
r
cell is contained within range
cell
r
cell partially overlaps range
cell
r

(a) (b) (c)

Fig. 82: Cases arising in orthogonal range searching.

safely included in the count. (In this case, this includes the two points p and h.) The subtrees rooted at k and g 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 two points of h's subtree, the total count returned is 6.

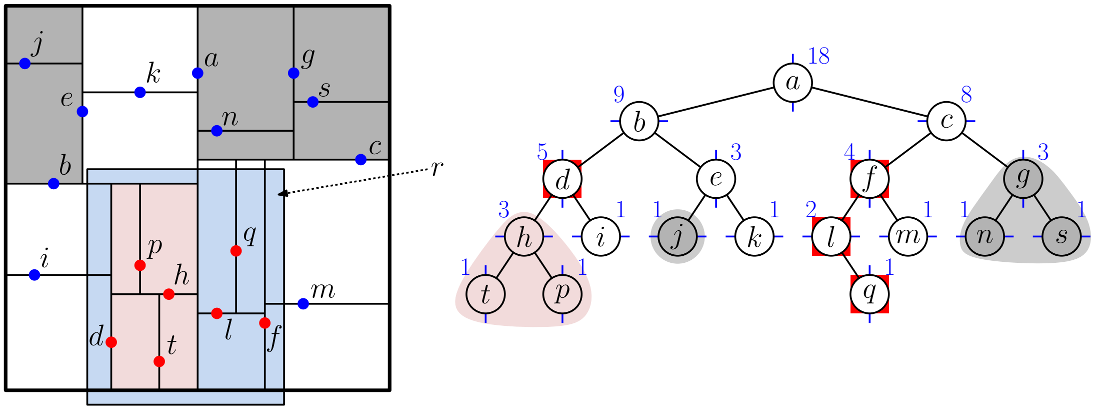


Fig. 83: Range search in kd-trees. The subtree rooted at h is counted entirely. The subtrees rooted at k and g are excluded entirely. The other points are checked individually.

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, 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 cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. 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, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

Proof: Since the tree is balanced, its height is $O(\log n)$. Since the constant factor will not really matter, it will simplify matters to assume that the height is exactly $\lg n$. Let us consider the case of a vertical line $x = x_0$. The horizontal case is symmetrical.

Consider a processed node which has a cutting dimension along x. The vertical line $x = x_0$ either stabs the left child or the right child but not both. If it fails to stab one of the children, then it cannot stab any of the cells belonging to the descendents of this child either. If the cutting dimension is along the y-axis (or generally any other axis in higher dimensions), then the line $x = x_0$ stabs both children's cells.

Since we alternate splitting on left and right, this means that after descending two levels in the tree, we may stab at most two of the possible four grandchildren of each node. (This is illustrated in Fig. 84.) In general each time we descend two more levels we double the number of nodes being stabbed. Thus, we stab the root node, at most 2 nodes at level 2 of the tree, at most 4 nodes at level 4, 8 nodes at level 6, and generally at most 2^i nodes at level $2i$.

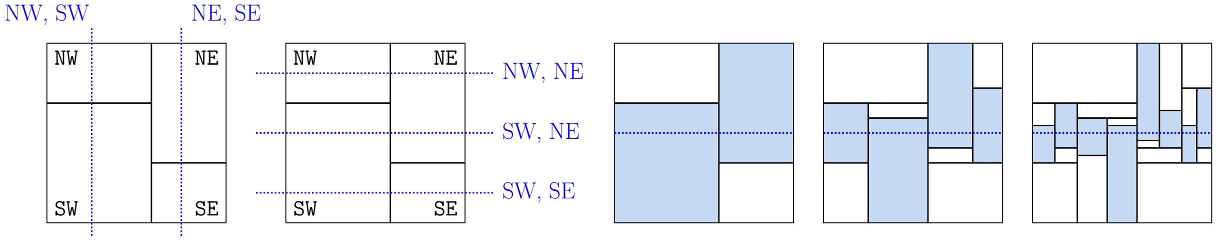


Fig. 84: An axis-parallel line in 2D can stab at most two out of four cells in two levels of the kd-tree decomposition. In general, it stabs 2^i cells at level $2i$.

Because we have an exponentially increasing number, the total sum is dominated by its last term. Thus, it suffices to count the number of nodes stabbed at the lowest level of the tree. If we assume that the kd-tree is balanced, then the tree has height of $h \approx \lg n$ (up to constant factors). The number of leaf nodes processed at the bottommost level is

$$2^{h/2} \approx 2^{(\lg n)/2} = (2^{\lg n})^{1/2} = n^{1/2} = \sqrt{n}.$$

This completes the proof.

We have shown that any vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. Thus, if we were to extend the four sides of Q into lines, the total number of cells stabbed by all these lines is at most $O(4\sqrt{n}) = O(\sqrt{n})$. Thus the total number of cells stabbed by the query range is $O(\sqrt{n})$, and hence the total query time is $O(\sqrt{n})$. Again, this assumes that the kd-tree is balanced (having $O(\log n)$ depth). If the points were inserted in random order, this will be true on average.

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 P stored in a kd-tree, and a query point q, and we want to return the point of P that is closest to q. Let's assume that distances are measured using Euclidean distances. In particular, given two points $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$, their Euclidean distance is

$$\text{dist}(p, q) \;=\; \sqrt{(p_1 - q_1)^2 + \cdots + (p_d - q_d)^2}.$$

Generalizations to other sorts of distance functions (e.g., the Manhattan or taxicab distance) is also possible. An example is shown in Fig. 85. Observe that the circle centered at q and passing through its nearest neighbor p 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 p 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.

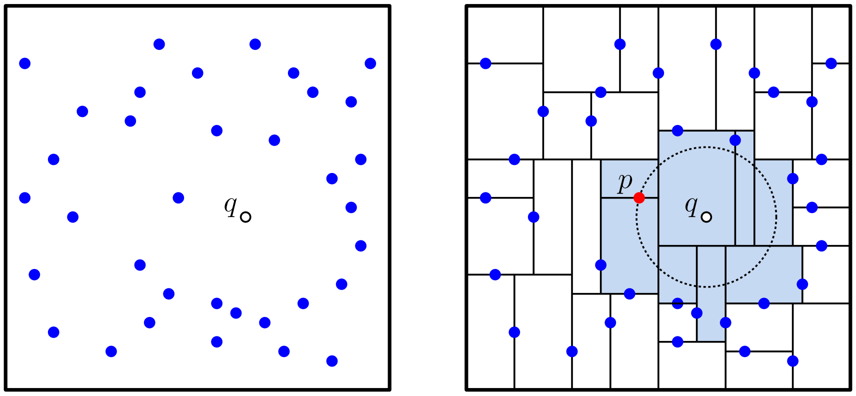


Fig. 85: Nearest-neighbor searching using a kd-tree.

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. 86, 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.

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

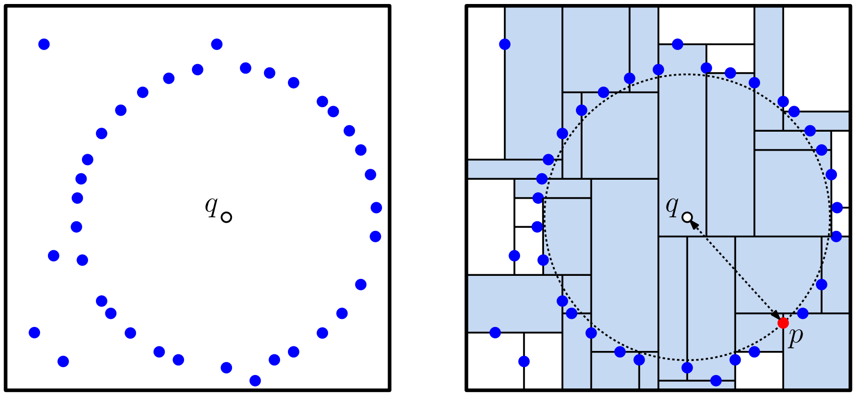


Fig. 86: A (nearly) worst-case scenario for nearest-neighbor searching. Almost all the nodes of the tree need to be visited, since any might be the nearest neighbor.

general, the rectangle consists of all points q such that $\text{low}_i \leq q_i \leq \text{high}_i$ (see Fig. 80(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 actually only returns the distance to the nearest neighbor, but it is an easy matter to modify the code to produce both the distance and the point achieving this distance. (Think about how you would do this.) As usual, we employ a recursive utility function that works on an individual node `p` of the tree. The function `nearNeighbor(q, p, cell, bestDist)` 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 smallest distance, `bestDist`, between `q` and any point seen so far in the search.

The procedure works as follows:

- First, if `p` is `null`, we must have fallen out of the tree, and we just return the current smallest distance, `bestDist` as the answer.
- Otherwise, we compute the distance from the point `p.point` to `q`, and update the `bestDist` value if this point is closer than the previous.
- 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 (see Fig. 87(a)).
  - Next, we check which side `p.point` the query point lies. The closer child of `p` is the one that lies on the same side of the splitter as `q` does.
  - We visit the closer subtree first (see Fig. 87(b)), since it is more likely to yield the nearest neighbor. The value of `bestDist` will be updated to the closest point seen so far.
  - After returning from this call, we compute `q`'s distance to the right subtree cell. Observe that if this distance is greater than `bestDist`, there is no chance that the other subtree contains the nearest neighbor, and so there is no need to visit

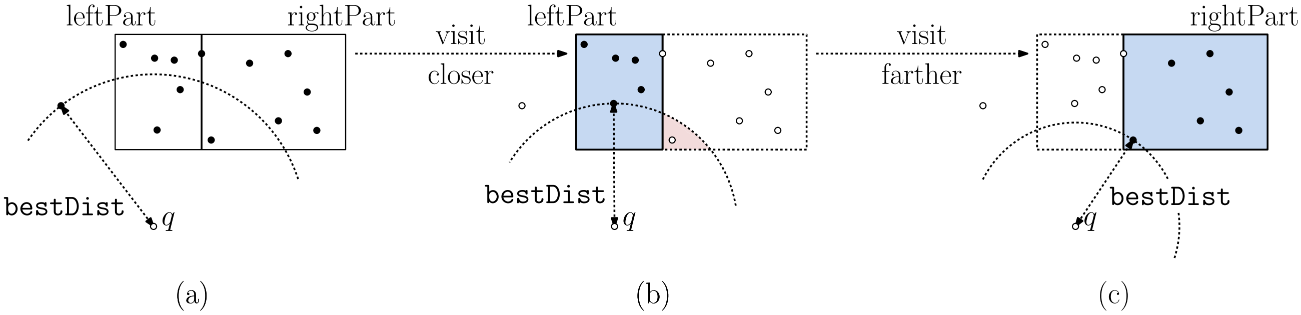


Fig. 87: Nearest-neighbor searching.

this subtree. Otherwise, we apply the search recursively to the right subtree (see Fig. 87(c)) and update `bestDist` accordingly.

Given a query point q, the initial call is `nearNeigh(q, root, rootCell, Float.MAX_VALUE)`, where `rootCell` is the rectangle that encloses the entire tree contents, and `Float.MAX_VALUE` is the maximum possible float value. The code is presented below.

