CMSC 420: Spring 2023

CMSC 420: Short Reference Guide

This document contains a short summary of information about algorithm analysis and data structures, which may be useful later in the semester.

Asymptotic Forms: The following gives both the formal "c and n_0" definitions and an equivalent limit definition for the standard asymptotic forms. Assume that f and g are nonnegative functions.

Asymptotic Form	Relationship	Limit Form	Formal Definition
$f(n) \in \Theta(g(n))$	$f(n) \equiv g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	$\exists c_1, c_2, n_0, \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$.
$f(n) \in O(g(n))$	$f(n) \preceq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	$\exists c, n_0, \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)$.
$f(n) \in \Omega(g(n))$	$f(n) \succeq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$	$\exists c, n_0, \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)$.
$f(n) \in o(g(n))$	$f(n) \prec g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$\forall c, \exists n_0, \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)$.
$f(n) \in \omega(g(n))$	$f(n) \succ g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	$\forall c, \exists n_0, \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)$.

Polylog-Polynomial-Exponential: For any constants a, b, and c, where $b > 0$ and $c > 1$.

$$\log^a n \prec n^b \prec c^n.$$

Common Summations: Let c be any constant, $c \neq 1$, and $n \geq 0$.

Name of Series	Formula	Closed-Form Solution	Asymptotic
Constant Series	$\sum_{i=a}^{b} 1$	$= \max(b-a+1, 0)$	$\Theta(b-a)$
Arithmetic Series	$\sum_{i=0}^{n} i = 0+1+2+\cdots+n$	$= \frac{n(n+1)}{2}$	$\Theta(n^2)$
Geometric Series	$\sum_{i=0}^{n} c^i = 1+c+c^2+\cdots+c^n$	$= \frac{c^{n+1}-1}{c-1}$	$\begin{cases} \Theta(c^n)\ (c>1) \\ \Theta(1)\ (c<1) \end{cases}$
Quadratic Series	$\sum_{i=0}^{n} i^2 = 1^2+2^2+\cdots+n^2$	$= \frac{2n^3+3n^2+n}{6}$	$\Theta(n^3)$
Linear-geom. Series	$\sum_{i=0}^{n-1} ic^i = c+2c^2+3c^3\cdots+nc^n$	$= \frac{(n-1)c^{(n+1)}-nc^n+c}{(c-1)^2}$	$\Theta(nc^n)$
Harmonic Series	$\sum_{i=1}^{n} \frac{1}{i} = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$	$\approx \ln n$	$\Theta(\log n)$

Recurrences: Recursive algorithms (especially those based on divide-and-conquer) can often be analyzed using the so-called *Master Theorem*, which states that given constants $a > 0$, $b > 1$, and $d \geq 0$, the function $T(n) = aT(n/b) + O(n^d)$, has the following asymptotic form:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

For example, suppose that we have the following recurrence (which arises in the analysis of quadtrees). Assuming $n \geq 1$ is a power of 4:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T\left(\frac{n}{4}\right) + 1 & \text{otherwise.} \end{cases}$$

We have $a = 2$, $b = 4$, and $d = 0$ (because $1 = O(n^0)$). We have $\log_b a = \log_4 2 = \frac{1}{2}$, and clearly $d = 0 < \frac{1}{2}$, and therefore the third case applies, yielding $T(n) = O(n^{\log_4 2}) = O(n^{1/2}) = O(\sqrt{n})$.

Sorting: The following algorithms sort a set of n keys over a totally ordered domain. Let $[m]$ denote the set $\{0, \ldots, m\}$, and let $[m]^k$ denote the set of ordered k-tuples, where each element is taken from $[m]$. A sorting algorithm is *stable* if it preserves the relative order of equal elements. A sorting algorithm is *in-place* if it uses no additional array storage other than the input array (although $O(\log n)$ additional space is allowed for the recursion stack). The *comparison-based algorithms* (Insertion-, Merge-, Heap-, and QuickSort) operate under the general assumption that there is a *comparator function* $f(x, y)$ that takes two elements x and y and determines whether $x < y$, $x = y$, or $x > y$.

Algorithm	Domain	Time	Space	Stable	In-place
CountingSort	Integers $[m]$	$O(n + m)$	$O(n + m)$	Yes	No
RadixSort	Integers $[m]^k$ or $[m^k]$	$O(k(n + m))$	$O(kn + m)$	Yes	No
InsertionSort	Total order	$O(n^2)$	$O(n)$	Yes	Yes
MergeSort HeapSort QuickSort	Total order	$O(n \log n)$	$O(n)$	Yes No Yes/No*	No Yes No/Yes

*There are two versions of QuickSort, one which is stable but not in-place, and one which is in-place but not stable.

Order statistics: For any k, $1 \leq k \leq n$, the kth smallest element of a set of size n (over a totally ordered domain) can be computed in $O(n)$ time.

Useful Data Structures: All the following data structures use $O(n)$ space to store n objects:

Unordered Dictionary: (by hashing) Insert, delete, and find in $O(1)$ expected time each. (Note that you can find an element exactly, but you cannot quickly find its predecessor or successor.)

Ordered Dictionary: (by balanced binary trees or skiplists) Insert, delete, find, predecessor, successor, merge, split in $O(\log n)$ time each. (Merge means combining the contents of two dictionaries, where the elements of one dictionary are all smaller than the elements of the other. Split means splitting a dictionary into two about a given value x, where one dictionary contains all the items less than or equal to x and the other contains the items greater than x.) Given the location of an item x in the data structure, it is possible to locate a given element y in time $O(\log k)$, where k is the number of elements between x and y (inclusive).

Priority Queues: (by binary heaps) Insert, delete, extract-min, union, decrease/increase-key in $O(\log n)$ time. Find-min in $O(1)$ time each. Make-heap from n keys in $O(n)$ time.

Priority Queues: (by Fibonacci heaps) Supports insert, find-min, decrease-key all in $O(1)$ amortized time. (That is, a sequence of length m takes $O(m)$ total time.) Extract-min and delete take $O(\log n)$ worst-case time, where n is the number of items in the heap.

Disjoint Set Union-Find: (by inverted trees with path compression) Union of two disjoint sets and find the set containing an element in $O(\log n)$ time each. A sequence of m operations can be done in $O(\alpha(m, n))$ amortized time. That is, the entire sequence can be done in $O(m \cdot \alpha(m, n))$ time. (α is the *extremely* slow growing inverse-Ackerman function.)

CMSC 420: Spring 2023

Programming Assignment 0: Adjustable Stack

Overview: This is a start-up project designed to acquaint you with the programming/testing environment and submission process we will be using this semester. This will involve only a small bit of data structure design and implementation.

The Adjustable Stack: In this assignment we will implement a generalization of the simple stack data structure, but along the way we will introduce a useful concept, called a *locator*. Like the standard Java stack data structure, our data structure is generic and stores objects of a given type, called `Element`.

```
public class AdjustableStack<Element> { ... }
```

It implements the standard stack operations of `push`, `pop`, and `peek` (which returns the top entry without removing it). In addition, we allow entries within the stack to be adjusted by either *promotion* or *demotion*. Promoting moves an entry one position closer to the top by swapping it with the entry immediately above it. Demoting moves it down one position by swapping it with the entry just below it. We will also support a *depth* operation, which returns how far an entry is from the top.

In order to implement these last three operations, we need an efficient mechanism for referring to specific entries of the stack. (Java provides a search operation, but this is very slow, since it involves searching the entire stack contents.) We do this using an object called a *locator*. A locator identifies an specific entry in the stack. This is implemented as a public class, called `Locator`, that is internal to the `AdjustableStack` class. Whenever an object is pushed into the stack, the `push` operation returns a locator referring to this entry. The promotion, demotion, and depth operations are each given a locator referencing the object on which to apply the operation.

How do we do this? Let's assume we implement the stack in the standard manner as an array of entries, along with an index `top`, which indexes the top entry of the stack. When an item is pushed on the stack, we create a new locator object, which stores the index of the entry in the stack array. Then, when we wish to promote or demote an entry, we provide the locator in order to identify this entry. An skeletal example is shown below.

```
public class AdjustableStack<Element> {
    public class Locator {
        private int index;
    }
    public Locator push(Element element) { ... }
    public void promote(Locator loc) { ... }
}
```

Here is an example of how this might be used. We create a new stack of strings, push three entries, and then we promote the middle entry.

```
AdjustableStack<String> stack = new AdjustableStack<String>();
AdjustableStack<String>.Locator loc1 = stack.push("cat"); // stack: cat
AdjustableStack<String>.Locator loc2 = stack.push("dog"); // stack: dog cat
AdjustableStack<String>.Locator loc3 = stack.push("pig"); // stack: pig dog cat
stack.promote(loc2); // promote dog. New stack: dog pig cat
```

The contents of the stack after the three pushes is shown in Fig. 1(a). The result after promoting `"dog"` is shown in Fig. 1(b). Observe that when `"dog"` and `"pig"` exchange places their associated locators are updated accordingly.

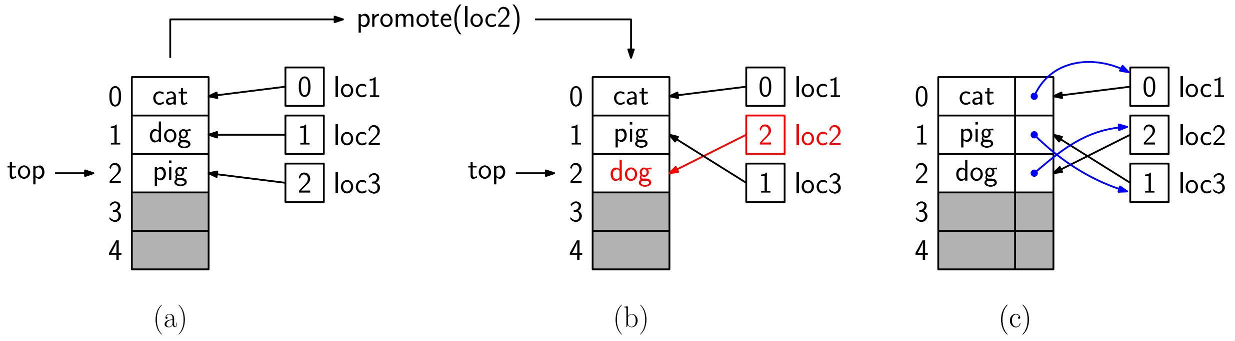


Figure 1: Locators and promotion.

You might wonder, why do we go through the extra effort of creating this special `Locator` class. Couldn't the `push` operator simply return the integer index of the newly inserting entry? The problem is that as other entries are promoted and demoted, and given items position in the stack will change dynamically. (As shown in Fig. 1(b), the locator for `"pig"` had to be changed, even though it was not named in the promotion opertion.) The purpose of the locator is to keep track of where the item is at all times.

But, you may see an obvious problem. If an entry's position is changed because one of its neighbors is promoted, how to we find its associated locator? To make this work efficiently, in addition to the locator referencing an entry in the stack, we need each stack entry to reference the associated locator. This way, when the entry moves, we can update the locator. This is illustrated in Fig. 1(c). You could do this in one of two ways. You could have two parallel arrays, one an array of type `Element`, containing the stack contents, and the other of type `Locator`, containing references to the locators. Alternatively, you could create a new (private) inner class within `AdjustableStack`, which contains two members, the element and the locator. Either approach is fine.

Operations: Here is a list of all the public methods your program is to implement.

`AdjustableStack()`: This creates a new empty stack. (`Hint:` While it is tempting to use Java's built-in `stack` object, this is not a good idea because you will need to implement your own `push` operation. It is inconvenient to use a fixed-size array, since you cannot predict in advance how many entries we will push, which necessitates expanding the array. We would recommend that you store your stack in an expandable array, such as Java's `ArrayList`. This allows you to efficiently access individual elements, and you can add as many entries as needed.)

`Locator push(Element element):` This pushes `element` onto your stack. It also creates a new `Locator` object that references this element and returns this `Locator`.

`Element pop():` This removes the element from the stack and returns its value. If the stack is empty, this throws an `Exception` with the message `"Pop of empty stack"`.

`Element peek():` This returns a reference to the top element of the stack, without altering its contents. If the stack is empty, this throws an `Exception` with the message `"Peek of empty stack"`.

`int size():` Returns the number of elements currently in the stack.

`void promote(Locator loc):` Promotes the stack entry referenced by `loc`. If `loc` references the top of the stack, then this does nothing. Otherwise, it swaps this entry with the entry that is one position closer to the top of the stack. You may assume that that given locator is valid, and it refers to an element that is currently in the stack.

`void demote(Locator loc):` Demotes the stack entry referenced by `loc`. If `loc` references the bottom entry of the stack, then this does nothing. Otherwise, it swaps this entry with the entry that is one position farther from the top of the stack. You may assume that that given locator is valid, and it refers to an element that is currently in the stack.

`int getDepth(Locator loc):` Returns the depth of the stack entry referenced by `loc`. This is defined to be the number of positions from the top of the stack (so, the depth of the top itself is defined to be zero). You may assume that that given locator is valid, and it refers to an element that is currently in the stack.

`ArrayList<String> list():` This returns a Java `ArrayList` whose members are the elements of the stack, listed from the top of the stack down to the bottom. For example, if you started with an empty stack and performed `push("cat"); push("dog"); push("pig")`, this returns an `ArrayList` containing ⟨`"pig", "dog", "cat"`⟩.

What you need to do: We will provide you with two programs that take care of the input and output (`Part0Tester.java` and `Part0CommandHandler.java`). All you need to do is to implement the above functions. In fact, we will give you a skeleton program, `AdjustableStack.java`, with all the function prototypes, and you just need to fill them in.

```
package cmsc420_s23; // Do not alter this line
import java.util.ArrayList;

public class AdjustableStack<Element> {

    public class Locator { /* ... */ }

    public AdjustableStack() { /* ... */ }
    public Locator push(Element element) { /* ... */ return null; }
    public Element pop() throws Exception { /* ... */ return null; }
    public Element peek() throws Exception { /* ... */ return null; }
    public int size() { /* ... */ return 0; }
    public ArrayList<Element> list() { /* ... */ return null; }
    public void promote(Locator loc) { /* ... */ }
    public void demote(Locator loc) { /* ... */ }
```

```
        public int getDepth(Locator loc) { /* ... */ return 0; }
    }
```

Sample input/output: Here is an example of what the input and output might look like.

Input:	**Output:**
`push:cat`	`push(cat): successful`
`push:dog`	`push(dog): successful`
`push:pig`	`push(pig): successful`
`list`	`list: pig dog cat`
`size`	`size: 3`
`promote:dog`	`promote(dog): successful`
`list`	`list: dog pig cat`
`depth:pig`	`depth(pig): 1`
`pop`	`pop: dog`

What we give you: We will provide you with skeleton code to get you started on the class Projects page (`Part0-Skeleton.zip`). This code will handle the input and output and provide you with the Java template for `AdjustableStack`. All you need to do is fill in the contents of this class. Note that directory structure has been set up carefully. You should not alter it unless you know what you are doing.

Files: Our skeleton code provides the following files, which can be found in the folder "`cmsc420_s23`". Note that all must begin with the statement "`package cmsc420_s23`".

`Part0Tester.java`**:** This contains the main Java program. It reads input commands from a file (by default `tests/test01-input.txt`) and it writes the output to a file (by default `tests/test01-output.txt`). You can alter the name of the input and output files.

▷ *You should not modify this except possibly to change the input and/or output file names. The output is sent to a file in the `tests` directory, not to the Java console. Also note that if you use `Eclipse`, the contents of the `File Explorer` window are not automatically updated. You will need to refresh its contents to see the new output file.*

We will provide you with a few sample test input files along with the "expected" output results (e.g., `tests/test01-expected.txt`). Of course, you should do your own testing. To check your results, use a difference-checking program like "diff".

Note that the tester program does not generate output to the console (unless there are errors). The output is stored in the output file in the `tests` directory.

`Part0CommandHandler.java`**:** This provides the interface between our `Part0Tester.java` and your `AdjustableStack.java`. It invokes the functions in your `AdjustableStack` class and outputs the results. It also catches and processes any exceptions.

▷ *You should not modify this file.*

Requirements: Since this is the first assignment, there are no requirements regarding efficiency or good coding style. The grade is based entirely on the Gradescope autograder.

Partial Credit: If you don't have time to implement the full version, you can get 50% partial credit by simply implementing the operations that do not involve locators, namely the constructor and the operations, `push`, `pop`, `peek`, and `size`.

CMSC 420: Spring 2023

Programming Assignment 1: Weight-based Leftist Heaps

Overview: In this programming assignment you will implement a variant of a leftist heap. Recall that a leftist heap is a mergeable form of the heap structure, which was presented in Lecture 5. Rather than using NPL (null-path length) values to balance the tree, our data structure will be "weight-based," in the sense that balance will be based on the number of nodes in each subtree.