Compute distance to nearest neighbor in kd-tree

```
float nearNeighbor(Point q, KDNode p, Rectangle cell, float bestDist) {
    if (p != null) {
        float thisDist = q.distanceTo(p.point);             // distance to p's point
        bestDist = Math.min(thisDist,  bestDist);           // keep smaller distance

        int cd = p.cutDim;                                  // cutting dimension
        Rectangle leftCell = cell.leftPart(cd, p.point);    // left child's cell
        Rectangle rightCell = cell.rightPart(cd, p.point);  // right child's cell

        if (q[cd] < p.point[cd]) {                          // q is closer to left
            bestDist = nearNeighbor(q, p.left, leftCell, bestDist);
            if (rightCell.distanceTo(q) < bestDist) {       // worth visiting right?
                bestDist = nearNeighbor(q, p.right, rightCell, bestDist);
            }
        } else {                                            // q is closer to right
            bestDist = nearNeighbor(q, p.right, rightCell, bestDist);
            if (leftCell.distanceTo(q) < bestDist) {        // worth visiting left?
                bestDist = nearNeighbor(q, p.left, leftCell, bestDist);
            }
        }
    }
    return bestDist;
}
```

An example of the algorithm in action is shown in Fig. 88. 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 the closest, and its distance to q defines `bestDist`. Because the lower child of $(70, 30)$ overlaps the ball of radius `bestDist`, 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 ball of radius `bestDist`. 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 ball of radius `bestDist`, 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 `bestDist` 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 even closer to the query point, and thus `bestDist` is again reduced. After this, all the remaining cells (shaded in white in the figure) lie outside the nearest-neighbor ball, and so we can terminate the search.

Analysis: How efficient is this procedure? It is quite difficult to analyze from the perspective of its worst-case performance, because as seen in Fig. 86, there are cases where we may need to visit almost every node of the tree, because almost all the points are equidistant from the query point. 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 15: 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 over the next couple of lectures because there are a number interesting data structures issues that arise. In addition, 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.

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. Variables allocated through `new` 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

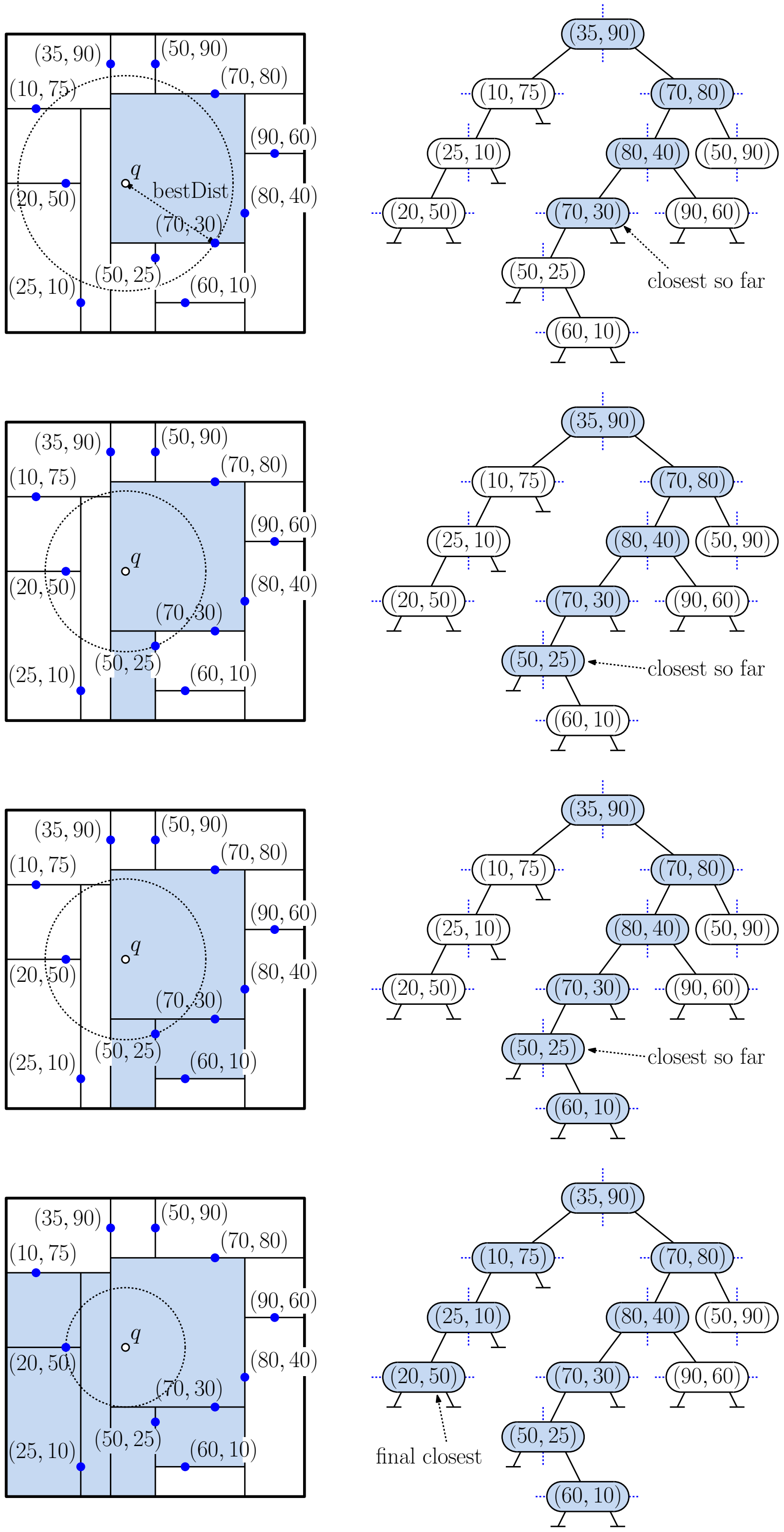


Fig. 88: Nearest-neighbor search.

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.

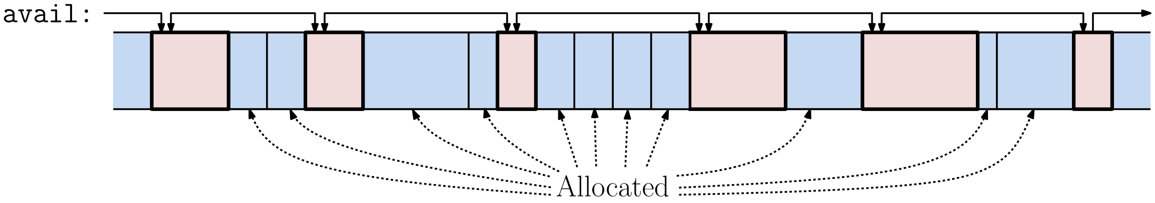


Fig. 89: Dynamic memory allocation block structure.

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. 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).

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.)

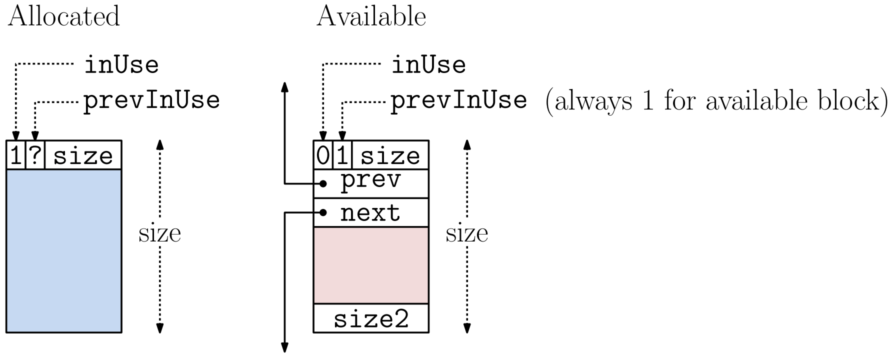


Fig. 90: Block structure for dynamic storage allocation.

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. 91). 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.

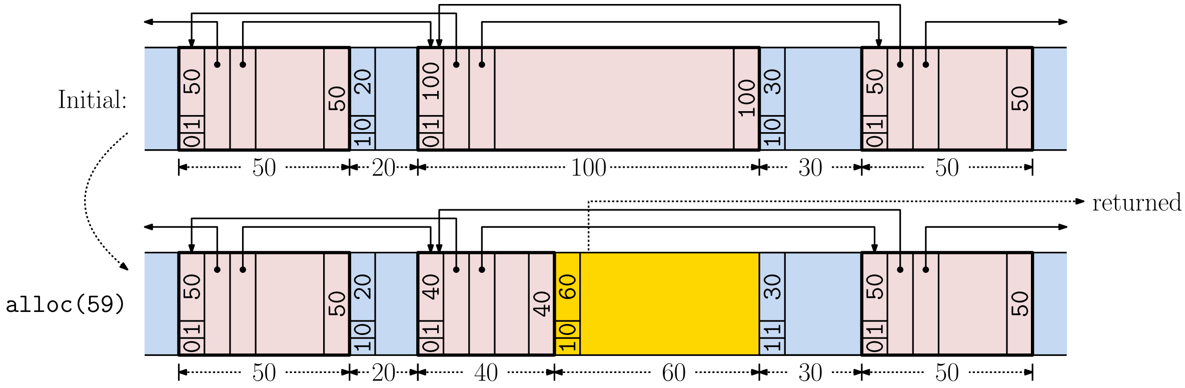


Fig. 91: An example of block allocation.

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. 91. 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. 92, 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.

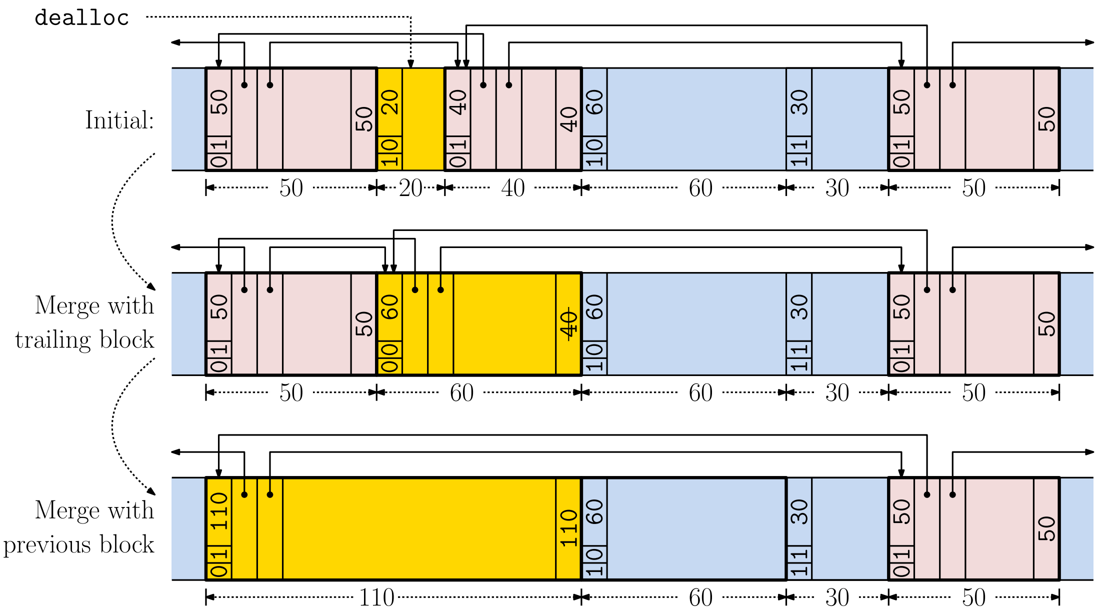


Fig. 92: An example of block deallocation and merging.

- If the preceding block is not in use, we merge this block with p (see Fig. 92, 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.

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.

Deallocating a block of storage

```
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
     }
}
```

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. 94). 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. 94 below illustrates a buddy system allocation of blocks, where the blocks of size 2^k are shown at the same level. Available blocks are shown in white and allocated blocks are shaded. 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.

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

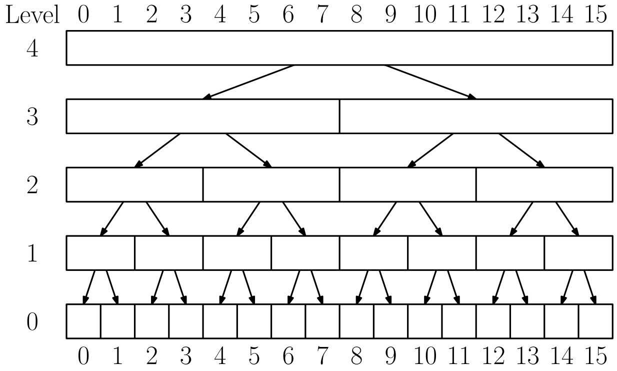


Fig. 93: Buddy system block structure.

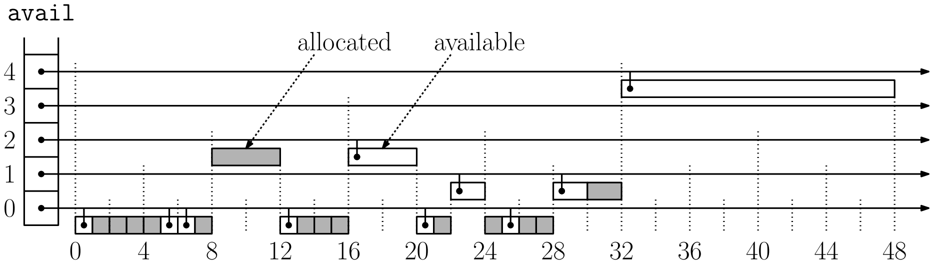


Fig. 94: Buddy system example. Allocated blocks are shaded. Available blocks are shown in white and are linked into the available-block list of the appropriate size.

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.

$$\begin{aligned} 80 &= \texttt{1010000}_2 \\ 88 &= \texttt{1011000}_2. \end{aligned}$$

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. 95 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.

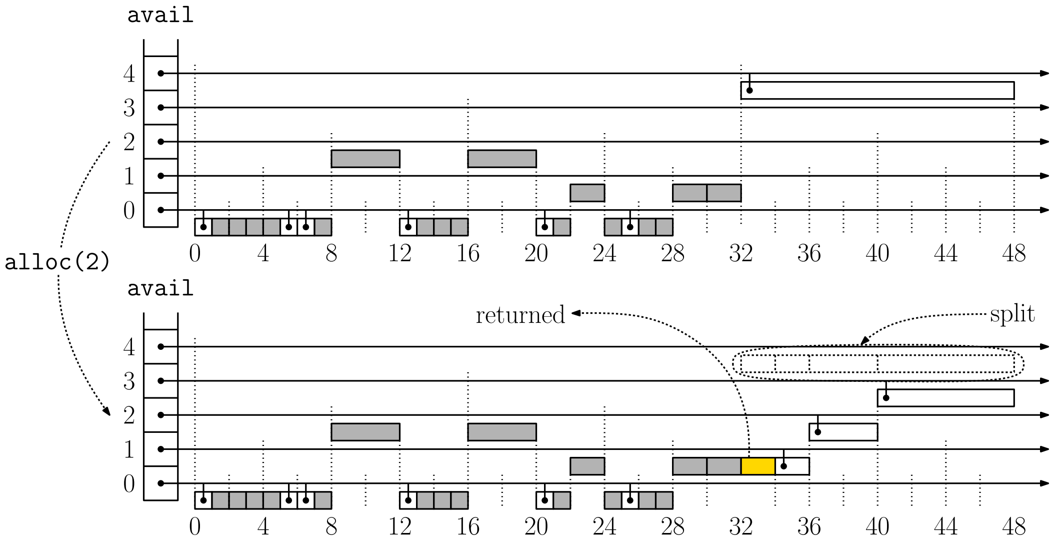


Fig. 95: Example of allocating a block in the buddy system.

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. 96 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.

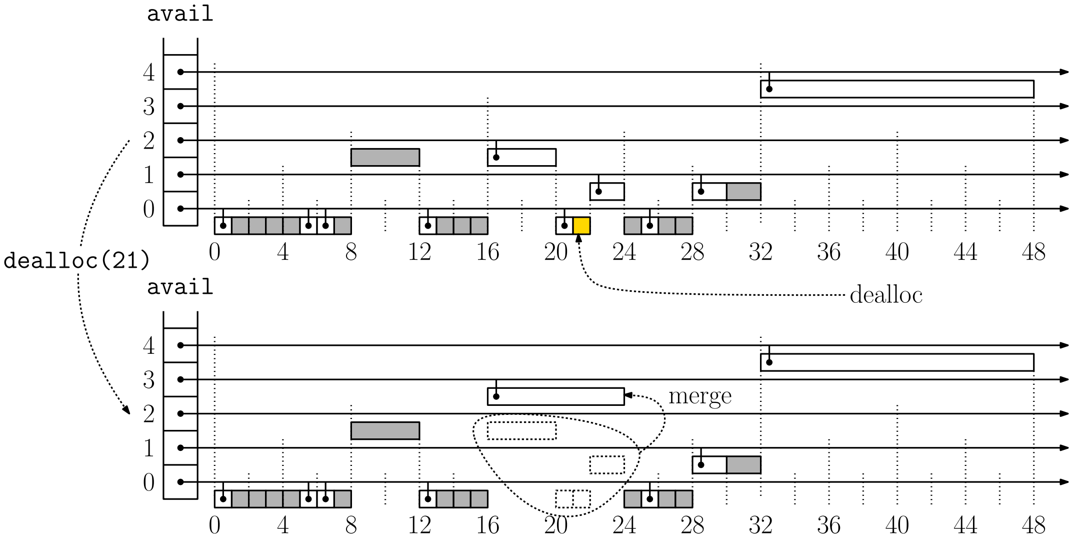


Fig. 96: Example of deallocating a block in the buddy system.

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 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.

Suuplemental Lectures

Supplemental Lecture 1: Mathematical Prelimaries

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

$$\begin{aligned}
\sum_{i=1}^{n} 1 &= n \qquad \text{(The constant series)}\\
\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} \qquad \text{(The arithmetic series)}\\
\sum_{i=1}^{n} \frac{1}{i} &= \ln n + O(1) \qquad \text{(The harmonic series)}\\
\sum_{i=0}^{n} c^i &= \frac{c^{n+1}-1}{c-1} \qquad c \neq 1 \qquad \text{(The geometric series)}
\end{aligned}$$

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: Leftist and Skew Heaps

Leftist Heaps: The standard binary heap data structure is an simple and efficient data structure for the basic priority queue operations `insert(x)` and `x = extractMin()`. It is often the case in data structure design that the user of the data structure wants to add additional capabilities to the abstract data structure. When this happens, it may be necessary to redesign components of the data structure to achieve efficient performance.

For example, consider an application in which in addition to insert and extractMin, we want to be able to *merge* the contents of two different queues into one queue. As an application, suppose that a set of jobs in a computer system are separate queues waiting for the use of two resources. If one of the resources fails, we need to merge these two queues into a single queue.

We introduce a new operation `Q = merge(Q1, Q2)`, which takes two existing priority queues Q_1 and Q_2, and merges them into a new priority queue, Q. (Duplicate keys are allowed.) This operation is *destructive*, which means that the priority queues Q_1 and Q_2 are destroyed in order to form Q. (Destructiveness is quite common for operations that map two data structures into a single combined data structure, since we can simply reuse the same nodes without having to create duplicate copies.)

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 Q_1 and Q_2. Unfortunately, it does not seem to be possible to do this with the standard binary heap data structure (because of its highly rigid structure and the fact that it is stored in an array, without the use of pointers).

We introduce a new data structure called a *leftist heap*, which is fairly simple, and can provide the operations `insert(x)`, `extractMin()`, and `merge(Q1,Q2)`. This data structure has many of similar features to binary heaps. It is a binary tree which is *partially ordered*, which means that the key value in each parent node is less than or equal to the key values in its children's nodes. However, unlike a binary heap, we will not require that the tree is complete, or even balanced. In fact, it is entirely possible that the tree may be quite unbalanced.

Leftist Heap Property: Define the *null path length*, $\mathrm{npl}(v)$, of any node v to be the length of the shortest path to a descendent has either 0 or 1 child. The value of $\mathrm{npl}(v)$ can be defined recursively as follows.

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

We will assume that each node has an extra field, `v.npl` that contains the node's npl value. The *leftist property* states that for every node v in the tree, the npl of its left child is at least as large as the npl of its right child. We say that the keys of a tree are *partially ordered* if each node's key is greater than or equal to its parent's key.

Leftist heap: Is a binary tree whose keys are partially ordered (parent is less than or equal to child) and which satisfies the leftist property ($\mathrm{npl}(\mathit{left}) \geq \mathrm{npl}(\mathit{right})$). An example is shown in the figure below, where the npl values are shown next to each node.

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 property. Observe that this

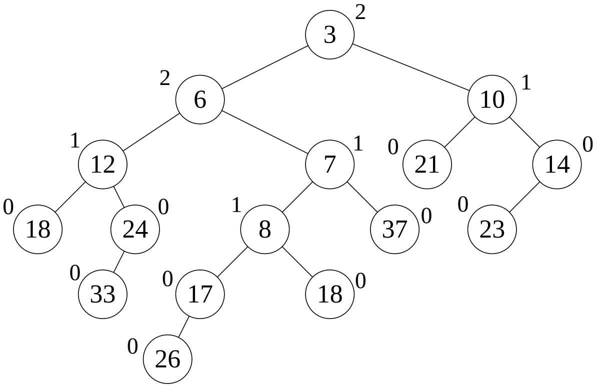


Fig. 97: Leftist heap structure (with npl values shown).

does not affect the partial ordering property. Also observe that satisfying the leftist heap property does not imply that the tree is balanced. Indeed, a degenerate binary tree in which is formed from a chain of nodes each attached as the left child of its parent does satisfy this 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). We prove the following lemma, which implies that the rightmost path in the tree cannot be of length greater than $O(\log n)$.

Lemma: A leftist heap with $r \geq 1$ nodes on its rightmost path has has at least $2^r - 1$ nodes.

Proof: The proof is by induction on the size of the rightmost path. Before beginning the proof, we begin with two observations, which are easy to see: (1) the shortest path in any leftist heap is the rightmost path in the heap, and (2) any subtree of a leftist heap is a leftist heap. For the basis case, if there is only one node on the rightmost path, then the tree has at least one node. Since $1 \geq 2^1 - 1$, the basis case is satisfied.