A Weight-Based Leftist Heap: Our data structure, called `WtLeftHeap` will store key-value pairs, where the key represents the entry's priority. The data type is generic and is templated by two types `Key` and `Value`. The `Key` type implements the Java `Comparable` interface, meaning that it must provide a function `compareTo()` for comparing keys. It is declared as follows:

```
public class WtLeftHeap<Key extends Comparable<Key>, Value>
```

Our weight-based leftist heap differs in a number of respects from the standard leftist heap presented in class:

- Rather than a min-heap, this will be a *max-heap*, meaning that the each node's parent key is greater than or equal to its key (see Fig. 1).

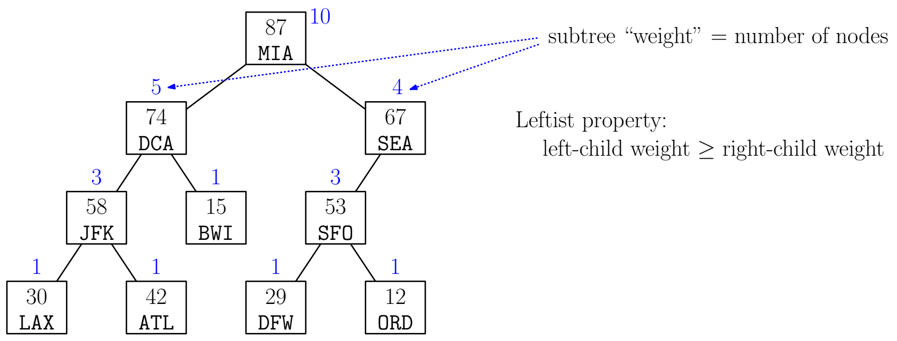


Figure 1: Weight-based leftist heap.

- Rather than basing the tree structure on the NPL (null path length), it will be based on a node's *weight*, which is defined to be the number of nodes in the subtree rooted at this node. Each node will store its weight. The weight of its left subtree can never be smaller than the weight of its right subtree (see Fig. 1).
- In addition to the standard leftist-heap operations, you will also implement a function that *updates* the key (priority) of a given entry. Since this alteration may violate the max-heap structure, the tree structure may need to be updateed as well.
- In order to efficiently support the above update operation, we will need to make two additional enhancements to the tree structure. First, (as with the adjustable stack from

the earlier assignment) *locators* will be used to identify the node to be updateed. Since the subsequent reorganization may involve entries moving both up and down the tree, we will need to add a *parent link* to each node of the tree.

Locators and Adjustments: In many applications of heaps, it is necessary to update the priorities of entries that reside in the heap. This raises two questions: (1) how to efficiently identify an entry within the heap and (2) how to efficiently update the heap's structure after making the modification. As in the earlier programming assignment, we will handle the first issue with the use of *locators*.

Whenever we insert a key-value pair in the tree, we return a reference to the node containing the newly created entry. Since a node will be a protected object within our data structure, we cannot just return a reference directly to it. Instead, we create a special public class within our data structure, called a `Locator`, which encapsulates this reference. In addition to creating a new node and adding it to the data structure, the insert function also creates a new `Locator` object that references this node and returns it to the user of our data structure. Now, when the user wants to update a key, it can use this locator to efficiently identify the node to be updateed. A skeletal example of how to set this up is provided below.

```
public class WtLeftHeap<Key extends Comparable<Key>, Value> {
    private class Node { ... }            // a node of the tree (private)
    public class Locator {                // a node locator (public)
        private Node node;                // hidden reference to the node
        ...
    }
    public Locator insert(Key x, Value v) // insert method returns a Locator
    ...
}
```

It is the user's responsibility to save these locators, and it is the data structure's responsibility to see that they are properly updated. As an example, consider the weight-based leftist heap shown in Fig. 2(a), and suppose that the user wants to update the key of `ORD` from 12 to 82. When `ORD` was added to the heap, the `insert` function returned a locator object. To modify the key, we pass the locator into the `updateKey` function, which allows us to access the node directly. We can then change its key to 82. Unfortunately, the heap order is now violated. Since the key has increased, we can fix this by sifting the entry up the tree, repeatedly swapping with its parent, until its parent's key is at least as larger (see Fig. 2(b)). Depending on how you implement this operation (moving nodes or copying their contents) other locators may need to be updated as well.

Operations: You will implement the following public functions. Subject to the efficiency requirements described below, you are free to create whatever additional private/protected data and utility functions as you like.

`WtLeftHeap()`: This constructs an empty heap. This creates an empty tree by initializing the `root` to `null` (and performs any other initializations as needed by your particular implementation).

`int size()`: Returns the number of entries in the current heap.

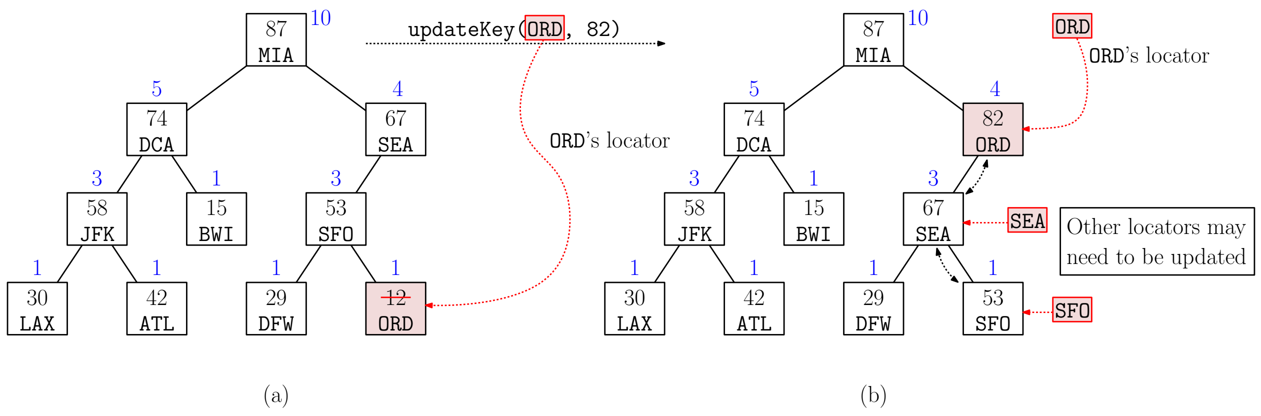


Figure 2: (a) A weight-based leftist heap (with weights indicated in blue) and (b) the `updateKey` operation.

`void clear()`: This resets the structure to its initial state, effectively removing all its existing contents. (You do not need to do anything to existing locators that point into your data structure. It is the responsibility of the user to remove these.)

`Locator insert(Key x, Value v)`: Inserts the key-value pair $(\mathtt{x}, \mathtt{v})$, where `x` is the key and `v` is the value.

Hint: This can be implemented without the need for loops or recursion by making a single call to the `merge` utility function.

`void mergeWith(WtLeftHeap<Key, Value> h2))`: This merges the current heap with the heap `h2`. If `h2` is `null` or it references this same heap (that is, `this == h2`) then this operation has no effect. Otherwise, the two heaps are merged, with the current heap holding the union of both heaps, and `h2` becoming an empty heap.

For testing purposes, you should implement merge operation so it produces exactly the same tree as in the lecture notes. The only difference is that rather than using NPL values to determine which subtree is on the left, use the weight of each subtree, that is, the number of nodes. (As an exercise, convince yourself that the rightmost path of such a tree has $O(\log n)$ nodes.)

`Value extract()`: If the heap is empty, this throws an `Exception` with the error message `"Extract from empty heap"` Otherwise, this locates the entry with the maximum key value, deletes it from the heap, and returns its associated value.

Hint: This can be implemented without the need for loops or recursion by making a single call to the `merge` utility function.

`void updateKey(Locator loc, Key x)`: Change the key associated with entry `loc` to `x`, and update the structure appropriately. You may assume that `loc` is a valid locator for this instance of the data structure.

If the key increases, repeatedly swap this entry with its parent until reaching the root or until the parent key is greater than or equal. If the key decreases, repeatedly swap with the larger of the two children until reaching the leaf level or until both children's keys are less than or equal.

Hint: There are two obvious methods on how to do this. One involves swapping entire nodes by unlinking and relinking them, and the other involves leaving the nodes where they are, but swapping their contents. You may implement whichever version you prefer (since it won't affect the results).

`Key peekKey()`: Returns the maximum key in the heap (that is, the key associated with the root node). If the tree is empty, return `null`.

`Value peekValue()`: Returns the value associated with the maximum key in the heap (that is, the value associated with the root node). If the tree is empty, return `null`.

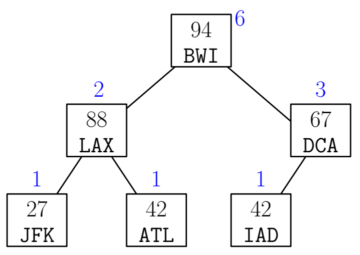
`ArrayList<String> list()`: This operation lists the contents of your tree in the form of a Java `ArrayList` of strings. The precise format is important, since we check for correctness by "diff-ing" your strings against ours.

Starting at the root node, visit all the nodes of this tree based on a **right-to-left preorder traversal**. In particular, when visiting a node reference `u`, we do the following:

Null: ($\texttt{u} = \texttt{null}$) Add the string `"[]"` to the end of the array-list and return.

Non-null: ($\texttt{u} \neq \texttt{null}$) Add the string `"(" + u.key + ",␣" + u.value + ")␣[" + u.weight + "]"` with the node's key, value, and weight to the end of the array-list. (The symbol "␣" is a space.) Then recursively visit `u.right` and then `u.left`.

Our command handler program takes the result of the `list` command and generates a nicely formatted tree (see Fig. 3(c)). The reason for performing the traversal in right-to-left order, rather than the traditional left-to-right, is so that the printed result looks like a 90° rotation of the tree.



(a)

Raw output

```
(94, BWI) [6]
(67, DCA) [2]
[]
(42, IAD) [1]
[]
[]
(88, LAX) [3]
(48, ATL) [1]
[]
[]
(27, JFK) [1]
[]
[]
```

(b)

Formatted output

```
 | (67, DCA) [2]
 | | (42, IAD) [1]
(94, BWI) [6]
 | | (48, ATL) [1]
 | (88, LAX) [3]
 | | (27, JFK) [1]
```

(c)

Figure 3: (a) A heap, (b) the result of `list`, and (c) the formatted output produced by our program.

Skeleton Code: As in the earlier assignment, we will provide skeleton code on the class Projects Page. The only file that you should need to modify is `WtLeftHeap.java`.

Remember that you must use the package "`cmsc420_s23`" in all your source files in order for the autgrader to work. As before, we will provide the programs `Part1Tester.java` and `Part1CommandHandler.java` to process input and output. You need only implement the data structure and the functions listed above.

Class Structure: The high-level `WtLeftHeap` class structure is presented below. There are two inner classes, a private one for the node and a public one for the locator. The entries each consist of a key (priority) and associated value. These can be any two types, but it must be possible to make comparisons between keys. Our class is parameterized with two types, `Key` and `Value`. We assume that the `Key` object implements Java's `Comparable` interface, which means that is supports a method `compareTo` for comparing two such objects. This is satisfied for all of the Java's standard number types, such as `Integer`, `Float`, and `Double` as well as for `String`.

We recommend that the tree's node type, called `Node`, is declared to be an inner class. (But you can implement it anyway you like and give it any name you like.) This way, your entire source code can be self contained in a single file.

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

    private class Node { ... }
    public class Locator { ... }

    // ... any private and protected data and utility functions

    public WtLeftHeap() { ... }
    public int size() { ... }
    public void clear() { ... }
    public Locator insert(Key x, Value v) { ... }
    public void mergeWith(WtLeftHeap<Key, Value> h2) { ... }
    public Value extract() throws Exception { ... }
    public void updateKey(Locator loc, Key x) { ... }
    public Key peekKey() { ... }
    public Value peekValue() { ... }
    public ArrayList<String> list() { ... }
}
```

Efficiency requirements: (10% of the grade) The functions `insert`, `mergeWith`, and `extract` should all run time proportional to the length of the rightmost chain of the trees involved. The function `updateKey` should run in time proportional to the distance that the node needs to travel during the sifting process. (This is not guaranteed to be $O(\log n)$, since the height of the tree may be much larger than this.) The functions `size()`, `peekKey()`, and `peekValue()` should all run in $O(1)$ time. The function `list()` should run in time proportional to the number of nodes in the tree. A portion of your grade will depend on the efficiency of your program.

Style requirements: (5% of the final grade) Good style is not a major component of the grade, but you should demonstrate some effort here. Part of the grade is based on clean, elegant coding, and a reasonable amount of comments. There is no hard rules here, and we will not be picky. If we deduct points, it will because you used an excessively complicated structure to implement a relatively simple computation.

The other part is based on commenting. You should have a comment at the top of each file you submit. This identifies you as the author of the program and provides a short description of what the program does. For each function (other than the most trivial), you should also include a comment that describes what the function does, what its parameters are, and what it returns. (If you would like to see an example, check out our *canonical solution* to Programming Assignment 0, on the class Project Page.)

Testing/Grading: Submissions will be made through Gradescope. You need only upload your modified `WtLeftHeap.java` file. We will be using Gradescope's autograder and JUnit for testing and grading your submissions. We will provide some testing data and expected results along with the skeleton code.

Challenge Problem: (Remember that challenge problems are for extra credit points, which are not part of the standard grade.)

One of the dangers of locators is that they contain a link pointing to a node within your data structure. If a user makes an error and attempts to use an invalid locator, it can destroy the integrity of your data structure. Modify your program by adding an additional check to `updateKey` that verifies that the given locator is valid for the given heap. If not, your program should throw an `Exception` with the error message `"Invalid locator"`. This may necessitate changes to other heap functions as well.

There are two ways that a locator may become invalid:[1]

- An entry is extracted from the heap, and later an attempt is made to access its locator to update the key of this nonexistent entry.
- A locator `loc` points to a node that is currently in one heap (say, `h1`) but the user attempts to access it through an `updateKey` on a different heap (say, `h2.updateKey(loc, 999)`).

The test run `testEC1` tests the above functionality. You may assume that the user is not allowed to copy the contents of one locator to another.

If you do the challenge problem, add a comment to the top of your `WtLeftHeap.java` stating that you attempted the challenge problem. Also, explain how you implemented it (what was your method and which additional class objects and functions were added).

Grading will depend in part on how efficient your implementation is. For basic credit, your locator validation should run in time proportional to the height of the tree. Note, however, that a leftist tree is generally not of $O(\log n)$ height, because left-side paths can be arbitrarily long. For additional credit (extra-extra credit!), perform the validation in $O(\log n)$ time, and provide a careful explanation of how your algorithm works.

[1] There is actually a third way. All the locators become invalid when the tree is cleared. Don't worry about this case, however, since we never test it.

CMSC 420: Spring 2023

Programming Assignment 2: Sliding-Midpoint kd-Trees

Overview: In this assignment we will implement a variant of the kd-tree data structure, called a *sliding-midpoint kd-tree* (or SMkdTree) to store a set of points in 2-dimensional space. This data structure involves a number of notable features:

- It is an extended binary tree consisting of internal and external nodes. Points are stored only in the external nodes.
- Its subtrees periodically rebalance themselves. Each node maintains a counter which is incremented as insertions are made. When the counter is large enough, we rebuild the subtree completely from scratch.
- Splitting is based on a method called the *sliding-midpoint rule*, which attempts to keep cells as nearly square as possible, subject to the restriction that every cell has at least one point.

Points, Labeled Points, and Rectangles: The objects to be stored in the trees are 2-dimensional points. To save you some effort, as part of the skeleton code, we will provide you with a class for 2-dimensional points, called `Point2D.java`. This class will provide you with some utility functions, such as accessing individual coordinates and computing distances.

The points stored in your tree will be enhancements of the `Point2D` type, called an `LPoint` (for *labeled point*), which also stores a string label. An `LPoint` implements the following Java interface:

```
public interface LabeledPoint2D {
    public double getX();          // get point's x-coordinate
    public double getY();          // get point's y-coordinate
    public double get(int i);      // get point's i-th coordinate (0=x, 1=y)
    public Point2D getPoint2D();   // get the point itself (without the label)
    public String getLabel();      // get the label (without the point)
}
```

For example, given an `LPoint` called `q` with coordinates $(31.2, -7.5)$ and label `"ABC"`, the operation `q.getX()` returns 31.2, `q.getLabel()` returns `"ABC"`, and `q.getPoint2D()` returns the `Point2D` structure for the point $(31.2, -7.5)$. The `SMkdTree` is templated with the labeled point type:

```
public class SMkdTree<LPoint extends LabeledPoint2D> { /* fill this in */ }
```

We will also provide you with a class for storing axis-aligned rectangles, called `Rectangle2D.java`. This also provides a number of useful functions, such as testing whether two rectangles are disjoint or whether one contains the other.

Node Structure: The `SMkdTree` data structure is an *extended binary tree* (see Fig. 1). As mentioned above, this is an extended binary tree. Each *external node* stores a single point (that is, a reference to an `LPoint`). We allow external nodes to be *empty*, in the sense that they store no point. Each *internal node* stores the node's cutting dimension (either 0 or 1), its cutting value (a `double`), and pointers to its left and right children. Following the same convention as standard kd-trees, if a point lies on the cutting line, it is placed in the right subtree.

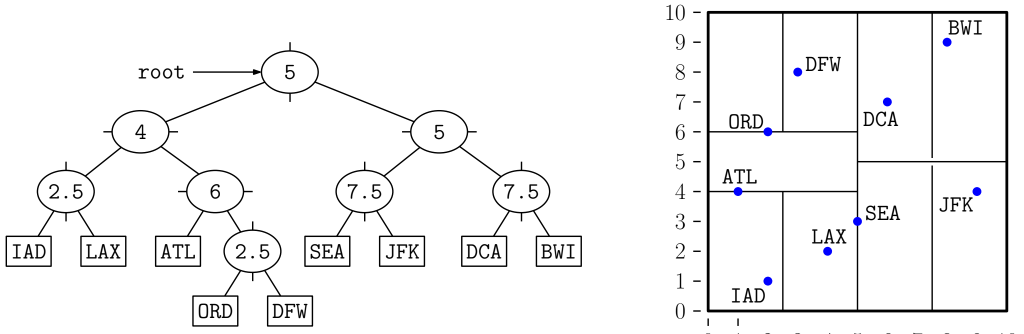


Figure 1: An example of an SMkd-Tree with bounding box $[0, 10] \times [0, 10]$.

Note that there are no `null` child pointers. Every internal node has two non-null children, which may be either internal or external nodes. External nodes have no children by definition. Even the root pointer is non-null. An empty tree is represented having the `root` point to a single external node whose point is `null`.

A natural way to handle the two different node types in Java is to create an abstract inner class, `Node`. This stores all information that is common to both node types (such as declarations of helper functions). From this we derive two subclasses, one for internal nodes, and the other for external nodes, as shown below.

```
public class SMkdTree<LPoint extends LabeledPoint2D> {

    private abstract class Node {                  // abstract node type
        abstract LPoint find(...);                 // helpers and utilities
        abstract Node insert(...);
        abstract Node delete(...);
        // ... (other abstract methods)
    }

    private class InternalNode extends Node {      // internal node
        int cutDim;                                // cutting dimension
        double cutVal;                             // the cutting value
        Node left, right;                          // children
        // ... (other data and implementation of methods)
    }

    private class ExternalNode extends Node {      // external node
        LPoint point;                              // the point (null if empty)
        // ... (other data and implementation of methods)
    }
```

The tree itself stores a pointer to the root node and a bounding axis-aligned box (i.e., `Rectangle2D`), which contains all the points. This bounding box serves as the cell for the root node. It maintains additional information, which will be discussed below.

You might observe that we don't need to store the node types. This is handled automatically by Java's inheritance mechanisms. For example, suppose that we want to invoke the `find` helper function on the root. Each node type defines its own helper. We then invoke `root.find(pt)`. If `root` is an internal node, this invokes the internal-node find helper, and otherwise it invokes the external-node find helper.

Sliding-Midpoint Rule: This splitting rule is unusual in that it prioritizes producing cells that are nearly square, rather than producing a tree of balanced height. This is useful in applications, such as finite element analysis, where the shapes of the cells are of greater importance than the height of the tree.

Suppose for now that we want to store a set S of two or more points, all of which lie within a given rectangular cell R. (Later we will see how to apply this to insert individual points.)

The sliding-midpoint rule first determines the splitting dimension as follows. It takes the longer dimension of R as the default (see Fig. 2(a)).[1] If the length and width are the same, then cut vertically.

Next, sort the points along the cutting dimension.[2] If we are in a badly degenerate situation where all the points have the same coordinates along the cutting dimension (which is easily determined by comparing the first and last points in sorted order), then this cutting dimension is no good. Instead, use the other axis as the cutting dimension (see Fig. 2(b)).[3]

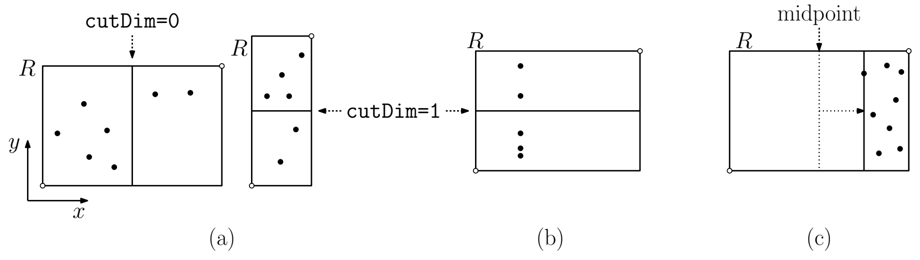


Figure 2: Sliding-midpoint splitting rule.

Now that we have selected the cutting dimension, we need to select the cutting value. The default cutting value is the midpoint of R along the cutting dimension. However, if all the points lie on one side of the midpoint, we move (or "slide") the cutting line until it coincides with the first point (see Fig. 2(c)).

By our convention that points lying on the splitting line are placed in the right subtree, it is possible to produce external nodes that contain no point (this happens to the left subtree in Fig. 2(c)).

[1]Thus, if the rectangle is wider than tall, the cut is vertical, and otherwise it is horizontal. Remember that the cutting dimension is perpendicular to the direction of the cut, so a vertical cut uses 0 (or x) for the cutting dimension, and a horizontal cut uses 1 (or y).

[2]**Hint:** Don't write your own sorter. The best way to do this is to use Java's built-in sorting function, `Collections.sort`, and create two `Comparator` objects, one that sorts by x and another by y.

[3]**Hint:** If you implemented your comparator to sort lexicographically, then you won't need to resort after changing your cutting dimension.

Bulk Creation: A useful utility program that you should implement creates a new subtree for a set of zero or more points and a given rectangular cell. If there are zero points in the set, create a single external node whose associated point is `null`. If there is one point in the set, create a single external node that contains this point. If there are two or more points in the set, apply the above sliding-midpoint rule to determine the cutting dimension and cutting value. Partition the points about the splitting line (remembering that ties are broken in favor of the right subtree). Then recursively build the left subtree and right subtree from these two subsets.[4] Finally, create a new internal node having this cutting dimension, cutting value, and these subtrees as children.

Here is a suggestion of how this function might be defined. It is passed in the points as a Java `ArrayList` and the rectangular cell, and it returns a pointer to the root node of the kd-tree that is constructed.

```
Node bulkCreate(List<LPoint> pts, Rectangle2D cell)
```

Comparing and Sorting Points: Buld creation involves sorting points, either by x or by y. You should use Java's built-in sorting function, `Collections.sort`. To instruct it how to sort, you provide it a comparison function. (In the case of partitioning for insertion, this is either lexicographically by (x, y) or lexicographically by (y, x), depending on the cutting dimension).

In Java, this is done defining a class that implements the `Comparator` interface. Such a class defines a single function, called `compare`, which compares two objects of the desired type, and returns a negative, zero, or positive result depending on which argument is larger. In our case, the objects are labeled points, `LPoint`. Here is a brief example on how you might set this up. (Some examples can be found here.)

```
private class ByXThenY implements Comparator<LPoint> {
    public int compare(LPoint pt1, LPoint pt2) {
        /* compare pt1 and pt2 lexicographically by x then y */
    }
}
private class ByYThenX implements Comparator<LPoint> {
    public int compare(LPoint pt1, LPoint pt2) {
        /* compare pt1 and pt2 lexicographically by y then x */
    }
}
```

Beware when trying to compare points for equality. Given two `LPoint` objects `p` and `q`, the operation `p == q` just checks whether they reference the same block of memory. Instead, you can use the `equals` function in the `Point2D` class. To access this, you first need to convert the labeled points into regular points, and then they can be compared: `p.getPoint2D().equals(q.getPoint2D())`.

Inserting and Deleting: To insert a new point in the sliding-midpoint tree, we first search for the point. If we find it in the tree, then we throw an exception. If not, the search ends at

[4]**Hint:** Given that you already sorted your points, a convenient way to do this is to use Java's `subList` function, which allows you to treat a Java `ArrayList` as two separate sublists. Be careful when using it. The starting index of the sublist is inclusive, but the ending index is non-inclusive.

an external node. If this node has no point, then we store the point here. If it has a point, then we create a Java `ArrayList` with two points, the existing point and the newly inserted point. We then invoke the `bulkCreate` function described above, and replace the external node with the resulting subtree.

Deleting a point is very simple. We first search for the point to be deleted. If we do not find it, we thrown an exception. Otherwise, it will be found in some external node. We simply make the node empty by setting the point reference in this node to `null`. This will produce a hole in the tree. If we are lucky, a subsequent insertion may fill in this hole. But if not, we may have many holes. We describe a process below for eliminating these holes by rebuilding the tree.

Keeping Balance: As points are inserted to and deleted, the tree will become unbalanced and may have many holes resulting from deletions. (In the context of the sliding-midpoint rule, good balance is reflected in having cells that are nearly square. The height of the tree is not important.) We balance subtrees by invoking the `bulkCreate` function to rebuilding them from scratch. We use two different mechanisms to trigger rebuilding, one for insertion and one for deletion.

For insertion, Intuitively, if the number of insertions is high relative to the subtree's size, then the tree may be out of balance. To remedy this, we rebuild the entire subtree using the `bulkCreate` function described above.

There are two ways that subtree rebuilding is triggered.

Insertion: Each internal node maintains two quantities. Its *size* stores the total number of points within the subtree rooted at this node, and its *insertion counter* stores the number of insertions that have taken place into this subtree since it was created. Each time we insert a point that lies within the subtree rooted at this node, we increment both its insertion counter and its size. When we delete a point, we decrement its size, but its insertion counter is unchanged. Rebuilding is triggered by the relationship between these quantities, as described below.

When the tree is first constructed, the user provides an integer parameter called the *rebuild offset*, which never changes. After we successfully perform an insertion of some point `p` and update the insertion counters and subtree sizes, we retrace the search path top-down from the root to the external node containing `p`, and for each node `u` on this path, we check whether

```
u.insertionCounter > (u.size + rebuildOffset)/2
```

Ignoring the rebuild offset, intuitively, this means that we have inserted sufficiently many points that this subtree may be out of balance. The purpose of the rebuild offset is to provide the user with a bit of fine control, so we are not constantly rebuilding small subtrees.

If this condition is not satisfied by any node, then there is nothing to do. Otherwise, for the first node `u` that satisfies this condition, we rebuild its subtree from scratch. To do this, we perform a traversal of the tree, storing all of its points in a Java `ArrayList`, and then we invoke `bulkCreate` on the resulting list and the existing cell for this node. This subtree replaces the current one. All internal nodes in the rebuilt subtree should have their insertion counters reset to zero, and all sizes should be correctly computed. (It is important to note that the process runs *top-down*, and it stops with the first rebuilding.)

Deletion: Deletion is handled through a more global process. We maintain a single counter, called the *deletion counter*, for the entire tree. Each time a successful deletion is performed, we increment this counter. Following a successful deletion, we check whether the deletion counter is strictly larger than the number of points currently in the tree. If so, we rebuild the entire tree.[5]

Operations: You are to fill in the missing details of the file `SMkdTree.java`, which we will provide to you. Here is the public interface.

`SMkdTree(int rebuildOffset, Rectangle2D rootCell):` This constructs a new (empty) `SMkdTree` with the given rebuild offset and bounding box.

`void clear():` This resets the tree to its initial (empty) condition.

`int size():` Returns the number of points in the tree.

`int deleteCount():` Returns the current value of the deletion counter, described above. (This is for testing and debugging purposes, since the user need not be aware of its value.)

`LPoint find(Point2D q):` Determines whether a point coordinates `pt` occurs within the tree, and if so, it returns a reference to the associated `LPoint`. Otherwise, it returns `null`.

`void insert(LPoint pt) throws Exception:` If the given point lies outside the root's bounding box (given in the constructor), this throws an `Exception` with the message `"Attempt to insert a point outside bounding box"`. If there exists a point with the same coordinates already in the tree, it throws an `Exception` with the message `"Insertion of duplicate point"`. Otherwise, it inserts the given in the tree, through the process described above. (Note that this may involve rebuilding one of the subtrees.)

`void delete(Point2D pt) throws Exception:` If there is no point with the given coordinates, this throws an `Exception` with the message `"Deletion of nonexistent point"`. Otherwise, it deletes the point from the tree by the process described above.

`ArrayList<String> list():` This operation generates a right-to-left preorder traversal of the nodes in the tree. There is one entry in the list for each node of the tree. The output is described below:

Internal nodes: Depending on whether the cutting dimension is x or y, this generates either:

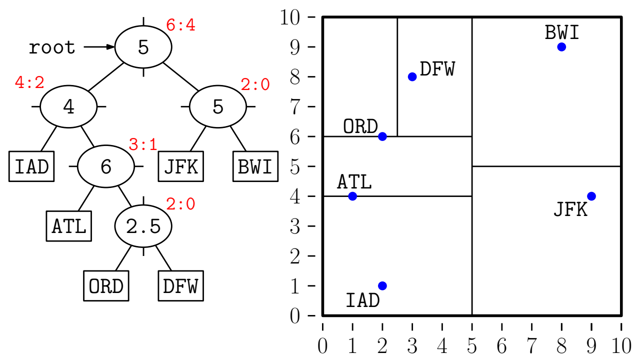
```
"(x=" + cutVal + ")␣" + size + ":" + insertCt     – or –
"(y=" + cutVal + ")␣" + size + ":" + insertCt
```

where `cutVal` is the node's cutting value, `size` is the size of the node's subtree, `insertCt` is the value of the node's insertion counter, and ␣ is a space.

[5]You might wonder why we use different mechanisms for insertion and deletion. We will discuss this when we cover the topic of *scapegoat trees*, which use a similar rebuilding process.

External nodes: If the node contains a non-null point, called `point`, this generates the string `"[" + point.toString() + "]"` (where we are using the `toString` function provided by the `LPoint` object). If the point is null, this generates the string `"[null]"`.

An example of the result of list is shown in Fig. 3(c). Our command handler will convert this into a more readable indented form as shown in Fig. 3(d).[6]



(a) (b)

```
(x=5.0) 6:4
(y=5.0) 2:0
[BWI: (8.0,9.0)]
[JFK: (9.0,4.0)]
(y=4.0) 4:2
(y=6.0) 3:1
(x=2.5) 2:0
[DFW: (3.0,8.0)]
[ORD: (2.0,6.0)]
[ATL: (1.0,4.0)]
[IAD: (2.0,1.0)]
```

(c)