For the induction step, let us suppose that the lemma is true for any leftist heap with strictly fewer than r nodes on its rightmost path, and we will prove it for a binary tree with exactly r nodes on its rightmost path. Remove the root of the tree, resulting in two subtrees. The right subtree has exactly $r - 1$ nodes on its rightmost path (since we have eliminated only the root), and the left subtree must have a at least $r - 1$ nodes on its rightmost path (since otherwise the rightmost path in the original tree would not be the shortest, violating (1)). Thus, by applying the induction hypothesis, it follows that the right and left subtrees have at least $2^{r-1} - 1$ nodes each, and summing them, together with the root node we get a total of at least

$$2(2^{r-1} - 1) + 1 \;=\; 2^r - 1$$

nodes in the entire tree.

Leftist Heap Operations: The basic operation upon which leftist heaps are based is the merge operation. Observe, for example, that both the operations `insert()` and `extractMin()` can be implemented by using the operation `merge()`. (Note the similarity with splay trees, in which all operations were centered around the splay operation.)

The basic node of the leftist heap is a `LeftHeapNode`. Each such node contains an data field of type `Element` (upon which comparisons can be made) a left and right child, and an npl value. The constructor is given each of these values. The leftist heap class consists of a single `root`, which points to the root node of the heap. Later we will describe the main procedure `merge(LeftHeapNode h1, LeftHeapNode h2)`, which merges the two (sub)heaps rooted at h_1 and h_2. For now, assuming that we have this operation we define the main heap operations. Recall that merge is a destructive operation.

Leftist Operations

```
void insert(Element x) {
    root = merge(root, new LeftHeapNode(x, null, null, 0))
}

Element extractMin() {
    if (root == null) return null       // empty heap
    Element minItem = root.data         // minItem is root's element
    root = merge(root.left, root.right)
    return minItem
}

void merge(LeftistHeap Q1, LeftistHeap Q2) {
    root = merge(Q1.root, Q2.root)
}
```

Leftist Heap Merge: All that remains, is to show how `merge(h1, h2)` is implemented. The formal description of the procedure is recursive. However it is somewhat easier to understand in its nonrecursive form. Let h_1 and h_2 be the two leftist heaps to be merged. Consider the rightmost paths of both heaps. The keys along each of these paths form an increasing sequence. We could merge these paths into a single sorted path (as in merging two lists of sorted keys in the merge-sort algorithm). However the resulting tree might not satisfy the leftist property. Thus we update the npl values, and swap left and right children at each node along this path where the leftist property is violated. A recursive implementation of the algorithm is given below. It is essentially the same as the one given in Weiss, with some minor coding changes.

For the analysis, observe that because the recursive algorithm spends $O(1)$ time for each node on the rightmost path of either h_1 or h_2, the total running time is $O(\log n)$, where n is the total number of nodes in both heaps.

This recursive procedure is a bit hard to understand. A somewhat easier *2-step interpretation* is as follows. First, merge the two right paths, and update the npl values as you do so. Second, walk back up along this path and swap children whenever the leftist condition is violated. The figure below illustrates this way of thinking about the algorithm. The recursive algorithm just combines these two steps, rather than making the separate.

Skew Heaps: Recall that a splay tree is a self-adjusting balanced binary tree. Unlike the AVL tree, which maintains balance information at each node and performs rotations to maintain balance, the splay tree performs a series of rotations to continuously "mix-up" the tree's structure, and hence achieve a sort of balance from an amortized perspective.

A *skew heap* is a self-organizing heap, and applies this same idea as in splay trees, but to the leftist heaps instead. As with splay trees we store no balance information with each node (no npl values). We perform the merge operation as described above, but we swap the left and

Merge two leftist heaps

```
LeftHeapNode merge(LeftHeapNode h1, LeftHeapNode h2) {
    if (h1 == null) return h2               // if one is empty, return the other
    if (h2 == null) return h1
    if (h1.data > h2.data)                  // swap so h1 has the smaller root
        swap(h1, h2)

    if (h1.left == null)                    // h1 must be a leaf in this case
        h1.left = h2
    else {                                  // merge h2 on right and swap if needed
        h1.right = merge(h1.right, h2)
        if (h1.left.npl < h1.right.npl) {
            swap(h1.left, h1.right)         // swap children to make leftist
        }
        h1.npl = h1.right.npl + 1           // update npl value
    }
    return h1
}
```

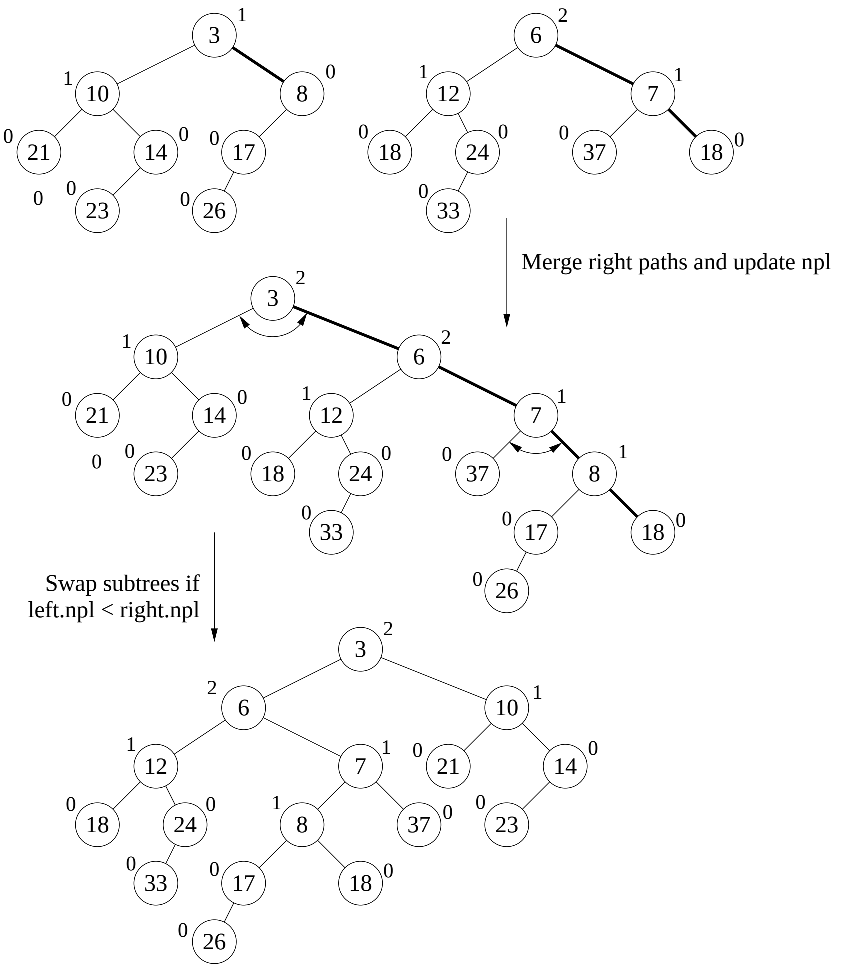


Fig. 98: Example of merging two leftist heaps (2-step interpretation.

right children from *every* node along the rightmost path. (Note that in the process the old rightmost path gets flipped over to become the leftmost path. An example is given in Weiss's book, Section 6.7.)

It can be shown that if the heap contains at most n elements, then a sequence of m heap operations (insert, extractMin, merge) starting from empty heaps takes $O(m \log n)$ time. Thus the average time per operation is $O(\log n)$. Because no balance information is needed, the code is quite simple.

Supplemental Lecture 3: 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). The 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.

In this implementation a *set identifier* is simply a pointer to the root of an inverted tree. Thus the type `Set` is just an alias for the type `Element` (which is perhaps not very good practice, but is very convenient). We will assume that each element x of the set has a pointer `x.parent`, which points to its parent in this tree. To perform the operation `find(x)`, we walk along parent links and return the root of the tree. 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 `find1()`. Later we will propose an improvement.

The Find Operation (First version)

```
Set find1(Element x) {
    while (x.parent != null) x = x.parent;  // follow chain to root
    return x;                               // return the root
}
```

For example, suppose that $S = \{a, b, c, \ldots, m\}$ and the current partition is:

$$\{a, f, g, h, k, l\}, \{b\}, \{c, d, e, m\}, \{i, j\}.$$

This might be stored in an inverted tree as shown in the following figure. The operation `find(k)` would return a pointer to node `g`.

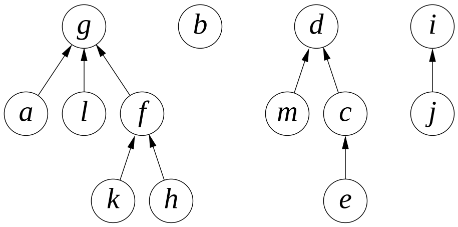


Fig. 99: Union-find Tree Example.

Note that there is no particular order to how the individual trees are structured, as long as they contain the proper elements. (Again recall that unlike existing data structures that we have discussed in which there are key values, here the arguments are pointers to the nodes of the inverted tree. Thus, there is never a question of "what" is in the data structure, the issue is "which" tree are you in.) Initially each element is in its own set. To do this we just set all parent pointers to null.

A union is straightforward to implement. To perform the union of two sets, e.g. to take the union of the set containing $\{b\}$ with the set $\{i, j\}$ we just link the root of one tree into the root of the other tree. But here is where it pays to be smart. If we link the root of $\{i, j\}$ into $\{b\}$, the height of the resulting tree is 2, whereas if we do it the other way the height of the tree is only 1. (Recall that *height* of a tree is the maximum number of edges from any leaf to the root.) It is best to keep the tree's height small, because in doing so we make the find's run faster.

In order to perform union's intelligently, we maintain an extra piece of information, which records the height of each tree. For historic reasons we call this height the *rank* of the tree. We assume that the rank is stored as a field in each element. The intelligent rule for doing unions is to link the root of the set of smaller rank as a child of the root of the set of larger rank.

Observe will only need the rank information for each tree root, and each tree root has no need for a parent pointer. So in a really tricky implementation, the ranks and parent pointers can share the same storage field, provided that you have some way of distinguishing them. For example, in our text, parent pointers (actually indices) are stored with positive integers and ranks are stored as negative integers. We will not worry about being so clever.

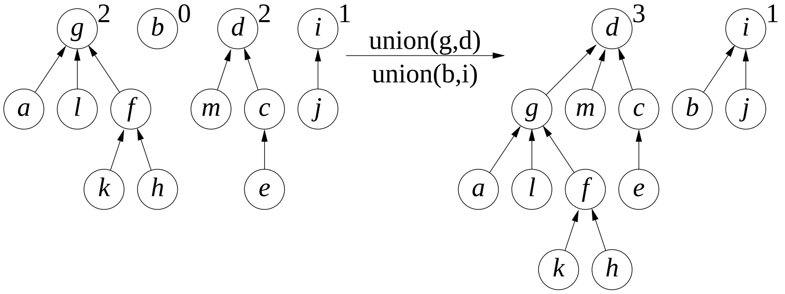


Fig. 100: Union-find with ranks.

The code for union operation is shown in the following figure. When updating the ranks there are two cases. If the trees have the same ranks, then the rank of the result is one larger than this value. If not, then the rank of the result is the same as the rank of the higher ranked tree. (You should convince yourself of this.)