```
| | [BWI: (8.0,9.0)]
| (y=5.0) 2:0
| | [JFK: (9.0,4.0)]
(x=5.0) 6:4
| | | | [DFW: (3.0,8.0)]
| | | (x=2.5) 2:0
| | | | [ORD: (2.0,6.0)]
| | (y=6.0) 3:1
| | | [ATL: (1.0,4.0)]
| (y=4.0) 4:2
| | [IAD: (2.0,1.0)]
```

(d)

Figure 3: (a) An SMkd-tree, (b) the associated subdivision, (c) the output of the `list` function, and (d) formatted output generated by the command handler. The red values associated with each node are the size and insertion counter.

`LPoint nearestNeighbor(Point2D center)`: This function returns a reference to the closest point in terms of squared Euclidean distances to the given query point. If the tree is empty, it returns `null`. If there are multiple nearest neighbors at the same distance (see Fig. 4), it returns the point that is smallest in (x, y) lexicographical order (smallest x with ties broken in favor of smallest y).

`ArrayList<LPoint> nearestNeighborVisit(Point2D center)`: This is the same as the above function `nearestNeighbor`, but it is designed for testing/debugging. It performs the same search as `nearestNeighbor`, but it returns a list of all the points visited by the search algorithm. For the sake of consistency, these points should be listed in increasing (x, y) lexicographical order.[7]

Computing distances involves computing square roots, which is both unnecessary and introduces floating-point errors. Instead, you should compute squared Euclidean distances. (This does not change the identity of the closest point). To assist you, the `Point2D` and `Rectangle2D` classes both provide a utility function `distanceSq(Point2D pt)`, which computes the squared distance from the current object to point `pt`.

[6]**Hint:** Differences between your tree and ours will first show up in the unformatted list output. Rather than trying to interpret the error there, instead look for the difference in the formatted output that is closest to the root. That will indicate which subtree is incorrect.

[7]**Hint:** You can implement both functions using just one helper function. For example, you can use the return value from the helper function to store the closest point seen so far in the search, and pass a reference to the array list of visited point as an argument. As you visit successive points, just append them to the array list. Depending on the desired result, you will either return the closest point or the sorted array list.

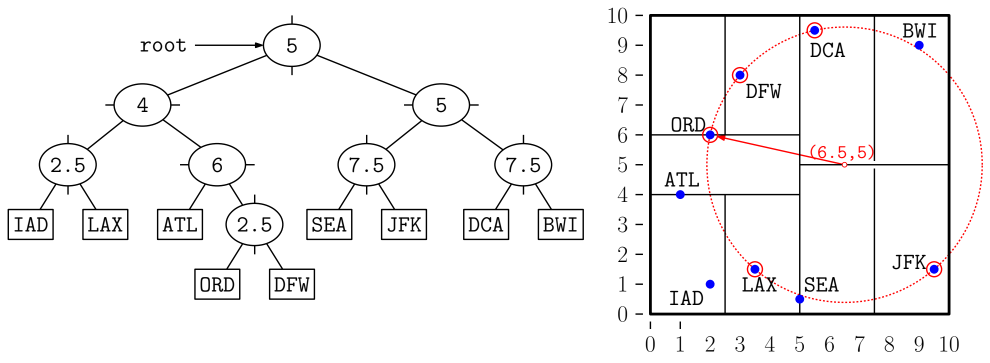


Figure 4: Nearest-neighbor query. Among the equally close candidates (`ORD`, `DFW`, `DCA`, ...), `ORD` is smallest in lexicographical order.

You should adapt the algorithm given in class to the context of extended trees. In particular, when visiting an internal node, you should first visit the subtree that is closer to the query point. Second, you should avoid visiting nodes that you can infer cannot contain the nearest neighbor. To do this, keep track of the closest point seen so far, and if node's cell is farther away from the query point than this, then you should avoid visiting the node. (Or if you visit it, you should discover this and return immediately.)

Challenge Problem: Implement an iterator for your tree, which generates all the points stored in the tree according to an inorder traversal. To do this, generate a (private) inner class that implements the Java interface `Iterator<LPoint>` (defined in Java.util). You may name this whatever you like, but for concreteness, let's call it `LPointIterator`. Your `SMkdTree` should support a public function `LPointIterator iterator()`, which returns such an iterator object. Your `LPointIterator` class should implement the following functions:

`LPointIterator()`: A constructor, which initializes the iterator.

`boolean hasNext()`: Returns `true` if there are elements remaining to be iterated.

`LPoint next()`: Returns a reference to the next point in the the tree, according to an inorder traversal. If there are no more point (that is, if `hasNext()` returns `false`), this throws a `NoSuchElementException` (which is defined in Java.util).

Note that the iterator should not return references to `null` points. If points have been deleted from your structure, your iterator should jump over them, and return the next available non-null point. Your iterator should take $O(h)$ space, where h is the height of your tree. If there are no empty external nodes, the operations should run in time proportional to the height of the tree. If there are empty external nodes, each operation can take $O(h \cdot e)$ time, where e is the number of empty external nodes you need to skip over.

Hint: A natural approach is to represent the iterator as a reference to a node in your tree, and the `next()` function advances this pointer to the next external node in inorder, skipping over empty external nodes. Implementing this in $O(1)$ space requires that you have parent links (or some other way of navigating the tree from any node). Alternatively, you can store a stack of ancestor nodes in $O(h)$ space, and use this to guide the search for the next node according to an inorder traversal. Either approach is fine for full credit.

Beware: If you are considering exploiting an auxiliary structure (such as a Java `ArrayList`) that provides its own iterator, you will need to do this within the style/efficiency requirement given above. For example, a `HashMap` supports fast insertion and deletion, but sorting it will take too long. List structures like `ArrayList` and `LinkedList` can be maintained in sorted order, but insertions and deletions are probably too slow.

Insertion/BulkCreate Example: To get a better understanding of how insertion works with bulk creation, suppose we have external node that stores the labeled point `MSP` at coordinates $(2.5, 8.5)$ (see Fig. 5(a)). It lies within a cell whose lower corner is $(2, 6)$ and whose upper corner is $(5, 10)$. We will represent rectangles by giving their lower-left and upper-right points, so this is $[(2, 6), (5, 10)]$. Suppose we want to insert a new point `SEA` with coordinates $(4.0, 9.5)$ (see Fig. 5(b)).

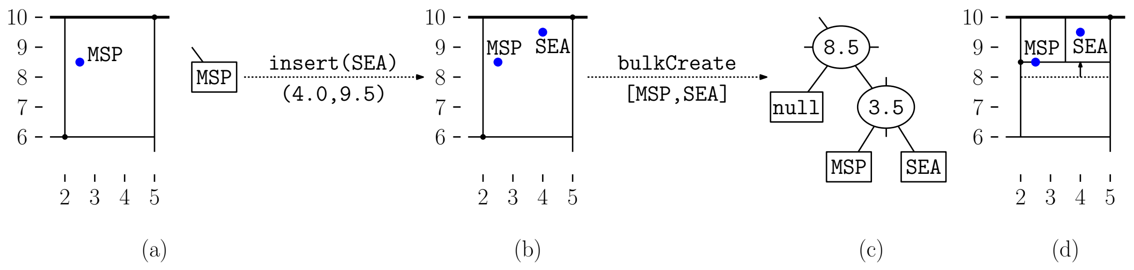


Figure 5: Insertion and bulk create. The insertion of `SEA` at $(4.0, 9.5)$ causes the external node with `MSP` to split. Given these two points, `bulkCreate` generate a a four-node subtree, which replaces the original external node.

We create a 2-element `ArrayList` storing these two points, and invoke `bulkCreate` on them with the current cell.

`bulkCreate([MSP,SEA], [(2,6),(5,10)])`: Since this cell is taller than wide, we take the cutting dimension to be 1 and split horizontally. The ideal split would be at $y = 8$, but since both points lie above this, we "slide" the splitting line up to $y = 8.5$ so it passes through `MSP`. We generate an internal node with `cutDim = 1` and `cutVal = 8.5`, and partition the points about $y = 8.5$. By our convention, the point `MSP` goes in the right subtree as does `SEA`. The left side of the partition is empty. We then invoke `bulkCreate` recursively to generate its children:

- `bulkCreate([], [(2,6),(5,8.5)])`: There are no points, so this generates an empty external node and returns.
- `bulkCreate([MSP,SEA], [(2,8.5),(5,10)])`: Since this cell is wider than tall, we take the cutting dimension to be 0 and split vertically through the midpoint at $x = 3.5$ (no sliding needed). We partition the points, with `MSP` on the left side and `SEA` on the right. We create a new internal node with `cutDim = 0` and `cutVal = 3.5`. We then invoke `bulkCreate` recursively to generate its children:
  - `bulkCreate([MSP], [(2,8.5),(3.5,10)])`: This generates a single external node with `MSP`.
  - `bulkCreate([SEA], [(3.5,8.5),(5,10)])`: This generates a single external node with `SEA`.

The final subtree is shown in Fig. 5(c) and the resulting subdivision of the cell is shown in Fig. 5(d). The insertion procedure returns this subtree to replace the original external node with `MSP`.

CMSC 420: Spring 2023

Programming Assignment 3: Clustering and the Farthest-First Traversal

Overview: In this assignment we will extend our kd-tree and heap data structures from the earlier assignments to an application in data science and machine learning. The basic assignment (which is worth 80% of the grade, excluding the style/efficiency points) involves utilities for maintaining a clustering of a set of points, and the full assignment involves an interesting algorithm called the *farthest-first traversal*. The basic assignment largely involves modifications to the `SMkdTree` data structure, and the full assignment involves both both the `SMkdTree` and the `WtLeftHeap` structure. (We will make versions of our solutions to Programming Assignments 1 and 2 available to you. You may either use yours or modify ours without penalty.)

Center-Based Clustering: Suppose that we are given a large set of points, called *sites*, which we wish cluster. Each cluster is modeled by a point called the *cluster center*. Each site is assigned to a cluster based on its closest cluster center. For example, in Fig. 1(a), we show a set of sites (black) and cluster centers (blue), and in Fig. 1(b) we illustrate the assignment of sites to centers, where the sites in each colored region are assigned to the cluster center at the middle of the region.

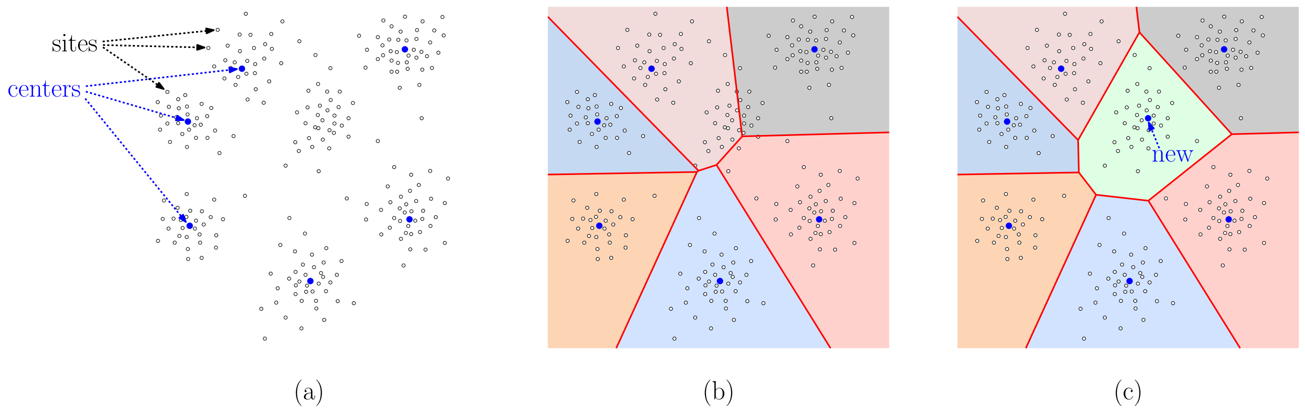


Figure 1: Center-based clustering. Each site is assigned to the cluster center that is closest to it.

In many applications, the centers are computed through an incremental process, where the centers are added one by one, based on where the need is greatest. Think of sites as phones and the cluster centers cell phone towers. Whenever a new tower is opened, many sites change their cluster membership (as is the case for the sites in the green region of Fig. 1(c)).

In many applications, cluster centers must be chosen from among the sites. We will not assume this, and in our tests, the cluster centers will always be distinct from the sites.

Basic Assignment: Cluster Assignment: In this part of the assignment, you will both modify the `SMkdTree` and implement new data structure, called `ClusterAssignment`, which maintains two point sets in $\mathbb{R}^2$, a set of sites and a set of cluster centers. It supports the insertion and deletion of sites and the insertion of cluster centers. The objective is to efficiently maintain an *assignment* of each site to its closest center.

The sites are stored in the kd-tree in the same manner as in Programming Assignment 2, but we will augment the kd-tree to efficiently process the addition of new cluster centers. Recall that each node of the tree is associated with a rectangular cell, call it `cell`. In addition, for each node we maintain a set of center points, called *contender*, which might be the closest center to some site within this cell. How do we determine which centers are contenders for a given node?

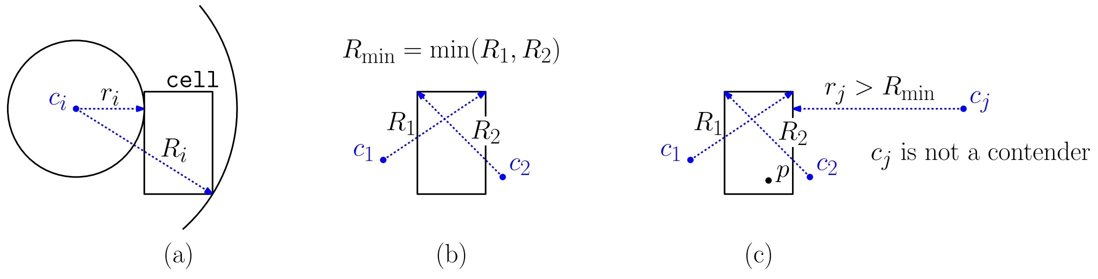


Figure 2: Contenders for the closest center.

Suppose that we have a set of existing contenders for some cell, $\{c_1, \ldots, c_k\}$. For each of these cluster centers c_i, we compute its closest squared distance to the cell, r_i, and its largest squared distance to the cell R_i (see Fig. 2(a)). For any site p in the cell, we know that $r_i \le \text{dist}^2(p, c_i) \le R_i$. Let $R_{\min} = \min_{1 \le i \le k} R_i$ be the minimum among the R_i values (see Fig. 2(b)). If for any center c_j, we have $r_j > R_{\min}$, we can infer that c_j cannot be a contender. Why? Suppose that c_1 is the contender with the smallest R value, so that $R_{\min} = R_1$. The closest that any site p of the cell could be to c_j is r_j. The farthest it could be from c_1 is R_1. Therefore, we have

$$\text{dist}^2(p, c_j) \;\ge\; r_j \;>\; R_{\min} \;=\; R_1 \;\ge\; \text{dist}^2(p, c_1),$$

which implies that p is closer to c_1 than c_j, no matter which point p is selected within the cell. (Note that we cannot infer that c_1 is p's closest center, only that c_j *not* its closest center).

Augmenting the kd-tree: We will augment our kd-tree as follows. For every node (both internal and external) we will maintain a list of closest-center contenders as described above. (Because we will need to both insert and delete entries, it is recommended that you implement this as a Java `LinkedList`, rather than an `ArrayList`.) For example, this could be made a member of your general kd-tree node, so that both internal and external nodes will inherit it.

```
public class SMkdTree<LPoint extends LabeledPoint2D> {

    private abstract class Node {
        LinkedList<LPoint> contenders; // list of this node's contenders
        // ... other Node stuff
    }

    class InternalNode extends Node { /* ... */ }
```

```
        class ExternalNode extends Node { /* ... */ }
        // ... and so on
    }
```

Every node will have at least one contender. To guarantee this, even when the tree is empty, the tree constructor will be given an inition starting center, `startCenter`. Initially, the tree consists of an empty leaf node whose cell is `rootCell`, and whose associated contender list contains a single entry `startCenter`.

Adding a Center: A new center, called `center`, is added as follows. Observe that if a center is a contender for some node, it is automatically a contender for all its ancestors. (We'll leave this as an exercise for you.) This means that we can employ a top-down process for filtering this center into all the nodes for which it is a contender.

The process works as follows. We create a recursive helper to do this, which will be a member function of the internal and external node classes. The initial call is at the root node. We pass in the new `center` and the root cell. In general, let `u` denote the current node being visited in the recursive process.

- Iterate through the points of `u.contenders` and for each, compute its R_i value,[1] that is, the largest squared distance from this center to the current node's cell. Save the minimum value among all of these, including the R value for `center` itself, as $R_{\min}$.
- Iterate through the elements of `u.contenders` a second time. For each compute its r_i value,[2] that is, the closest squared distance from this center to the current node's cell. If for any contender, r_i is strictly greater than $R_{\min}$, we remove the center corresponding to r_i from the contender's list.
- Letting r_{new} denote the r_i value for the new center, if $r_{\text{new}} \leq R_{\min}$, add `center` to `u.contenders`. Further, if this is an internal node, apply the function recursively to add `center` to its left and right subtrees (using `leftPart` and `rightPart` to determine their associated cells). On the other hand, if $r_{\text{new}} > R_{\min}$, we are done and simply return.

The above process computes the R_i and r_i values each time we visit the node. For greater efficiency, it would be a good idea to compute these values once, and save them in the cell. We can also compute $R_{\min}$ and update it incrementally. (This does not affect the asymptotic time, however, because we will still need to traverse the list to determine which contenders to delete.)

You might wonder why we need to maintain a full list of contenders for each external node. Why not just compute the *exact* closest center to the site stored in this node. The reason is for the sake of future insertions and rebuildings, which we discuss next.

Deleting a Center: For this assignment, we *do not* delete centers, but see the challenge problem below.

[1]To help you, we have added a function to the `Rectangle2D` class, `maxDistanceSq`, which compute the maximum squared distance between a point and a rectangle.

[2]You can use the function `distanceSq` in the `Rectangle2D` class.

Inserting/Deleting Sites and Rebuilding: Inserting and deleting sites is essentially the same as in Programming Assignment 2, but some changes are needed to deal with the contenders. Because contenders depend only on the cell, not the points within the cell, the insertion and deletion of sites does not affect the node contenders.

However, rebuilding a subtree will generate new nodes with new cells, and hence with new contenders. The approach is quite simple, however. Whenever we rebuild the subtree rooted at some node `u`, we save a reference to its contender list, `u.contenders`. We then employ the rebuilding process from Programming Assignment 2. Finally, we iterate through the saved contender list, and add each of these centers one by one. (There may be more efficient ways to process them as a batch, but this is sufficient for full credit.)

Basic-Assignment Requirements: The basic part of the assignment will involve creating a new class for storing sites, centers, and assignments, called `ClusterAssignment`. To implement this, it will also be necessary to make modifications to your `SMkdTree`.

Let's first describe this new class, called `ClusterAssignment`. Its job is to maintain a dynamic collection of sites and cluster centers and to keep track of the assignment of each site to its closest cluster center. It supports inserting and deleting sites and adding new centers. It has the same generic structure as the `SMkdTree`:

```
public class ClusterAssignment<LPoint extends LabeledPoint2D> { /* ... */ }
```

At a minimum, the private data for the class consists of a kd-tree for storing the sites and a list for storing the centers. (You will probably need to add additional data items as well.)

```
private SMkdTree<LPoint> kdTree; // storage for the sites
private ArrayList<LPoint> centers; // storage for the centers
```

The `ClusterAssignment` class supports the following public functions:

`ClusterAssignment(int rebuildOffset, Rectangle2D bbox, LPoint startCenter):` The constructor creates a new structure with an empty kd-tree, which will eventually hold the sites and it initializes the cluster center set to consist of the single cluster point `startCenter`.

`void addSite(LPoint site):` Inserts a new site in the structure. This will involve inserting the site in the kd-tree, which may throw an exception if the site is a duplicate of an existing site. (If any new nodes are created, their contender lists must be created as well.)

`void deleteSite(LPoint site):` Deletes an existing site from the structure. This will involve deleting the site from the kd-tree, which may throw an exception if the site does not exist. (If this induces a rebuild, the contender lists for all the nodes of the tree will need to be generated.)

`void addCenter(LPoint center):` This adds a new center point. The point is added to the list of centers and it is processed through the kd-tree, adding it to all the nodes for which this point is a contender.

`int sitesSize():` Returns the number of sites in the structure. (This just invokes `size()` on the kd-tree.)

`int centersSize()`: Returns the number of centers in the structure. (This just invokes `size()` on the list of center points.)

`void clear()`: Removes all the sites by clearing the kd-tree. It also removes all the centers, except `startCenter`, given in the constructor.

`ArrayList<String> listKdWithCenters()`: This just invokes `listWithCenters()` (described below) on the kd-tree.

`ArrayList<String> listCenters()`: This produces an `ArrayList` of strings one per center, where each string contains the label and coordinates of the center. The list should be sorted in alphabetic order. An example is shown below, left.

```
listCenters():            listAssignments():
------------------------- ------------------------------
BWI: (80.0,80.0)          [DCA->BWI] distSq = 181.0
ORD: (20.0,60.0)          [ATL->ORD] distSq = 200.0
PVD: (90.0,20.0)          [JFK->PVD] distSq = 400.0
...                       ...
```

`ArrayList<String> listAssignments()`: This produces an `ArrayList` of strings, one per site. Each string contains the site, the associated closest center, and the squared distance between the site and center. The list should be sorted in increasing order of squared distance. If there are duplicates, ties should be broken lexicographically by the (x, y) coordinates of the site. (An example is shown above, right. The first line means that the site `DCA`'s closest center is `BWI`, and the squared distance between these two points is 181.)

We can compute this as follows. First, we perform a traversal of the kd-tree. (Pick any traversal order you like.) On arriving at a non-empty external node `u`, we access its list of contenders `u.contenders`, and select the one that minimizes the squared distance to `u.point`. If there are ties for the closest, take the contender that has the lexicographically minimum (x, y) coordinates. Create a triple consisting of the site, the closest center, and the squared distance, and add it to the list of assignments. On returning, sort this list by distances, breaking ties as described above.

How to implement these triples? A good way to do this is to implement a new class, call it `AssignedPair`. (This could be a standalone public class in `AssignedPair.java` or an inner class within `ClusterAssignment`, its up to you.) Each instance of this class contains a site (`LPoint`), a center (`LPoint`), and the squared distance between them (`double`). This class should "`implement Comparable<AssignedPair>`". This means that it must implement a public function "`int compareTo(AssignedPair o)`", which compares this assigned pair to another one. This compare function should be based primarily on the squared distance, but if two distances are equal, it breaks the tie based on the lexicographical order of the sites. (This always succeeds, because each site appears only one assignment, and no two sites have the same coordinates.) Because this class implements the `Comparable` interface, you can feed it into `Collections.sort` to obtain the sorted list.

Next, let us present the possible modifications to the kd-tree. You will need to make changes to handle the processing of center points. *The following are suggestions.* Since these functions are just going to be used internally by your `ClusterAssignment` and `FarthestFirst` classes, you may add/remove/modify these however you see fit.

`SMkdTree(int rebuildOffset, Rectangle2D rootCell, LPoint startCenter):` The constructor has been modified to include a new argument, `startCenter`. As described earlier, this is the initial center. (This is mainly a convenience, which allows you to assume that the contender lists are always non-empty.) The initial structure consists of a single empty external node whose contender list contains a single entry `startCenter`.

`void addCenter(LPoint center):` Adds `center` as a new center and updates all affected contender lists appropriately. You may assume that `center` is distinct from any of the sites that are currently in the tree (in both its label and coordinates). Note however, that it is possible to delete a site, and add it later as a center. (My version of this function returned an array-list of sites whose center had changed, but you are free to do whatever you like.)

`void clear():` Clears the kd-tree, removing all the sites. It also removes all the centers, except `startCenter`, given in the constructor.

`ArrayList<String> listWithCenters():` This is identical to the standard `list` function, but for each node we also include a list of the contenders (just the labels) in curly braces. For the sake of consistency, order the list alphabetically by the label. (The most efficient way to do this is to maintain the contenders lists sorted by label, but you will get full credit if you just invoke `Collections.sort` to sort them each time you perform this operation.) We show an example below, where we have added centers `BWI`, `ORD`, and `PVD`.

```
list():                   listWithCenters():
------------------------- -------------------------------
(x=50.0) 6:4              (x=50.0) 6:4 => {BWI ORD PVD}
(y=50.0) 3:1              (y=50.0) 3:1 => {BWI ORD PVD}
(x=70.0) 2:0              (x=70.0) 2:0 => {BWI PVD}
[DCA: (70.0,71.0)]        [DCA: (70.0,71.0)] => {PVD}
[SEA: (50.0,51.0)]        [SEA: (50.0,51.0)] => {ORD PVD}
...                       ...
```

When we are working with much larger examples, the contender lists can be quite long. If there are 11 or more contenders, just list the first 10 in alphabetical order and add "`...`" at the end, for example.

```
(y=0.0) 30:0 => {ABJ ABV AKR AZR BHX BLI BNI BOY BSK BYK...}
```

Full Assignment: Farthest-First Traversal: Modern data sets are massive. An important step in analyzing such large sets involves *sketching*, where the original data set is represented by a much smaller subset. The most common approach to sketching is through random sampling, but there are times when random sampling is not the best option.

Consider, for example, an application where the point set consists of all the cell towers in the USA, and the question under consideration is "What is the farthest distance from any phone in the USA to the nearest cell tower?". A random sample of cell towers would heavily sample urban regions where there are lots of phones and lots of towers, but it would miss large rural regions where there are few (see Fig. 3(a)). For this particular query, a better approach would be to distribute the samples based instead on distance. In particular, we want our sample to be as close as possible to every cell phone user in the USA (see Fig. 3(b)).

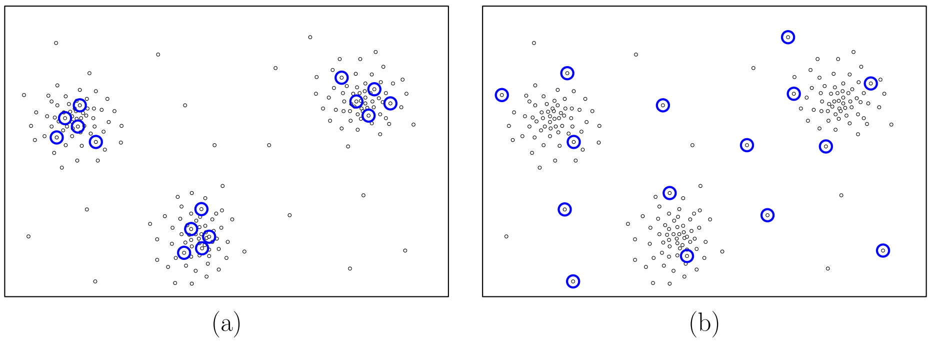


Figure 3: Sketching via (a) random sampling versus (b) distance-based sampling.

A common way to produce such a sketch is based on a concept called the *farthest-first traversal*. Given a set S of n points such a traversal visits every point of the set exactly once. The initial point of the traversal, call it $c_0 \in S$, can be chosen arbitrarily. The next point is the point of S that is farthest from c_0. The next is the point that maximizes the distance to the closer of c_0 and c_1. Generally, suppose that we have already chosen some subset $C_k = \{c_0, \ldots, c_{k-1}\}$. The next point of the traversal is the point $s \in S \setminus C_k$ that is a far away as possible from any of the points chosen so far, or more formally

$$c_k \;=\; \text{argmax}_{s \in S} \left\{ \min_{0 \le i < k} \text{dist}(s, c_i) \right\},$$

where "argmax" means point that achieves the value of the stuff inside the curly braces. The final sequence $\{c_0, \ldots, c_{n-1}\}$ is called the *farthest-firt traversal* of S (given the starting point c_0). Intuitively, the farthest-first traversal attempts to select the points so that in any initial subsequence, the points are spread out as much as possible. (It does not succeed in this formally because this is an NP-hard problem, but it is a provably good approximation. This is related to a famous approximation algorithm for k-center clustering, called *Gonzalez's algorithm*.)

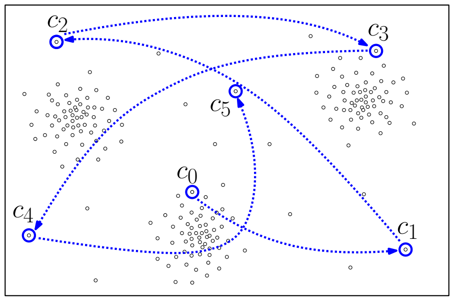


Figure 4: The first six points in a farthest-first traversal starting at c_0.

What does this have to do with distance-based sketching? Well, a provably good way to obtain a sketch of any desired size m is to compute the entire farthest-first traversal, and then select the first m points of the traversal. The purpose of Part 2 of the assignment is to implement an efficient algorithm for computing the farthest-first traversal.

Full Assignment: Farthest-First Traversal: For the full assignment, you will extend the basic part by implementing a class `FarthestFirst`, which implements an efficient algorithm for the farthest-first traversal of a point set. This will involve a spatial index (the `SMkdTree`) and a priority queue (the `WtLeftHeap` from Programming Assignment 1). It will also make use of much of the functionality of the `ClusterAssignment` class.

The algorithm is given a point set S, which we call the *sites*, and an starting point c_0, which we will call `startCenter`. The sites are inserted into a kd-tree, and c_0 is the initial cluster center for all the points. The algorithm proceeds by selecting one of the sites, removing it from the kd-tree, adding it to the existing set of cluster centers (the points in the traversal), and updating the closest cluster center for all the sites. Thus, at any time in the process, the cluster centers consist of the points that have been added to the traversal, and the sites consist of the points that are waiting to be added to the traversal.

We have almost everything we need to implement this algorithm efficiently. The insertion and deletion of sites is handled by the kd-tree. Centers can be inserted by invoking the `addCenter` function of the `ClusterAssignment` structure. This in turn causes the kd-tree to update the closest center contenders for each node.

There is only one operation remaining, that of finding the site that has the largest distance to its closest cluster center. It would too inefficient to generate all the site-center assignments and select the one with the largest distance. Instead, we will use our `WtLeftHeap` data structure to store all the site-center assignments, sorted by the squared distance between the site and its associated center.

Recall that when discussing `listAssignments()` above, we suggested defining an inner class `AssignedPair` within `ClusterAssignment` for storing site-center-distance triples. We can create an instance of `WtLeftHeap`, where the keys are squared distances (`double`) and the values are site-center pairs (`AssignedPair`).[3] The heap allows us to efficiently extract the assigned pair with the maximum distance. When we add the new center point to the traversal, a number of sites may discover that they are closer to the newly inserted center. To handle this, we need to perform an `updateKey` operation on all of these sites. This means that we need make use of locators to identify where each assigned pair appears in the heap. We need a place to store these locators. Our solution will be to use a Java `HashMap` to map site labels to heap locators.

We can describe a single step of the farthest-first traversal. Initially, we have the starting center c_0 and a collection of points (sites) stored in a kd-tree. (The point c_0 itself is *not* stored in the kd-tree.) In general, we will have computed the points in a partial traversal $C_k = \{c_0, c_1, \ldots, c_{k-1}\}$. Call these points the *centers*. The remaining points of S, called *sites*, are stored in the kd-tree. Every site is assigned to its closest center, and each of these

[3]You might wonder why we chose to use the squared distance as the key, rather than just using the comparator within the `AssignedPair` class. The latter is really the better approach, because it provides us with automatic tie-breaking when squared distances are equal. However, there are subtle programming issues that can arise when the key and value are the same. We have decided to avoid these issues by keeping keys and values distinct. We will avoid distance ties in our test data.

site-center pairs has an associated `AssignedPair` for it. These pairs are stored in a max heap, sorted by the squared distance between the site and its center. Finally, every entry in the heap has a locator, and these locators are stored in the Java `HashMap`, where the key is the site's label. We proceed as follows.

- If there are no more remaining sites (the kd-tree is empty), we are done, and the traversal is complete.
- Otherwise, extract the site-center pair from the heap with the maximum squared distance. Let s denote the associated site.
- Delete s from the kd-tree. This former site now becomes a new center, called c_k.
- Add c_k to the end of the traversal.
- Update kd-tree contenders appropriately for the addition of c_k. Let $U = \{s_1, \ldots, s_m\}$ denote sites of the kd-tree whose closest center has changed to c_k as a result. For each element s_i in U, do the following:
  - Use s_i's code to access its entry in the hash-map to obtain its heap locator.
  - Using this locator, update the key of s_i's heap entry to the squared distance between s_i and c_k.

Full-Assignment Requirements: For the full assignment, we will create a new class, called `FarthestFirst`. You can start by making a copy of `ClusterAssignment` and make modifications to it. This class has the same generic structure as the `SMkdTree`:

```
public class FarthestFirst<LPoint extends LabeledPoint2D> { /* ... */ }
```

At a minimum, the private data for the class consists of a kd-tree for storing the sites and a list for storing the centers. (You will probably need to add additional data items as well.) As mentioned earlier, the class `AssignedPair` can be either be an inner class or a standalone class. It's up to you.