Union operation

```
Set union(Set s, Set t) {
    if (s.rank > t.rank) {      // s has strictly higher rank
        t.parent = s            // make s the root (rank does not change)
        return s
    }
    else {                      // t has equal or higher rank
        if (s.rank == t.rank) t.rank++
        s.parent = t            // make t the root
        return t
    }
}
```

Analysis of Running Time: Consider a sequence of m union-find operations, performed on a 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, and let u denote the number of unions needed to build T. We prove the lemma by strong induction on the number of unions performed to build the tree. For the basis (no unions) we have a tree with 1 element of height 0. Since $2^0 = 1$, the basis case is established.

For the induction step, assume that the hypothesis is true for all trees built with strictly fewer than u union operations, and we want to prove the lemma for a union-find tree built with exactly u union operations. Let T be such a tree. Let T' and T'' be the two subtrees that were joined as part of the final union. Let h' and h'' be the heights of T' and T'', respectively, and define n' and n'' similarly. Each of T' and T'' were built with strictly fewer than u union operations. By the induction hypothesis we have

$$n' \;\geq\; 2^{h'} \qquad \text{and} \qquad n'' \geq 2^{h''}.$$

Let us assume that T' was made the child of T'' (the other case is symmetrical). This implies that $h' \leq h''$, and $h = \max(h'+1, h'')$. There are two cases. First, if $h' = h''$ then $h = h'+1$ and we have

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

Second, if $h' < h''$ then $h = h''$ and we have

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

In either case we get the desired result.

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

Theorem: After initialization, any sequence of m union's and find's can be performed in time $O(m \log m)$.

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.

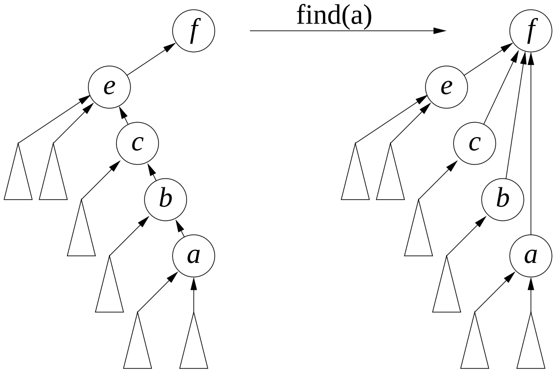


Fig. 101: Find using path compression.

Find Operation using Path Compression

```
Set find2(Element x) {
    if (x.parent == null) return x          // return root
    else return x.parent = find2(x.parent)  // find root and update parent
}
```

The running time of `find2` 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. It was actually believed at one time that if path compression is used, then (after initialization) the running time of the algorithm for a sequence of m union and finds was $O(m)$, and hence the amortized cost of each operation is $O(1)$. However, this turned out to be false, but the amortized time is much less than $O(\log m)$.

Analyzing this algorithm is quite tricky. (Our text gives the full analysis if you are interested.) 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, $A(m, n)$ and $\alpha(n)$. The function $A(m, n)$ is called *Ackerman's function*. It is famous for being just about the fastest growing function imaginable.

$$\begin{aligned} A(1, j) &= 2^j \qquad \text{for } j \geq 1, \\ A(i, 1) &= A(i-1, 2) \qquad \text{for } i \geq 2, \\ A(i, j) &= A(i-1, A(i, j-1)) \qquad \text{for } i, j \geq 2. \end{aligned}$$

In spite of its innocuous appearance, this function is a real monster. To get a feeling for how fast this function grows, observe that

$$A(2, j) = 2^{2^{\cdot^{\cdot^{2}}}},$$

where the tower of powers of 2 is j high. (Try expanding the recurrence to prove this.) Hence,

$$A(4,1) \;=\; A(3,2) \;=\; A(2,A(3,1)) \;=\; A(2,A(2,2)) \;=\; A(2,2^{2^2}) \;=\; A(2,16) \;\approx\; 10^{80},$$

which is already greater than the estimated number of atoms in the observable universe.

Since Ackerman's function grows so fast, its inverse, called α grows incredibly slowly. 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 atoms in the universe (which is certainly true for most input sets!) The following result (which we present without proof) shows that an sequence of union-find operations take amortized time $O(\alpha(m,n))$, and so each operation (for all practical purposes) takes amortized constant time.

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 $O(m\alpha(m,n))$ time. Thus the amortized cost of each union-find operation is $O(\alpha(m,n))$.

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 $\vec{v}$.

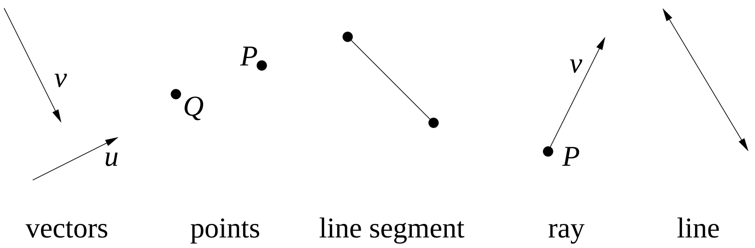


Fig. 102: Basic geometric objects.

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

$$a_0 p_0 + a_1 p_1 + \ldots + a_{d-1} p_{d-1} \ = \ c$$

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 "≥".

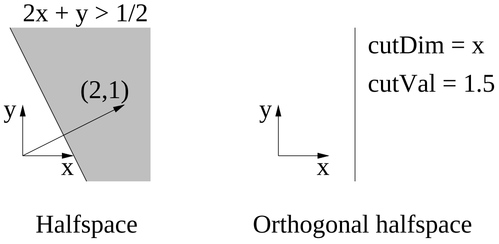


Fig. 103: Planes and Halfspaces.

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 π.

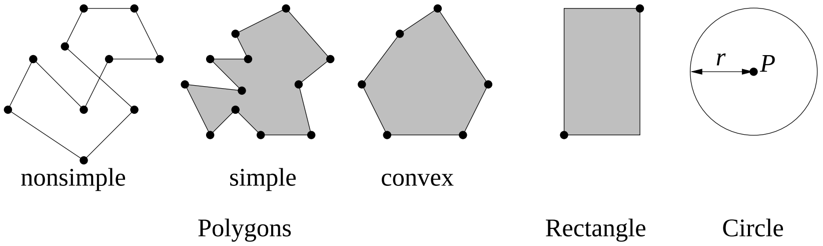


Fig. 104: Polygons, rectangles and spheres.

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.

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 points 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.

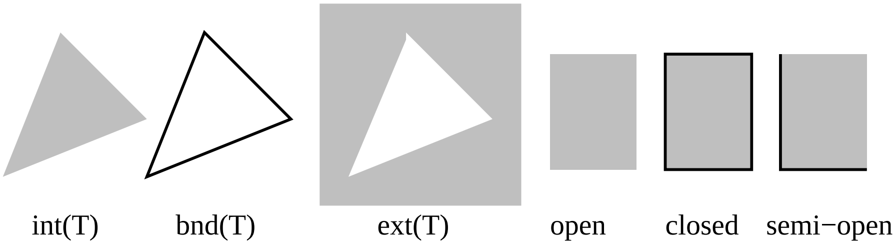


Fig. 105: Topological notation.

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 α 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.

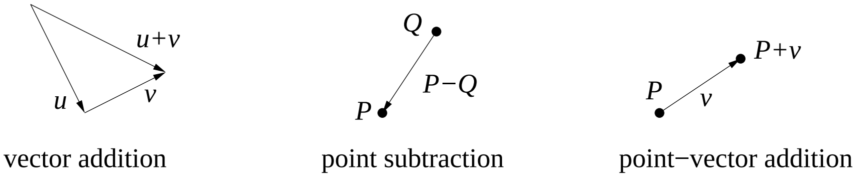


Fig. 106: Point-vector operators.

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 α approaches 0 the point is closer to P and as α approaches 1 the point is closer to Q. This is called

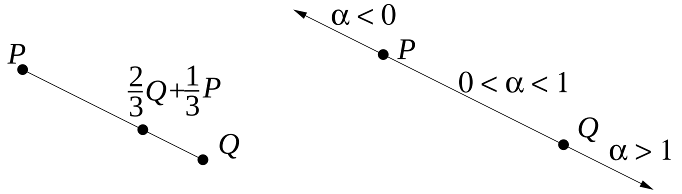


Fig. 107: Affine combinations.

an *affine combination* of P and Q. If $0 \leq \alpha \leq 1$, then the point lies on the line segment $\overline{PQ}$.

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.

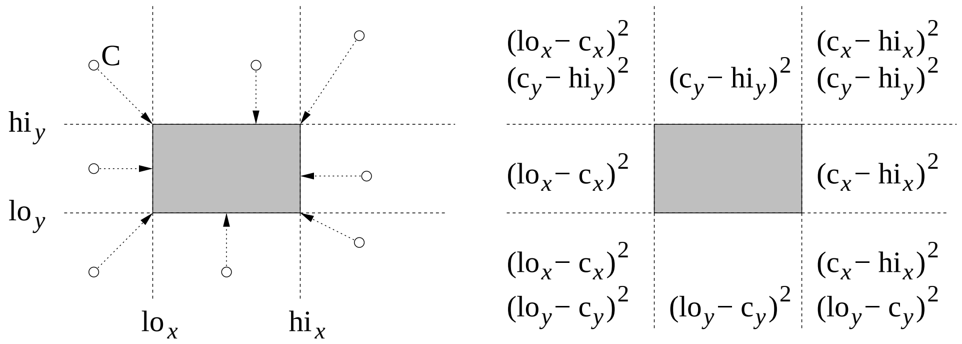

C
hi$_y$
lo$_y$
lo$_x$
hi$_x$
$(\mathrm{lo}_x - c_x)^2$
$(c_y - \mathrm{hi}_y)^2$
$(c_y - \mathrm{hi}_y)^2$
$(c_x - \mathrm{hi}_x)^2$
$(c_y - \mathrm{hi}_y)^2$
$(\mathrm{lo}_x - c_x)^2$
$(c_x - \mathrm{hi}_x)^2$
$(\mathrm{lo}_x - c_x)^2$
$(\mathrm{lo}_y - c_y)^2$
$(\mathrm{lo}_y - c_y)^2$
$(c_x - \mathrm{hi}_x)^2$
$(\mathrm{lo}_y - c_y)^2$

Fig. 108: 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.