```
public class FarthestFirst<LPoint extends LabeledPoint2D> {
    private class AssignedPair implements Comparable<AssignedPair> {
        private LPoint site; // the site
        private LPoint center; // its assigned center
        private double distanceSq; // squared distance to the assigned center
        // ... addtional data as needed
    }

    private LPoint startCenter; // the initial center
    private WtLeftHeap<Double, AssignedPair> heap; // heap for assigned pairs
    private HashMap<String, ...> map; // a map used for saving heap locators
    private SMkdTree<LPoint> kdTree; // kd-tree for sites
    private ArrayList<LPoint> traversal; // the traversal
```

The role played by the set of centers in the basic assignment is taken by the `traversal` object. Here are the following public functions of `FarthestFirst`:

`public FarthestFirst(int rebuildOffset, Rectangle2D bbox, LPoint startCenter):` The constructor creates a new structure with an empty kd-tree, which will eventually hold the sites and it initializes the traversal to consist of the single cluster point `startCenter`.

`void addSite(LPoint site):` Inserts a new site in the structure. This will involve inserting the site in the kd-tree, which may throw an exception if the site is a duplicate of an existing site. (If any new nodes are created, their contender lists must be created as well.) Let `center` denote its closest center point. We create a new `AssignedPair` consisting of (`site`, `center`) which we add to the heap, where the key is the squared distance between them. We save the locator in the map.

`LPoint extractNext():` Performs one step of the farthest-first traversal (described in the bulleted list above). If there are no remaining sites (the kd-tree is empty), it returns `null`. Otherwise, it performs the various described actions (deletion from the kd-tree, adding to the traversal, updating the kd-tree contenders and the updating the keys of affected sites). Finally, it returns the point that has just been added to the traversal.

`int sitesSize():` Same as for `ClusterAssignment`.

`int traversalSize():` (Analogous to `centersSize` for `ClusterAssignment`.) Returns the number of centers in the traversal. (This just invokes `size()` on the traversal list.)

`void clear():` Clears everything as in `ClusterAssignment` and also clears the hash map, the heap, and the traversal.

`ArrayList<String> listKdWithCenters():` Same as for `ClusterAssignment`.

`ArrayList<LPoint> getTraversal():` This simply returns the `ArrayList` containing the points of the traversal.

`ArrayList<String> listCenters():` Same as for `ClusterAssignment`, except using the traversal rather than the list of centers.

`ArrayList<String> listAssignments():` Same as for `ClusterAssignment`.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You will need to fill in the implementations of the `SMkdTree.java`, `WtLeftHeap.java`, `ClusterAssignment` and (for the full assignment) `FarthestFirst`.

As in Programming Assignment 2, we will provide utilities (e.g., `Point2D` and `Rectangle2D`). You should not modify any of the other files, but you can add new files of your own. For example, you may want to create additional classes, like `AssignedPair.java`, or if you want to add additional functions to any of the classes, such as `Point2D` or `Rectangle2D`, it would be preferable to create an entirely new class (e.g., `MyRect2D`), which you will upload with your submission.

As with the previous assignment, the package "`cmsc420_s23`" is required for all your source files. As usual, we will provide a driver programs (tester and command-handler) for processing input and output. You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

Efficiency requirements: (10% of the final grade) Your functions for updating contender lists should operate efficiently. Imagine that your kd-tree has millions of points and your contender lists can contain thousands of entries. Java provides built-in functions to perform operations on basic data structures (array-lists, linked-lists, hash-maps, etc.) but these are not all equally efficient. Based on your knowledge of data structures, you should be able to determine how efficient these functions are and structure your program to apply them appropriately.

Also, your program should avoid visiting nodes that cannot possibly be affected by the operation. For example, if you are inserting a new center, you should only visit nodes that are close enough to the newly added center that they might reasonably need this center as a contender. We are not going to apply a rigid policy in grading, but we will deduct points if your algorithm visits nodes that are clearly irrelevant to the operation.

Style requirements: (5% of the final grade) The style requirements are essentially the same as for the previous assignment. Your code should be relatively clean, and comments should be provided to explain important functions. You may refer to our canonical solutions for guidance (but you do not need to be quite as excessive in commenting as I am).

Testing/Grading: As always, we will provide some sample test data and expected results along with the skeleton code.

As before, we will be using Gradescope's autograder for grading your submissions. Unless you created your own additional files, you should only submit only the four files mentioned above (`SMkdTree.java`, `WtLeftHeap.java`, `ClusterAssignment.java` and (for the full assignment) `FarthestFirst.java`). If you created any additional files for utility objects, you will need to upload those as well.

Challenge Problem: For the class `ClusterAssignment`, implement a new function,

```
void deleteCenter(LPoint center) throws Exception
```

that deletes the given center point. If this point does not exist among the current set of centers, an `Exception` is thrown with the message `"Deletion of nonexistent center"`. This removes `center` from all the contender lists in the kd-tree, and updates these lists. Note that deleting a center can increase the $R_{\min}$ value for a node, and this can result in "resurrecting" contenders that were filtered out by `center` earlier in the process.

At the top of your submission, add a comment indicating that you attempted this operation, and present a clear explanation (in plain English) of how you implemented this function.

Hint: For full credit, this should be implemented in an efficient manner. There is no clear optimum strategy for doing this, and any solution is likely to involve tradeoffs. There are some pitfalls you should avoid.

- First, deleting a center in one region of the kd-tree should not require visiting all the nodes of the kd-tree. Ideally, your code should only need to visit the nodes where `center` was among its list of contenders. (Or, if it does visit a node that does not contain `center` among its contenders, it should discover this and not recurse on its children.)
- Second, as mentioned above, deleting `center` from the contender list of some node may result in a need to resurrect contenders that were filtered out earlier. It would be inefficient to consider all possible centers as candidates for resurrection. (As an

example, if you remove a cell phone tower in Maryland, you should not need to consider cell phone towers in California as candidates for replacing it.) Consider more efficient ways to identify a smaller subset of local contenders.

This is an open-ended design problem, and there really is no single correct answer. Explain in your leading comment what strategy you decided upon.

CMSC 420: Spring 2023

Homework 1: Basics, Union-Find, and Heaps

Problem 1. (10 points) This question involves the rooted trees shown in Fig. 1.

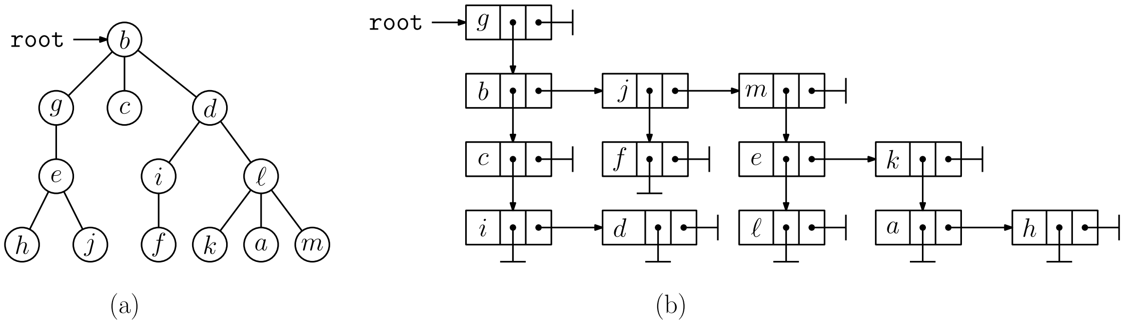


Figure 1: Rooted trees.

(a) (3 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the *first-child/next-sibling form*.

(b) (2 points) List the nodes Fig. 1(a) in *preorder*.

(c) (2 points) List the nodes Fig. 1(a) in *postorder*.

(d) (3 points) Consider the rooted tree of Fig. 1(b) given in its first-child/next-sibling form. Draw a figure in its *standard form*.

Problem 2. (10 points) Consider the union-find trees shown in Fig. 2. The `rank` values for each tree are indicated in blue next to each root.

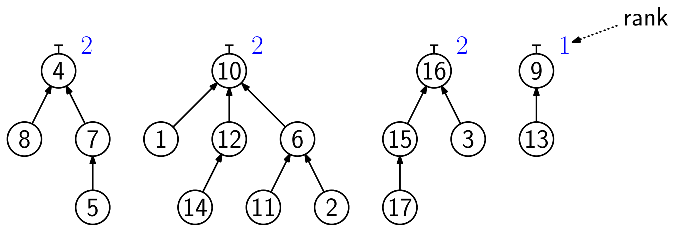


Figure 2: Rooted trees.

(a) (3 points) Show the array of `parent` indices for this set of trees and indicate which elements of this array are *set identifiers*. (See Fig. 1(b) of the latex notes for Lecture 4 as an example of what we are looking for.)

(b) (5 points) Show results after of performing the operations `union(4,10)` and `union(16,9)`. (For the sake of uniformity, use the same convention given in the lecture notes. When performing `union(s,t)`, if both trees have the same rank, link `s` as a subtree of `t`.)

(c) (2 points) Show the result of performing the operation `find(5)` on the data structure *after* the union operations of part (b). In particular, indicate the resulting set identifier returned by the find operation and show the final tree that results after path compression is applied.

Problem 3. (10 points) Consider the two leftist heaps, H_1 and H_2, shown in Fig. 3.

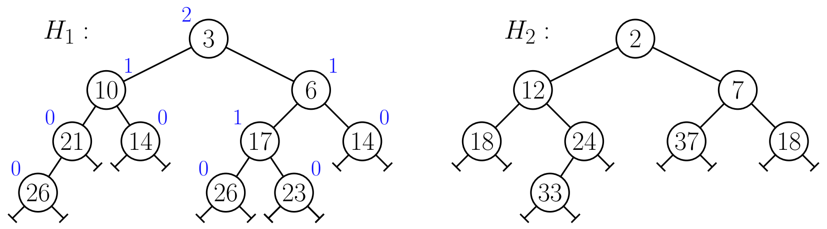


Figure 3: Leftist Heaps.

(a) (3 points) We have labeled each node of H_1 with its `npl` value. Redraw H_2 indicating the `npl` values for each node.

(b) (7 points) Show the result of merging these two heaps together. Indicate the `npl` values for each node. (You need only show the final tree, but the intermediate tree may be given for partial credit.)

Problem 4. (10 points) A useful operation on trees is that of changing the root of the tree, while maintaining all the existing connections. Suppose that you are given a rooted, multiway tree presented using the first-child/next-sibling representation as shown below. The member `root` points to the root node of the tree. For this problem, you may assume that the root is non-null and it has at least one child.

```
class Tree {                    // a rooted multiway tree
  class Node {                  // a node of the tree
    String data;
    Node firstChild;
    Node nextSibling;
  }
  Node root;                    // the root node
  ...
}
```

You are asked to implement a function `promote(Node v)`. This function is given a reference to a node `v`, which is one of the children of the root node. Let `r` denote the root. This function makes `v` the new root of the tree by (1) removing `v` as child of `r` and (2) making `r` the first child of `v` (see Fig. 1). All the other children of `v` and all the other children of `r` remain unchanged. If we think of the tree as an undirected graph, the graph structure is unchanged. We just have a different node as the root.

Present pseudocode for this function. Briefly explain how it works. For full credit, your algorithm should run in time proportional to the number of children the root has. You do not need to perform any error checking. You may assume that `v` is indeed one of the children of the root node.

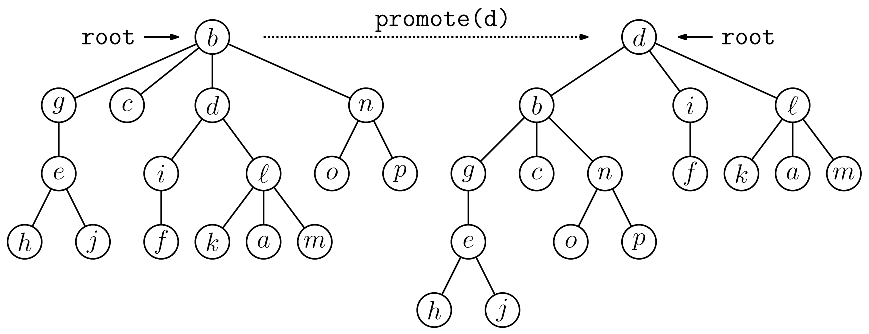


Figure 4: Promoting a child of the root to be the new root.

Problem 5. (10 points) In this problem we will analyze the amortized complexity of a new data structure, called a *double stack*. The idea is to store two separate stacks within a single array `A`. Let `m` denote the current size of this array. We split the array into two subarrays, one of size `s` and the other of size `m` − `s`. The *lower stack* occupies entries `A[0..s-1]` and the *upper stack* occupies entries `A[s..m-1]` (see Fig. 5(a)). Initially `m` = 4 and `s` = 2, but whenever we expand the array, we will adjust these two values.

Both stacks grow (through pushes) and shrinks (through pops) within their respective subarrays. Each push or pop costs +1 units. We continue until we encounter one of two overflow events:

Lower-stack overflow: A push to the lower stack, when it contains s entries (see Fig. 5(b)).

Upper-stack overflow: A push to the upper stack, when it contains $m - s$ entries (see Fig. 5(c)).

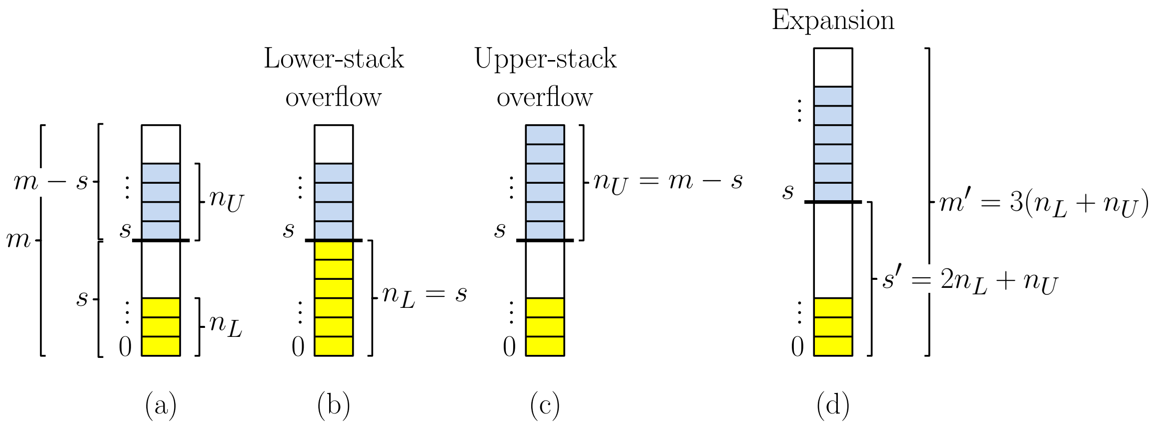


Figure 5: The double stack.

When either of these events occur, we trigger an *expansion*, which works as follows. Let n_L and n_U denote the number of elements currently in the lower and upper stacks, respectively, just before the push. We allocate a new array of size $m' = 3(n_L + n_U)$, and we set $s' = (2n_L) + n_U$. We copy the elements of the lower stack into entries $[0..n_L - 1]$, and we copy the elements of the upper stack into entries $[s'..s'+n_U-1]$. Note that the number of available entries in the lower stack is $s'-n_L = n_L+n_U$, and the number of available entries in the upper stack is $m'-(s'+n_U) = n_L+n_U$. Thus, both stacks have an equal amount of space in which to expand. After this, we push the new element onto the specified stack.

The cost of the expansion is n_L+n_U, accounting for the time to copy the elements from the old array into the new array. To this we add $+1$ for the actual push operation itself.s

As an example, suppose that $m=12$, $s=4$, $n_L=4$ and $n_U=2$, and suppose that we perform a push into the lower stack (see Fig. 6). This push will cause the lower stack to overflow. We allocate a new array of size $m'=3(n_L+n_U)=18$, and we set $s'=(2n_L)+n_U=10$. We then copy the four elements from the lower stack to entries $[0..3]$ and we copy the two elements of the upper stack to the entries $[10..11]$. At this point, both stacks can add another $n_L+n_U=6$ elements before they overflow. The cost for the expansion is $n_L+n_U=6$ (since this is the number of elements copied). Finally, we push the new item into entry $[4]$ of the lower stack at a cost of $+1$. Thus, the entire expansion has cost us $6+1=7$ units.

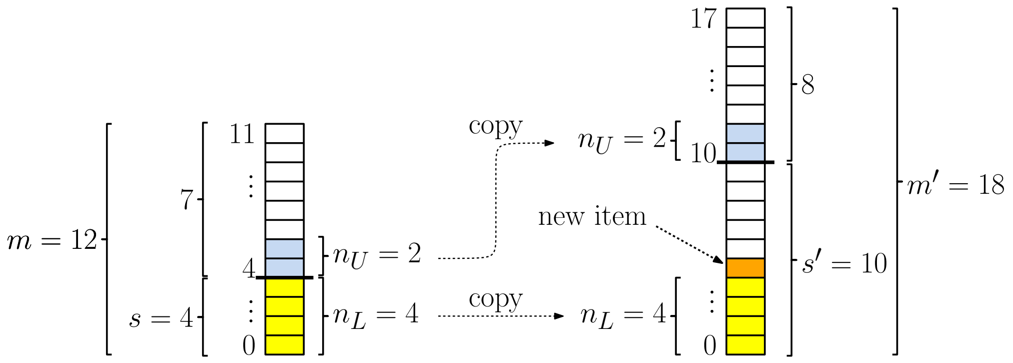


Figure 6: Example of double-stack expansion.

In this problem, we will derive the amortized running time of the double stack. As in class, we will use a token-based argument. We start with the initial configuration, with $m=4$, $s=2$, and an empty stack, so $n_L=n_U=0$. We break the sequence of operations (pushes and pops) into into subsequences called *runs*. The first run starts with the initialization and ends with the first expansion. In general, each run starts just after the prior expansion and runs until the next expansion. (The final run may not end in an expansion, but this is good, since all the operations of this run cost just $+1$ each.)

Answer the following questions:

(a) (2 points) Suppose that we have just performed an expansion. Prior to the expansion we had an array of size m, and our new array is of size $m'=3(n_L+n_U)$. As a function of m' (or if you prefer, n_L and n_U), what is the minimum number of operations needed until the next expansion occurs? (Briefly explain.)

(b) (2 points) As a function of m' (or if you prefer, n_L and n_U), what is the worst-case (maximum) cost for the next expansion? (**Hint:** This may depend on the relative sizes of the two stacks.)

(c) (6 points) Using parts (a) and (b), derive a constant τ such that the amortized cost of our expanding dual stack is at most τ. (**Hint:** Note that the worst-cases for (a) and (b) may arise from different scenarios. For the sake of obtaining an upper bound on the amortized cost, it is okay to simply use the worst-case for each, without regard for whether they can both happen simultaneously. See Challenge Problem 2.)

Note: Challenge problems are not graded as part of the homework. The grades are recorded separately. After final grades have been computed, I may "bump-up" a grade that is slightly below a cutoff threshold based on these extra points. (But there is no formal rule for this.)

Challenge Problem 1: You have two friends, Alice and Bob. They have just implemented the `merge` function for leftists heaps, but they each made one mistake. For each of the following, indicate what the consequences are of their error.

(a) When computing the `npl` values for node, Alice consistently used the *maximum* of the `npl` values of the two children (rather than the minimum). That is, she defined

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

Otherwise, her code is exactly the same as given in class.

Which of the following could be said of Alice's program. Select one and justify your answer.

(i) No difference at all. The tree structure is identical, the results are identical, and the running time to perform the `merge` operation is $O(\log n)$.

(ii) The tree structure may differ, but it is still in valid heap order, and the running time to perform the `merge` operation is $O(\log n)$.

(iii) The tree structure may differ, but it is still in valid heap order. However, the running time to perform the `merge` operation may be worse than $O(\log n)$.

(iv) The tree structure may fail to be in heap order.

(b) Bob made a different mistake. Whenever he checks the `npl` values of two subtrees, he always puts the subtree with the *larger* `npl` value on the right (rather than on the left). Otherwise, his code is identical. (In particular, he merges trees along their rightmost path.) Repeat part (a), but with this variant. Again, justify your answer.

Challenge Problem 2: It was noted in Problem 5(c) that the simple analysis based on the worst-case scenarios for parts (a) and (b) cannot both occur. Present a more accurate analysis of the amortized running time that corrects this shortcoming.

CMSC 420: Spring 2023

Homework 2: Search Trees

Problem 1. (8 points) Perform the following operations on the AVL trees shown in Fig. 1. In each case, show the final tree and list (in order) all the rebalancing operations performed (e.g., "`rotateLeftRight(7)`"). (We only need the final tree, but intermediate results may be shown for the sake of assigning partial credit.) Draw your final tree as in Fig. 1(b) from Lecture 7. Show the balance factors at each node. (Don't bother giving the heights.)

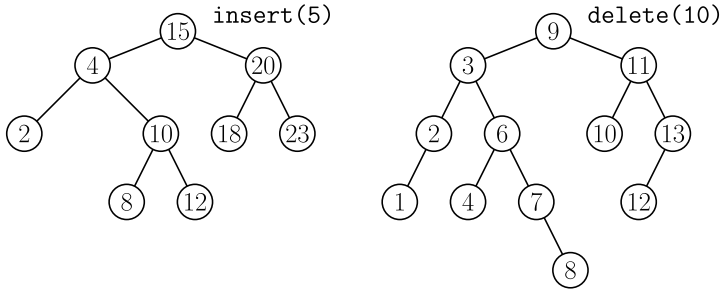


Figure 1: AVL-tree operations.

(a) (4 points) Show the result of executing the operation `insert(5)` to the tree of Fig. 1(a).

(b) (4 points) Show the result of executing the operation `delete(10)` to the tree of Fig. 1(b).

Problem 2. (8 points) Consider the AA trees shown in Fig. 2.

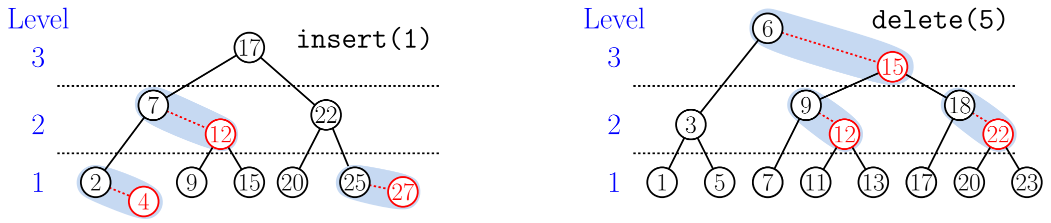


Figure 2: AA-tree operations.

(a) (4 points) Show the result of executing the operation `insert(1)` to the tree on the left.

(b) (4 points) Show the result of executing the operation `delete(5)` to the tree on the right.

In each case, show the final tree and list (in order) all the rebalancing operations (skew, split, and update-level) that result in changes to the tree (e.g., "`skew(13)`"). Intermediate results may be shown for the sake of partial credit.

Draw the tree as in Figs. 6 and 7 from Lecture 9. Indicate both the levels and distinguish red from black nodes. You do not need to color the nodes—a dashed line coming in from the parent indicates that a node is red. (Do not bother drawing `nil`.)

Problem 3. (6 points) Recall the minimal AVL trees from Lecture 7. Formally, we define T_0 and T_1 as shown in Fig. 3(a), and for any $h \geq 2$, T_h consists of a root with T_{h-1} as its left subtree and T_{h-2} as its right subtree (see Fig. 3(b)). Suppose we label the nodes in sequence $\langle 1, 2, 3, 4, 5, \ldots \rangle$ according to an inorder traversal (see Fig. 3(c)). Observe that the nodes along the leftmost chain of the tree will take on values from the Fibonacci sequence. Establish this by giving a (formal) proof for the following theorem.

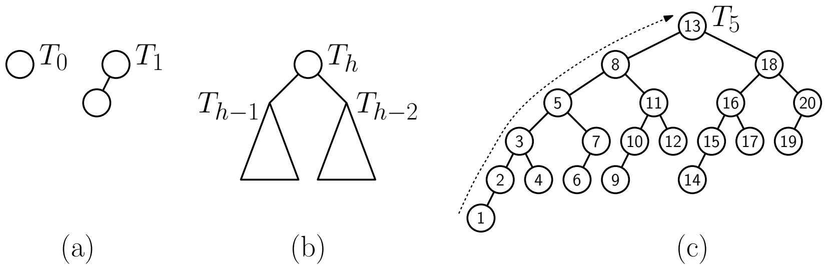


Figure 3: Inorder labeling of a Fibonacci-based trees.

Theorem: For any $h \geq 0$, if the nodes of T_h are labeled according to their position in an inorder traversal of the tree (starting with 1), then the labels along the leftmost chain of tree (from leaf to root) generate the Fibonacci sequence

$$\langle F(2), F(3), F(4), F(5), \ldots, F(h+2) \rangle,$$

where $F(h)$ denotes the hth Fibonacci number.

Hint: If it helps, you may assume the result proved in class about these trees, namely that T_h has $F(h+3) - 1$ nodes. (In the lecture notes, we called this $N(h)$.)

Problem 4. (12 points) Suppose that you are implementing the 2-3 tree as described in class, and you have the node structure shown below. Each node has an additional parent link. As in the lecture, we will "cheat" and allow nodes to have 4 children, but this is only temporarily allowed. Since they do not matter for this exercise, we will only store keys, no values. We also provide two constructors, one for 2-nodes and one for 3-nodes (see Fig. 4).