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)
}
```

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: 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 is *orthogonal range searching*, where the shape is an axis-aligned rectangle. In this lecture we present a data structure for *orthogonal range-reporting queries*, in which the objective is to report the subset of points of some data set that lie within a query rectangle. We will consider the problem in 2-dimensional space, but generalizations to higher dimensions are straightforward. In an earlier lecture, we discussed the use of kd-trees for answering orthogonal range queries. 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, and if there are k points in the range, then we can report them in total time $O(k+\sqrt{n})$. In this lecture we will present a faster solution. We will present a data structure called a *range tree* which can answer orthogonal counting range queries in $O(\log^2 n)$. (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.) The data structure can be generalized to higher dimensions. In dimension d it answers range queries in $O(\log^d n)$ time.

Range Trees (Basics): The range tree data structure works by reducing an orthogonal range query in 2-dimensions to a collection of $O(\log n)$ range queries in 1-dimension, then it solves each of these in $O(\log n)$ time, for a combined time of $O(\log^2 n)$. (In higher dimensions, it reduces a range query in dimension d to $O(\log n)$ range queries in dimension $d-1$.) It works by a technique called a *multi-level tree*, which involves cascading multiple data structures together in a clever way. Throughout we assume that a range is given by a pair of points $[lo, hi]$, and we wish to report all points p such that

$$lo_x \ \le \ p_x \le hi_x \qquad \text{and} \qquad lo_y \ \le \ p_y \le hi_y.$$

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 of points S, we want to preprocess these points so that given a 1-dimensional interval $Q=[lo, hi]$ along the x-axis, we can count all the points that lie in this interval. There are a number of simple solutions to this, but we will consider a particular method that generalizes to higher dimensions.

Let us begin by storing all the points of our data set in the external nodes (leaves) of a balanced binary tree sorted by x-coordinates (e.g., an AVL tree). The data values in the internal nodes will just be used for searching purposes. They may or may not correspond to actual data values stored in the leaves. We assume that if an internal node contains a value x_0 then the leaves in the left subtree are strictly less than x_0, and the leaves in the right subtree are greater than or equal to x_0. Each node t in this tree is implicitly associated with a subset $S(t) \subseteq S$ of elements of S that are in the leaves descended from t. (For example $S(\text{root}) = S$.) We assume that for each node t, we store the number of leaves that are descended from t, denoted `t.size`. Thus `t.size` is equal to the number of elements in $S(t)$.

Let us introduce a few definitions before continuing. Given the interval $Q=[lo, hi]$, we say that a node t is *relevant* to the query if $S(t) \subseteq Q$. That is, all the descendents of t lie within the interval. If t is relevant then clearly all of the nodes descended from t are also relevant. A relevant node t is *canonical* if t is relevant, but its parent it not. The canonical nodes are the roots of the maximal subtrees that are contained within Q. For each canonical node t, the subset $S(t)$ 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.

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 $[lo, hi]$, we search the tree to find the leftmost leaf u whose key is greater than or equal to lo and the rightmost leaf v whose key is less than or equal to *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. This is illustrated in the following figure.

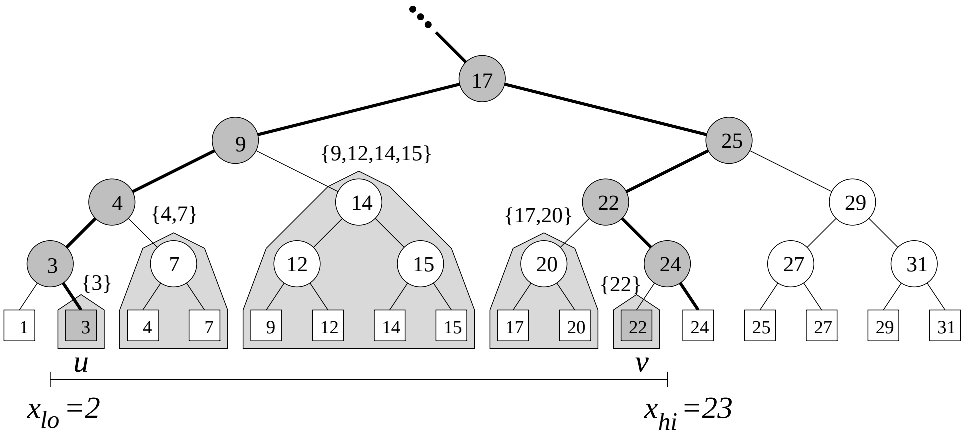


Fig. 109: Canonical sets for interval queries.

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 $[C_{lo}, C_{hi}]$. As with kd-trees, the cell for node t 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. The initial call is `range1D(root, Q, C0)`. Let $t.x$ denote the key associated with t. If $C = [x_0, x_1]$ denotes the current interval for node t, then when we recurse on the left subtree we trim this to the interval $[x_0, t.x]$ and when we recurse on the right subtree we trim the interval to $[t.x, x_1]$. 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 point x is contained within A.

Since the data are only stored in the leaf nodes, when we encounter such a node we consider whether it lies in the range and count it if so. Otherwise, observe that if $t.x \leq Q.lo$ then all the points in the left subtree are less than the interval, and hence it does not need to be visited. Similarly if $t.x > Q.hi$ then the right subtree does not need to be visited. Otherwise, we need to visit these subtrees recursively.

The external nodes counted in the second line and the internal nodes for which we return `t.size` are the canonical nodes. The above procedure answers range counting queries. To extend this to range reporting queries, 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 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 t, such that the answer to the query is the disjoint union of the associated canonical subsets $S(t)$.

1-Dimensional Range Counting Query

```
int range1Dx(Node t, Range Q, Interval C=[x0,x1]) {
    if (t.isExternal)                           // hit the leaf level?
        return (Q.contains(t.point) ? 1 : 0)    // count if point in range
    if (Q.contains(C))                          // Q contains entire cell?
        return t.size                           // return entire subtree size
    int count = 0
    if (t.x > Q.lo)                             // overlap left subtree?
        count += range1Dx(t.left, Q, [x0, t.x]) // count left subtree
    if (t.x <= Q.hi)                            // overlap right subtree?
        count += range1Dx(t.right, Q, [t.x, x1])// count right subtree
    return count
}
```

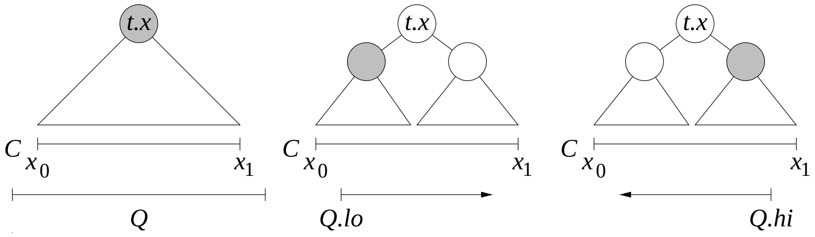


Fig. 110: Range search cases.

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. For each internal node t of T, recall that $S(t)$ denotes the points associated with the leaves descended from t. For each node t of this tree we build a 1-dimensional range tree for the points of $S(t)$, but sorted on y-coordinates. This called the *auxiliary tree* associated with t. Thus, there are $O(n)$ auxiliary trees, one for each internal node of T. An example of such a structure is shown in the following figure.

Notice that there is duplication here, because a given point in a leaf occurs in the point sets associated with each of its ancestors. We claim that the total sizes of all the auxiliary trees is $O(n \log n)$. To see why, observe that each point in a leaf of T has $O(\log n)$ ancestors, and so each point appears in $O(\log n)$ auxiliary trees. The total number of nodes in each auxiliary tree is proportional to the number of leaves in this tree. (Recall that the number of internal nodes in an extended tree one less than the number of leaves.) Since each of the n points appears as a leaf in at most $O(\log n)$ auxiliary trees, the sum of the number of leaves in all the auxiliary trees is at most $O(n \log n)$. Since T has $O(n)$ nodes, the overall total 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. For each such node t, 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 we search the 1-dimensional auxiliary range and return a count of the resulting points. The algorithm below is almost identical the previous one, except that we make explicit reference to the x-coordinates in the search, and rather than adding `t.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-part of Q's range, consisting of the interval `[Q.lo.x,Q.hi.x]`. The call `Q.contains(t.point)` is applied on both coordinates, but the call `Q.x.contains(C)` only checks the x-part of Q's range. The algorithm is given below. The procedure `range1Dy()` is the same procedure described above, except that it searches on y rather than x.

2-Dimensional Range Counting Query

```
int range2D(Node t, Range2D Q, Range1D C=[x0,x1]) {
    if (t.isExternal)                           // hit the leaf level?
        return (Q.contains(t.point) ? 1 : 0)    // count if point in range
    if (Q.x.contains(C)) {                      // Q's x-range contains C
        [y0,y1] = [-infinity, +infinity]        // initial y-cell
        return range1Dy(t.aux.root, Q, [y0, y1])// search auxiliary tree
    }
    int count = 0
    if (t.x > Q.lo.x)                           // overlap left subtree?
        count += range2D(t.left, Q, [x0, t.x])  // count left subtree
    if (t.x <= Q.hi.x)                          // overlap right subtree?
        count += range2D(t.right, Q, [t.x, x1]) // count right subtree
    return count
}
```

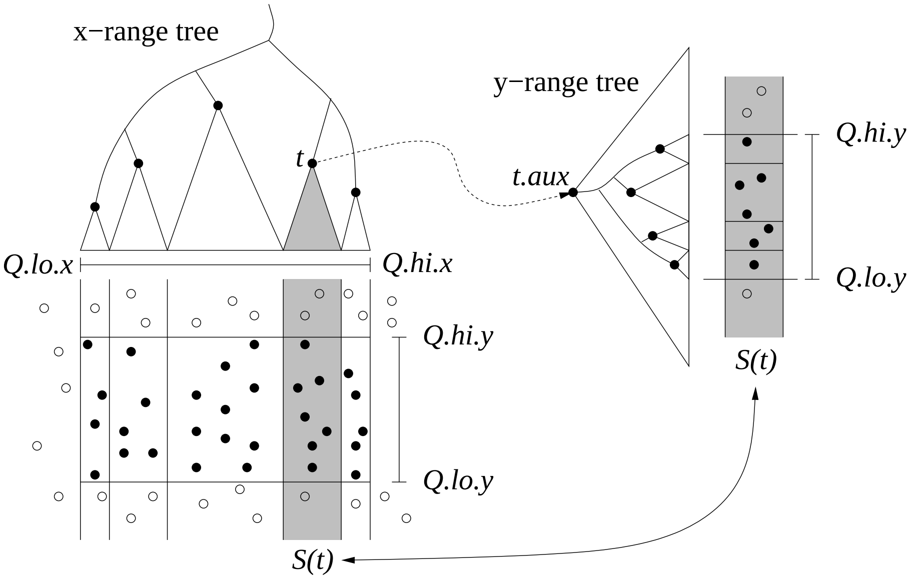


Fig. 111: Range tree.

Analysis: It takes $O(\log n)$ time to identify the canonical nodes along the x-coordinates. For each of these $O(\log n)$ nodes we make a call to a 1-dimensional range tree which contains no more than n points. As we argued above, this takes $O(\log n)$ time for each. Thus the total running 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.

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 $O(n \log^d n)$ space and can answer queries in $O(\log^d n)$ time.

Theorem: d-dimensional range counting queries can be answered in $O(\log^d n)$ time, and range reporting queries can be answered in $O(k + \log^d n)$ time, where k is the number of values reported.

Supplemental Lecture 6: 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 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.

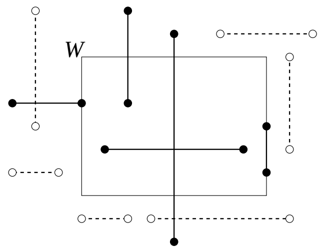


Fig. 112: Window Query.

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 ℓ_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.

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_{med} be the median of these $2n$ endpoints. Split the intervals into three groups, L, those that lie strictly to the left of x_{med}, R those that lie strictly to the right of x_{med}, and M those that contain the point x_{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.

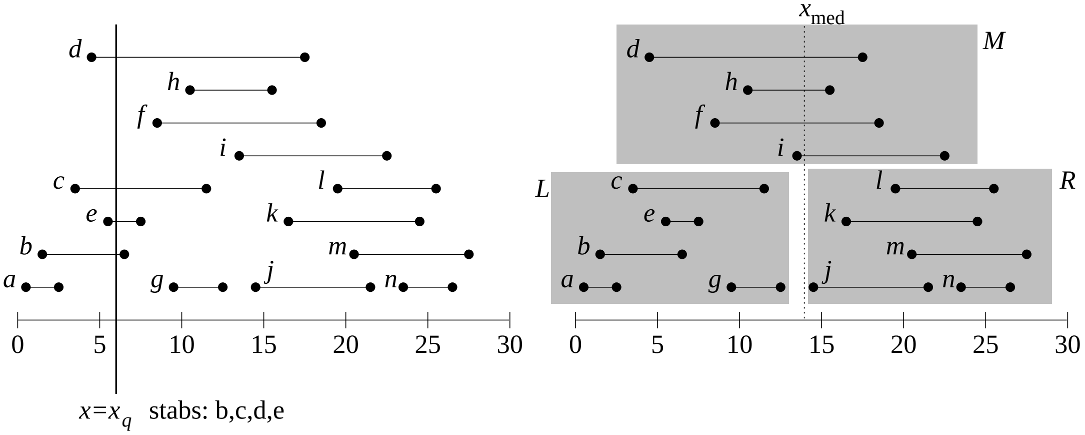


Fig. 113: Line Stabbing Query.

But how do we handle the intervals of M that contain x_{med}? We want to know which of these intervals intersects the vertical line ℓ_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 $x_q \le x_{\text{med}}$. How can we store the intervals of M to make it easier to report those that intersect ℓ_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 ℓ_q, then its left endpoint must be to the right of x_q, and hence none of the subsequent intervals intersects ℓ_q. Thus, to report all the segments of M_L that intersect ℓ_q, we simply traverse the sorted list and list elements until we find one that does not intersect ℓ_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 ℓ_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_{med}, and the two sorted sets M_L and M_R (represented either as arrays or as linked lists) of intervals that overlap x_{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 $2n$

Interval tree construction