```
class Node {
    int nChild // number of children (1 through 4)
    Node parent // parent of this node (null if root)
    Node[] child // a 4-element array of child references
    Key[] key // a 3-element array of keys

    // constructors for 2-nodes and 3-nodes
    Node(Node par, Node u, Key x, Node v)
    Node(Node par, Node u, Key x, Node v, Key y, Node w)
}
```

(a) (3 points) Present pseudocode for a function `Node leftSib(Node p)`, which returns a reference to the left sibling of `p`. If `p` has no left sibling, it returns `null`. Your function

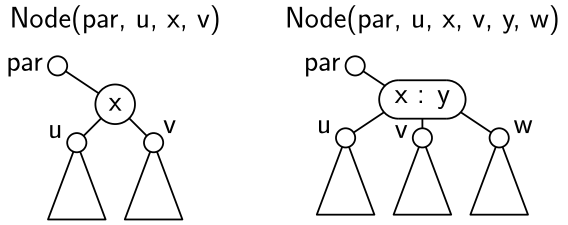


Figure 4: Constructors for 2-3 nodes.

should run in constant time. You may assume that all nodes other than p are valid 2-3 nodes (that is, they have 2 or 3 children.) `Hint:` Be sure to avoid dereferencing `null` pointers or indexing outside of array bounds.

(b) (3 points) Repeat (a) but now for `Node rightSib(Node p)`, which returns the right sibling.

(c) (6 points) Present pseudocode for a function `Node merge(Node p)`. It is given a non-root node p that contains only 1 child (and no keys). If it has a left sibling and this sibling is a 2-node, it merges its contents with this sibling node, creating a new node, which it returns. Otherwise, if it has a right sibling, and this sibling is a 2-node, it does the same with this sibling (see Fig. 5). Otherwise, it returns `null`. You may assume that all nodes other than p are valid 2-3 nodes (that is, they have 2 or 3 children.) **Do not worry about updating the parent node.** (We'll leave that for a future exercise.)

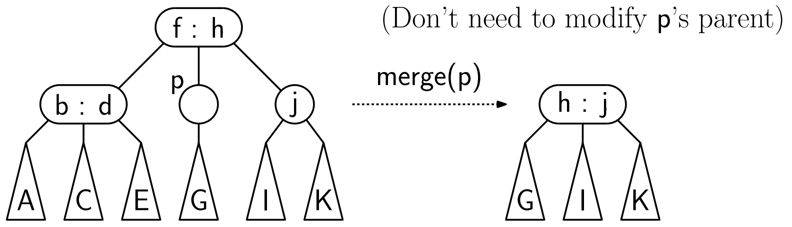


Figure 5: The function `merge(p)`.

Problem 5. (16 points) Alice and Bob want to test their implementation of a standard (unbalanced) binary search tree. They know that the tree will perform badly if keys are inserted in sorted order, and it will perform well (in expectation) if the keys are inserted in random order. They decide to analyze the performance of one more insertion order.

Let us assume that the number of keys n is chosen to be a perfect square, that is, $n = k^2$ for some $k \geq 1$. They first write the keys out row-by-row in a matrix. For example, here is what they would generate for $n = 16$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Then, they insert the keys by reading down the columns, working from right to left. In the above example, the insertion order is $\langle 4, 8, 12, 16, 3, 7, 11, 15, 2, 6, 10, 14, 1, 5, 9, 13 \rangle$.

Answer the following questions based on Alice and Bob's insertion process. **Throughout, you may assume that the number of nodes n is a strictly positive perfect square.**

(a) (3 points) Draw the final binary search tree that results for the above example involving $n = 16$ keys.

(b) (3 points) What is the height of the resulting tree? (Express your answer as a function of n.) **Hint:** To avoid being off by 1, recall the definition of height from the lecture notes. A tree with a single node has height 0, not 1.

(c) (4 points) Letting h denote the height of the final tree. Give a formula $d(i)$, which for $0 \leq i \leq h$ indicates the number of nodes at depth i in the resulting tree. Express your answer as a function of n and i. **Hint:** Recall the definition of depth from the lecture notes. The root is at depth 0, and depths increase from there.

(d) (6 points) Under the reasonable assumption that the time needed to insert a node at depth i is $i + 1$, what is the total time needed to insert all n keys in the tree? Express your answer as a function of n in closed form (no summations or recurrences). For full credit, give the exact formula. For partial credit, give an asymptotically tight bound. `Hint:` It may be useful to recall the summation formulas from the CMSC420 Reference Guide. **Show how you derived your answer.**

Challenge Problem 1: Augment your answer to Problem 4(c) by presenting pseudocode that not only creates the new merged node, but (assuming that the result is not `null`) also updates the contents of the parent node. Note that the parent's degree decreases by one, and so it may become a 1-node.

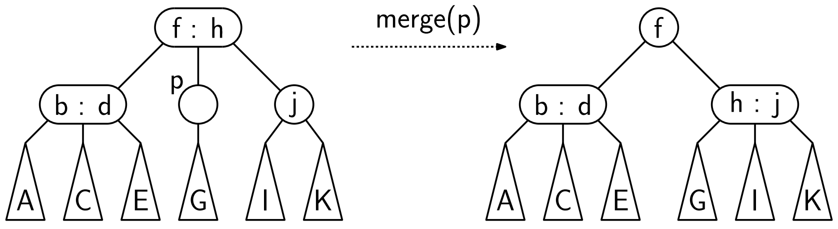


Figure 6: The complete function `merge(p)`.

Challenge Problem 2: In programming languages like C and C++, when a data structure is deallocated, its nodes must be explicitly deallocated one by one. Given a node `p`, the operation `delete(p)` deallocates the node, returning its memory block to the system. The simplest way to do this is to perform a postorder traversal of the tree, and delete nodes as you are backing out.

Performing a postorder traversal requires the use of system memory in the form of the recursion stack. If the tree is really huge, you may not have enough memory to store the recursion stack. Ironically, you may run out of memory in the process of trying to free up memory!

Describe a method for deleting all the nodes of a standard binary tree that uses only $O(1)$ additional working storage. (Note that using recursion, allocating arrays or other data structures, and allocating new nodes all contribute to your working storage.)

Your binary tree is absolutely minimal. Each node just has a left child link and a right child link, and nothing else (no keys, no values, no parent links.) Your algorithm should run in $O(n)$ time, where n is the number of nodes in the tree. **Hint:** You are allowed to modify the contents of the tree.

CMSC 420: Spring 2023

Practice Problems for Midterm 1

Problem 1. Short answer questions. Except where noted, explanations are not required, but may be given to help with partial credit. Whenever asked to express an answer as a function of n, you should assume that n is a large number.

(a) Recall that a binary tree is *full* if every node either has 0 or 2 children. Given a full binary tree with n total nodes, what is the maximum number of leaf nodes? What is the minimum number? Give your answer as a function of n (no explanation needed).

(b) You have an inorder-threaded binary tree with n nodes. Let u be an arbitrary non-leaf node in this tree. **True or False**: There must be at least one thread that points into u.

(c) You have a binary tree with inorder threads (for both inorder predecessor and inorder successor). Let u and v be two arbitrary nodes in this tree. **True or false**: There is a path from u to v, using some combination of child links and threads.

(d) You build a union-find data structure for a set of n objects. Initially, each element is in a set by itself. You then perform k union operations, where $k < n$. Each operation merges two different sets. Can the number of union-find trees be determined from k and n alone? If not, answer "It depends". If so, give the number as a function of k and n.

(e) You are given a binary heap containing n elements, which is stored in an array as `A[1...n]`. Given the index `i` of an element in this heap, present a formula that returns the index of its sibling. (Hint: You can either do this by manipulating the bits in the binary representation of `i` or by using a conditional (if-then-else).)

(f) In a leftist heap containing a large number of elements n, what is the minimum possible NPL value of the root? What is the maximum? (Express your answers as a function of n. It is okay to be off by an additive error of $\pm O(1)$.)

(g) Your boss asks you to program a new function for your leftist (min) heap. Given a leftist heap with a large number of elements n, the operation returns the *third smallest* item in the heap (without modifying its contents). What is the minimum number of heap entries that you might need to inspect to be certain that the third smallest item is among them?

(h) You have just performed an insertion into a 2-3 tree of height h. What is the maximum number of split operations that might be needed as a result? (Express your answer as a function of h.)

(i) You are given a 2-3 tree of height h, which has been converted into an AA-tree. As a function of h, what is the minimum number of red nodes that might appear on any path from a root to a leaf node in the AA tree? What is the maximum number? Explain.

(j) You are given a sorted set of n keys $x_1 < x_2 < \cdots < x_n$ (for some large number n). You insert them all into an AA tree in some arbitrary order. No matter what insertion order to choose, one of these keys *cannot* possibly be a red node. Which is it? Explain.

Problem 2. Suppose that we are given a set of n objects (initially each item in its own set) and we perform a sequence of m unions and finds (using height balanced union and path compression). Further suppose that all the unions occur *before* any of the finds. Prove that after initialization, the resulting sequence will take $O(m)$ time (rather than the $O(m\alpha(m, n))$ time given by the worst-case analysis).

Problem 3. You are given a degenerate binary search tree with n nodes in a left chain as shown on the left of Fig. 1, where $n = 2^k - 1$ for some $k \geq 1$.

(a) Derive an algorithm that, using only single left- and right-rotations, converts this tree into a perfectly balanced complete binary tree (right side of Fig. 1).

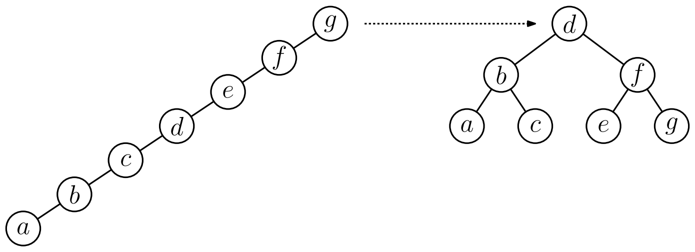


Figure 1: Rotating into balanced form.

(b) As an asymptotic function of n, how many rotations are needed to achieve this? $O(\log n)$? $O(n)$? $O(n \log n)$? $O(n^2)$? Briefly justify your answer.

Problem 4. You are given a binary tree (not necessarily a search tree) where, in addition to `p.left` and `p.right`, each node `p` has a *parent link*, `p.parent`. This points to `p`'s parent, and is `null` if `p` is the root. Given such a tree, present pseudocode for a function that returns the inorder successor of any node `p`. If `p` has no inorder successor, the function returns `null`.