```
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
}
```

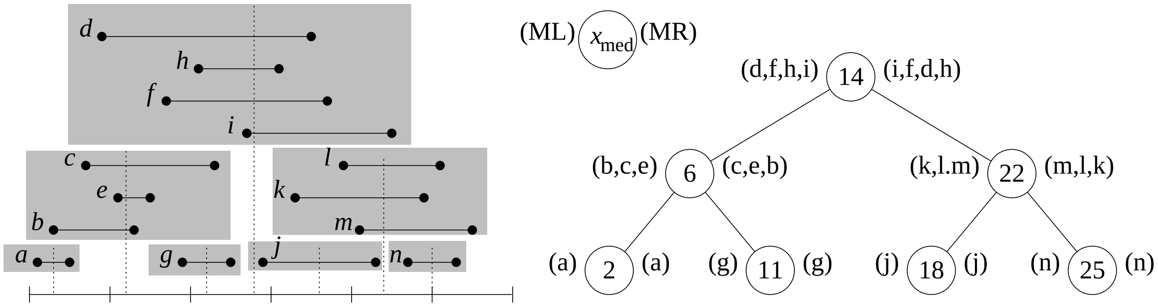


Fig. 114: Interval Tree.

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. 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 $x_q = t.x_{\mathrm{med}}$? In this case we will recursively search the right subtree. However this subtree contains only intervals that are strictly to the right of x_{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 $x = x_{\mathrm{med}}$) 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_{lo}) and (x_q, y_{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_{med} is symmetrical.

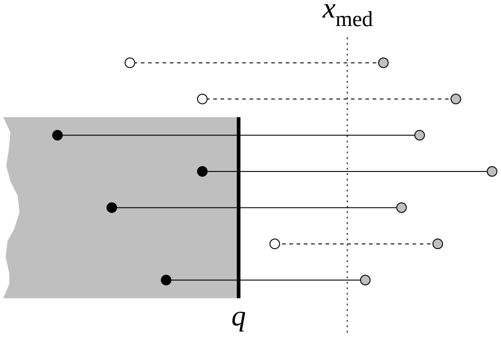


Fig. 115: The segments that stab q lie within the shaded semi-infinite rectangle.

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 7: 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.

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.

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 algo-

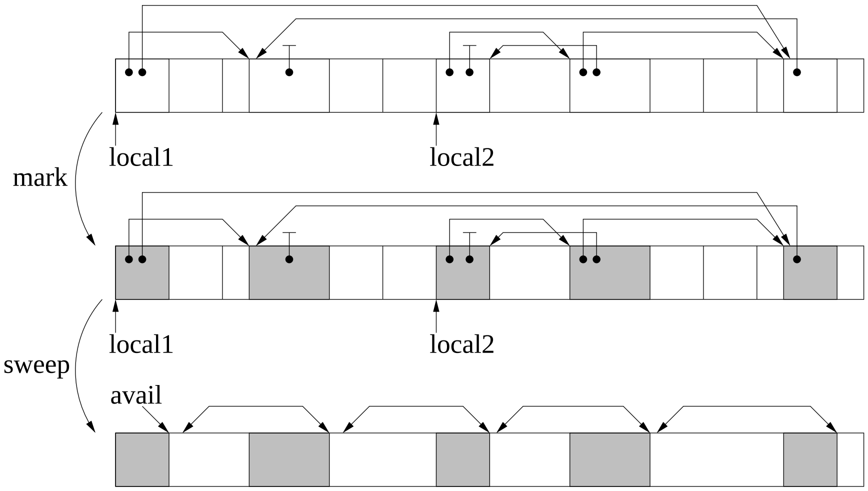


Fig. 116: Mark-and-sweep garbage collection.

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
}
```

rithm. 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 ith 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.

$$\text{descend}(p, t, t.c) : \begin{pmatrix} p \\ t \\ t.c \end{pmatrix} \longleftarrow \begin{pmatrix} t \\ t.c \\ p \end{pmatrix} \qquad \text{ascend}(p, t, p.c) : \begin{pmatrix} p \\ t \\ p.c \end{pmatrix} \longleftarrow \begin{pmatrix} p.c \\ p \\ t \end{pmatrix}.$$

Marking using Link Redirection

```
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
    }
}
```

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

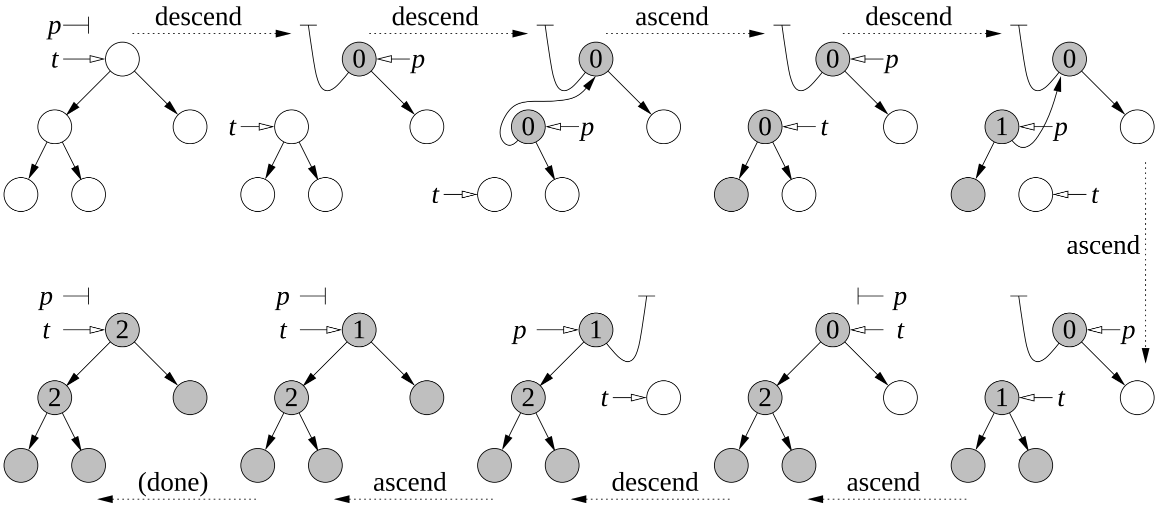


Fig. 117: Marking using link redirection.

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.

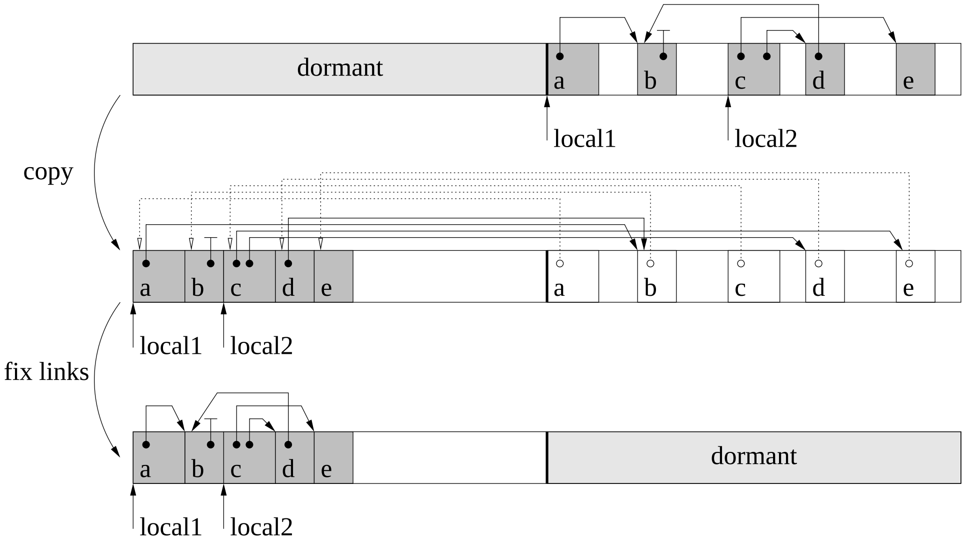


Fig. 118: Stop-and-copy and pointer redirection.

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 8: Tries and Digital Search Trees

Strings and Digital Data: Earlier this semester we studied data structures for storing and retrieving data from an ordered domain through the use of binary search trees, and related data structures such as skip lists. Since these data structures can store any type of sorted data, they can certainly be used for storing strings. However, this is not always the most efficient way to access and retrieve string data. One reason for this is that unlike floating point or integer values, which can be compared by performing a single machine-level operation (basically subtracting two numbers and comparing the sign bit of the result) strings are compared lexicographically, that is, character by character. Strings also possess additional structure that simple numeric keys do not have. It would be nice to have a data structure which takes better advantage of the structural properties of strings.

Character strings arise in a number of important applications. These include language dictionaries, computational linguistics, keyword retrieval systems for text databases and web search, and computational biology and genetics (where the strings may be strands of DNA encoded as sequences over the alphabet $\{C, G, T, A\}$).

Tries: As mentioned above, our goal in designing a search structure for strings is to avoid having to look at every character of the query string at every node of the data structure. The basic idea common to all string-based search structures is the notion of visiting the characters of the search string from left to right as we descend the search structure. The most straightforward implementation of this concept is a *trie*. The name is derived from the middle syllable of "retrieval", but is pronounced the same as "try". Each internal node of a trie is k-way rooted tree, which may be implemented as an array whose length is equal to the number of characters k in the alphabet. It is common to assume that the alphabet includes a special character, indicated by '.' in our figures, which represents a special end of string character. (In languages such as C++ and Java this would normally just be the null character.) Thus each path starting at the root is associated with a sequence of characters. We store each string along the associated path. The last pointer of the sequence (associated with the end of string character) points to a leaf node which contains the complete data associated with this string. An example is shown in the figure below.

The obvious disadvantage of this straightforward trie implementation is the amount of space it uses. Many of the entries in each of the arrays is empty. In most languages the distribution of characters is not uniform, and certain character sequences are highly unlikely. There are a number of ways to improve upon this implementation.

For example, rather than allocating nodes using the system storage allocation (e.g., `new`) we store nodes in a 2-dimensional array. The number of columns equals the size of the alphabet,

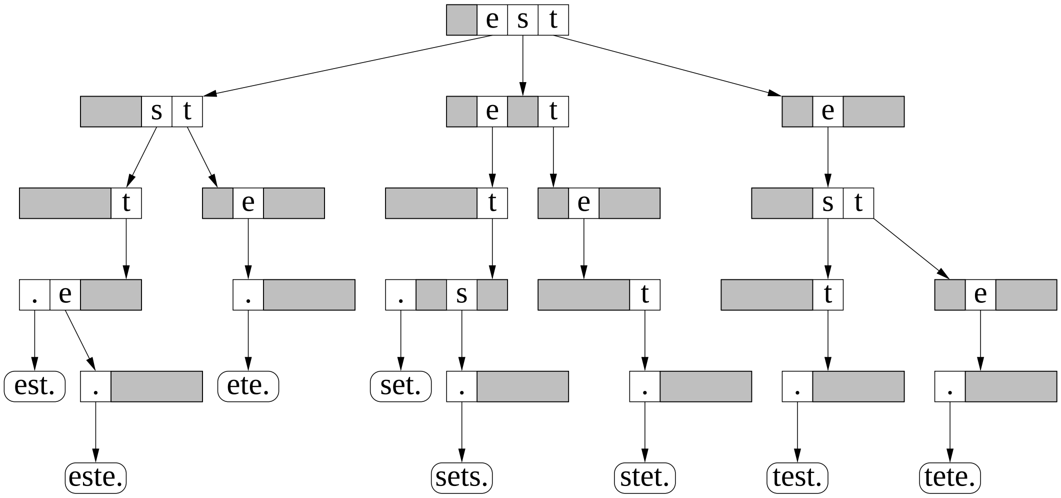


Fig. 119: A trie containing the strings: est, este, ete, set, sets, stet, test and tete. Only the nonnull entries are shown.

and the number of rows equals the total number of nodes. The entry $T[i, j]$ contains the index of the j-th pointer in node i. If there are m nodes in the trie, then $\lceil \lg m \rceil$ bits suffice to represent each index, which is much fewer than the number of bits needed for a pointer.

de la Brandais trees: Another idea for saving space is, rather than storing each node as an array whose size equals the alphabet size, we only store the nonnull entries in a linked list. Each entry of the list contains a character, a child link, and a link to the next entry in the linked list for the node. Note that this is essentially just a first-child, next-sibling representation of the trie structure. These are called *de la Brandais* trees. An example is shown in the figure below.

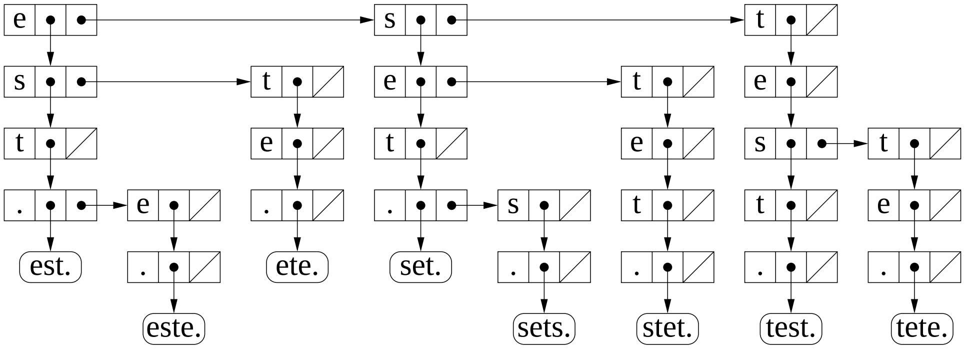


Fig. 120: A de la Brandais tree containing the strings: est, este, ete, set, sets, stet, test and tete.

Although de la Brandais trees have the nice feature that they only use twice as much pointer space as there are characters in the strings, the search time at each level is potentially linear in the number of characters in the alphabet. A hybrid method would be to use regular trie nodes when the branching factor is high and then convert to de la Brandais trees when the branching factor is low.

Patricia Tries: In many applications of strings there can be long sequences of repeated substrings. As an extreme example, suppose that you are creating a trie with the words "demystificational" and "demystifications" but no other words that contain the prefix "demys". In order to stores these words in a trie, we would need to create 10 trie nodes for the common substring "tification", with each node having a branching factor of just one each. To avoid this, we would like the tree to inform us that after reading the common prefix "demys" we should skip over the next 10 characters, and check whether the 11th is either 'a' or 's'. This is the idea behind *patricia tries*. (The word 'patricia' is an acronym for *Practical Algorithm To Retrieve Information Coded In Alphanumeric.*)

A patricia trie uses the same node structure as the standard trie, but in addition each node contains an *index field*, indicating which character is to be checked at this node. This index field value increases as we descend the tree, thus we still process strings from left to right. However, we may skip over any characters that provide no discriminating power between keys. An example is shown below. Note that once only one matching word remains, we can proceed immediately to a leaf node.

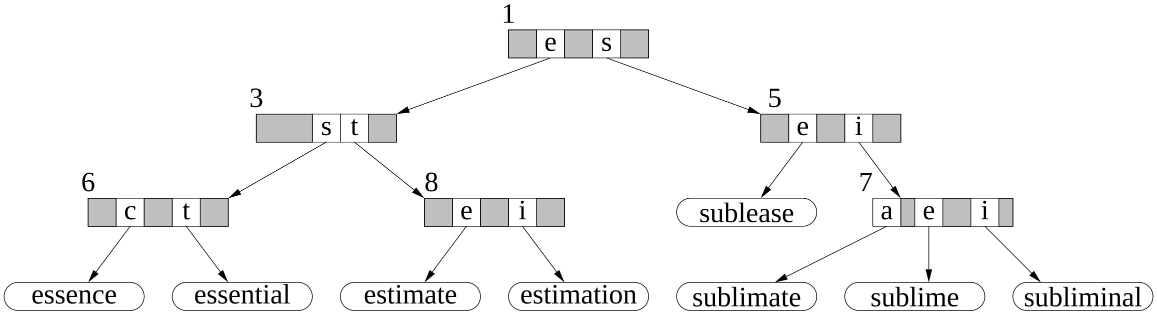


Fig. 121: A patricia trie for the strings: essence, essential, estimate, estimation, sublease, sublime, subliminal.

Observe that because we skip over characters in a patricia trie, it is generally necessary to *verify* the correctness of the final result. For example, if we had attempted to search for the word "survive" then we would match 's' at position 1, 'i' at position 5, and 'e' at position 7, and so we arrive at the leaf node for "sublime". This means that "sublime" is the only possible match, but it does not necessarily match this word. Hence the need for verification.

Suffix trees: In some applications of string pattern matching we want to perform a number of queries about a single long string. For example, this string might be a single DNA strand. We would like to store this string in a data structure so that we are able to perform queries on this string. For example, we might want to know how many occurrences there are of some given substring within this long string.

One interesting application of tries and patricia tries is for this purpose. Consider a string $s =$ "$a_1 a_2 \ldots a_n$.". We assume that the $(n+1)$-st character is the unique string termination character. Such a string implicitly defines $n+1$ suffixes. The i-th suffix is the string "$a_i a_{i+1} \ldots a_n$.". For each position i there is a minimum length substring starting at position i which uniquely identifies this substring. For example, consider the string "yabbadabbadoo". The substring "y" uniquely identifies the first position of the string. However the second position is not uniquely identified by "a" or "ab" or even "abbad", since all of these substrings occur at least twice in the string. However, "abbada" uniquely identifies the second position, because this substring occurs only once in the entire string. The *substring identifier* for position i is the minimum length substring starting at position i of the string which occurs

Position	Substring identifier
1	y
2	abbada
3	bbada
4	bada
5	ada
6	da
7	abbado
8	bbado
9	bado
10	ado
11	do
12	oo
13	o.
14	.

Fig. 122: Substring identifiers for the string "yabbadabbadoo.".

uniquely in s. 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.

A *suffix tree* for a string s is a trie in which we store each of the $n+1$ substring identifiers for s. An example is shown in the following figure. (Following standard suffix tree conventions, we put labels on the edges rather than in the nodes, but this data structure is typically implemented as a trie or a patricia trie.)

As an example of answering a query, suppose that we want to know how many times the substring "abb" occurs within the 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 the example, this is 2.) By storing this information in each node of the tree, we can answer these queries in time proportional to the length of the substring query (irrespective of the length of s).

Since suffix trees are often used for storing very large texts upon which many queries are to be performed (e.g. they were used for storing the entire contents of the Oxford English Dictionary and have been used in computational biology) it is important that the space used by the data structure be $O(n)$ where n is the number of characters in the string. Using the standard trie representation, this is not necessarily true. (You can try to generate your own counterexample where the data structure uses $O(n^2)$ space, even if the alphabet is limited to 2 symbols.) However, if a patricia trie is used the resulting suffix tree has $O(n)$ nodes. The reason is that the number of leaves in the suffix tree is equal to $n+1$. We showed earlier in the semester that if every internal node has two children (or generally at least two children) then the number of internal nodes is not greater than n. Hence the total number of nodes is $O(n)$.

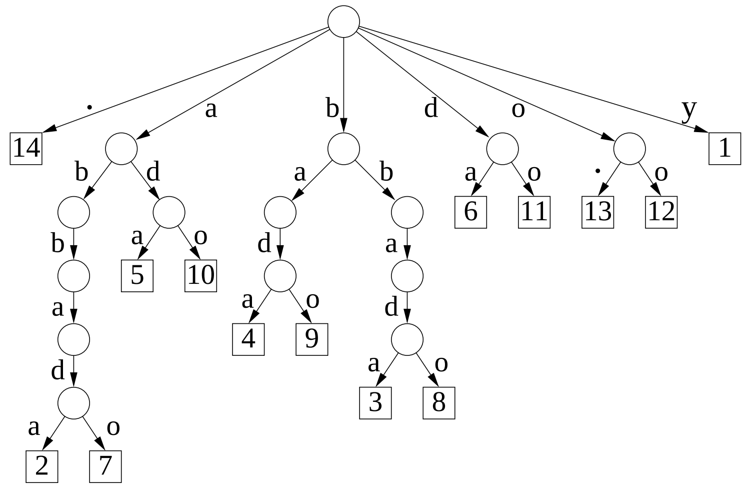


Fig. 123: A suffix tree for the string “yabbadabbadoo.”.