```
Node inorderSuccessor(Node p) {
    // ... fill this in
}
```

Briefly explain how your function works. Your function should run in time proportional to the *height* of the tree.

Problem 5. You are given a standard (unbalanced) binary search tree. Let `root` denote its root node. Present pseudocode for a function `atDepth(int d)`, which is given an integer $d \geq 0$, and outputs the keys for the nodes that are at depth d in the tree (see Fig. 2). The keys should be output in increasing order of key value.

If there are no nodes at depth d, the function returns an empty list. The running time of your algorithm should be proportional to the number of nodes at depths $\leq d$. (For example, in the case of `atDepth(2)`, there are 7 nodes of equal or lesser depth.)

Hint: Create a recursive helper function. Explain what the initial call is to this function.

Problem 6. Assume that you are given a tree-based heap structure, which is represented by a binary tree (not necessarily complete nor leftist). Each node `u` stores three things, its priority,

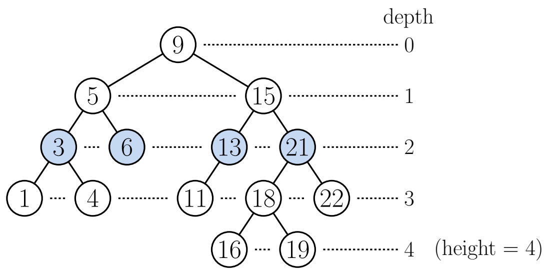


Figure 2: Nodes at some depth.

`u.key`, and the pointers to its subtrees, `u.left` and `u.right`. The keys are min-heap ordered (that is, a node's key is never smaller than its parent). There are no NPL values.

(a) Present pseudocode for a function `swapRight(Node u)` which is given a pointer to the root of a tree. It traverses the right chain of this tree and swaps the left and right subtrees of all nodes along this chain (see Fig. 3). It returns a pointer to the resulting tree. For full credit, your function should run in time proportional to the length of the right chain in the tree.

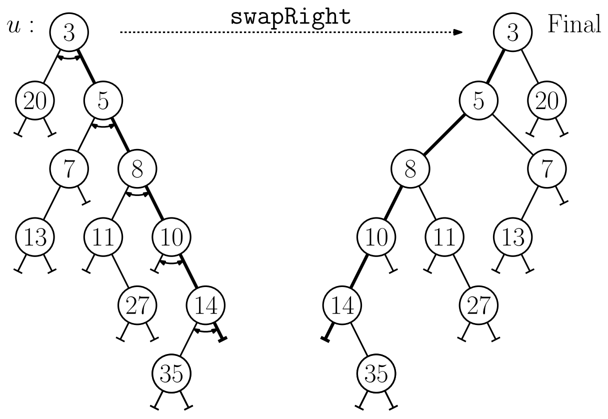


Figure 3: The function `swapRight`.

(b) Present pseudocode for a function `swapMerge(Node u, Node v)`, which is given pointers to the roots of two trees. It merges the right chains of these two trees according to min-heap order, and then performs `swapRight` on the resulting tree (see Fig. 4). It returns a pointer to the resulting tree.

Problem 7. Given any AVL tree T and an integer $d \geq 0$, we say that T is *full at depth d* if it has the maximum possible number of nodes (namely, 2^d) at depth d.

Prove that for any $h \geq 0$, an AVL tree of height h is full at all depths from 0 up to $\lfloor h/2 \rfloor$. (For example, the AVL tree of Fig. 2 has height 4, and is full at levels 0, 1, and 2, but it is not full at levels 3 and 4.)

Hint: Prove this by induction on the height of the tree.

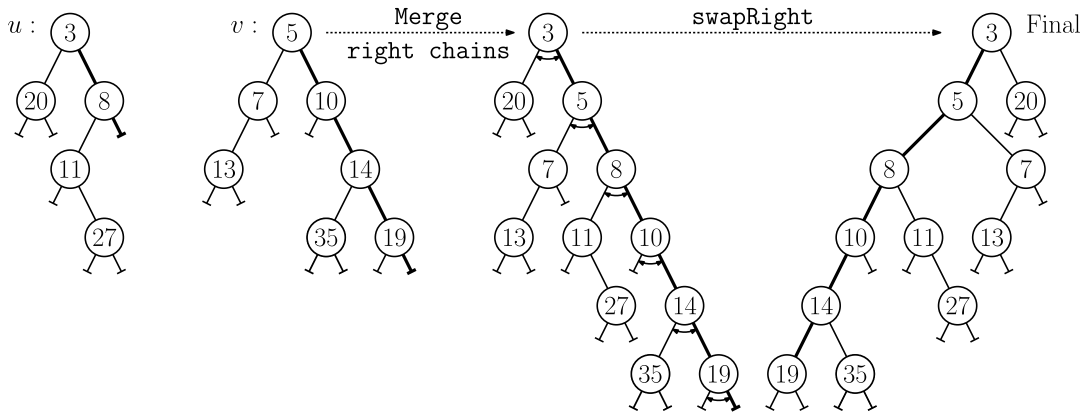


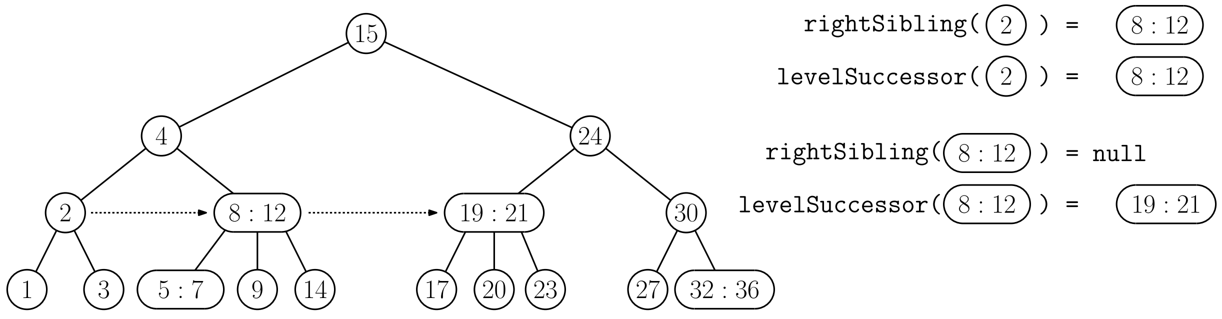
Figure 4: The function `swapMerge`.

Problem 8. Consider the following possible node structure for 2-3 trees, where in addition to the keys and children, we add a link to the parent node. The root's parent link is `null`.

```
class Node23 {                  // a node in a 2-3 tree
    int       nChildren         // number of children (2 or 3)
    Node23    child[3]          // our children (2 or 3)
    Key       key[2]            // our keys (1 or 2)
    Node23    parent            // our parent
}
```

Assuming this structure, answer each of the following questions:

(a) Present pseudocode for a function `Node23 rightSibling(Node23 p)`, which returns a reference to the sibling to the immediate right of node `p`, if it exists. If `p` is the rightmost child of its parent, or if `p` is the root, this function returns `null`. (For example, in Fig. 5, the right sibling of the node containing "2" is the node containing "8:12". Since the node containing "8:12" is the rightmost node of its parent ("4"), it has no right sibling.) Your function should run in $O(1)$ time.



rightSibling(2) = 8 : 12
levelSuccessor(2) = 8 : 12
rightSibling(8 : 12) = null
levelSuccessor(8 : 12) = 19 : 21

Figure 5: Sibling and level successor in a 2-3 tree.

(b) For a node p in a 2-3 tree, its *level successor* is the node to its immediate right at the same level. Give pseudocode for a function `Node23 levelSuccessor(Node23 p)`, which returns a reference to p's level successor, if it exists. If p is the rightmost node on its level (including the case where p is the root), this function returns `null`. (For example, in Fig. 5, the level successor of the node containing "2" is the node containing "8:12", and the level successor of "8:12" is the node containing "19:21".)

Your function should run in $O(\log n)$ time. If you like, you may use `rightSibling`.

(c) Suppose we start at any node p in a 2-3 tree with n nodes, and we repeatedly perform `p = levelSuccessor(p)` until `p == null`. What is the (worst-case) total time needed to perform all these operations? (Briefly justify your answer.)

Problem 9. Each node of a 2-3 tree may have either 2 or 3 children, and these nodes may appear anywhere within the tree. Let's imagine a much more rigid structure, where the node types alternate between levels. The root is a 2-node, its two children are both 3-nodes, their children are again 2-nodes, and so on (see Fig. 6). Generally, depth i of the tree consists entirely of 2-nodes when i is even and 3-nodes when i is odd. (Remember that the *depth* of a node is the number of edges on the path to the root, so the root is at depth 0.) We call this an *alternating 2-3 tree*. While such a structure is too rigid to be useful as a practical data structure, its properties are easy to analyze.

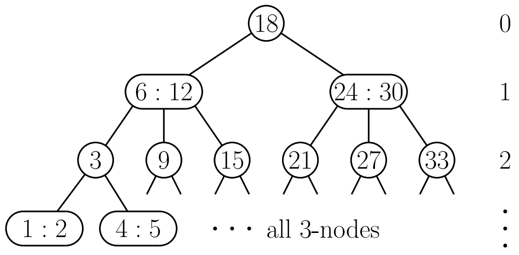


Figure 6: Alternating 2-3 tree.

(a) For $i \geq 0$, define $n(i)$ to be the number of nodes at depth i in an alternating 2-3 tree. Derive a closed-form mathematical formula (exact, not asymptotic) for $n(i)$. Present your formula and briefly explain how you derived it.

By "closed-form" we mean that your answer should just be an expression involving standard mathematical operations. It is *not* allowed to involve summations or recurrences, but it is allowed to include cases, however, such as

$$n(i) \;=\; \begin{cases} \dots & \text{if } i \text{ is even} \\ \dots & \text{if } i \text{ is odd.} \end{cases}$$

(b) For $i \geq 0$, define $k(i)$ to be the number of keys stored in the nodes at depth i in an alternating 2-3 tree. (Recall that each 2-node stores one key and each 3-node stores 2 key). Derive a closed-form mathematical formula for $k(i)$. Present your formula and briefly explain how you derived it. (The same rules apply for "closed form", and further your formula should stand on its own and not make reference to $n(i)$ from part (a).)

Problem 10. In this problem, we will consider variations on the amortized analysis of the dynamic stack. Let us assume that the array storage only *expands*, it never contracts. As usual, if the current array is of size m and the stack has fewer than m elements, a `push` costs 1 unit. When the mth element is pushed, an overflow occurs.

You are given two constants $\gamma, \delta > 1$. When an overflow occurs, we allocate a new array of size γm, copy the elements from the old array over to the new array. The total cost is 1 (for the push) plus δm (for copying). Derive a tight bound on the amortized cost, which holds in the limit as $m \to \infty$. Express your answer as a function of γ and δ. Explain your answer.

CMSC 420 (0201) - Midterm Exam 1

Problem 1. (10 points)

(a) (5 points) Show the final tree after performing `insert(25)` into the AVL tree shown below. Show both the **final tree** and the **balance factors** for all the nodes. (Intermediate results can be given for partial credit.)

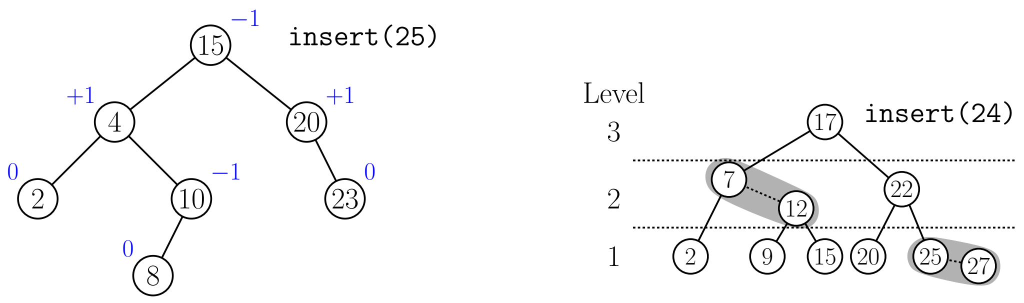


Figure 1: AVL and AA tree operations.

(b) (5 points) Consider the AA tree in Fig. 1 (right). (Note that nodes 12 and 27 are both red.) Show the final tree after performing `insert(24)` on this tree. Indicate the **levels** as well. (Intermediate results can be given for partial credit.)

Problem 2. (35 points) Short answer questions. Explanations are not required, but may always be given to help with partial credit.

(a) (4 points) You have a binary tree nodes with inorder threads. There are n nodes. How many threads are there? (If the answer cannot be determined from n alone, answer "It depends").

(b) (3 points) You have a union-find data structure for a set of n objects. After initialization, you perform k union operations, where $1 \leq k \leq n-1$. As a function of n and/or k, what is the maximum possible height of any tree in the data structure?

(c) (4 points) We showed that in any leftist heap containing n nodes the NPL (null path length) of the root is $O(\log n)$. What is the best that can be said about the NPL value of the root node in *any* binary tree having n nodes? (Select one.)

(1) $O(\log n)$
(2) $O(\log^2 n)$
(3) $O(\sqrt{n})$
(4) $O(n)$

(d) (4 points) You perform the operation `update-key` in a leftist heap containing n entries. Assuming you do this by sifting the entry up or down the tree, what is the worst-case time for this operation? (Select one)

(1) $O(1)$ time
(2) $O(\log n)$ time
(3) $O(\sqrt{n})$ time
(4) $O(n)$ time
(5) $O(n \log n)$ time

(e) (4 points) You perform a postorder traversal of a binary tree with $n \geq 1$ nodes. True or false: The first node in the traversal must be a leaf.

(f) (6 points) You want to maintain an ordered dictionary using a simple linear list. List (very briefly) one advantage and one disadvantage of an array-based approach versus a linked-list approach.

(g) (6 points) You have an AA tree whose root resides at level 13. What is the minimum and maximum number of red nodes that can be encountered along any path from the root to the leaf level (that is, level 1)?

(h) (4 points) You have a data structure in which the ith operation takes time $O(i)$. Given a string of n operations on this data structure, what is the best that can be said about the *amortized time* of each operation? **Hint:** $\sum_{i=1}^{m} i = m(m+1)/2$.

(1) $O(1)$
(2) $O(\sqrt{n})$
(3) $O(n)$
(4) $O(n^{3/2})$
(5) $O(n^2)$

Problem 3. (15 points) Throughout this problem assume that you have a multi-way tree represented in the first-child/next-sibling representation. Recall the node structure.

```
class Node {              // a node of a multi-way tree
  Key data                // data
  Node firstChild         // leftmost child (null if leaf)
  Node nextSibling        // next sibling to right (null if rightmost child)
}
```

Given any node `p` of the tree, define its *leftmost leaf* to be the leftmost leaf in the subtree rooted at `p`. If `p` is a leaf, then it is its own leftmost leaf. Otherwise, it is the leftmost leaf of its leftmost child. The *rightmost leaf* of `p` is defined symmetrically (see Fig. 2).

(a) (5 points) Present pseudocode for a function `Node leftLeaf(Node p)` that returns `p`'s leftmost leaf. For full credit, express your answer using just recursion (no loops). (You may create additional helper functions.) For half credit, you may use loops.

(b) (10 points) Same as (a), but for the rightmost leaf of `p`, `Node rightLeaf(Node p)`. Again, for full credit use recursion only, no loops. Loops are okay for partial credit.

Problem 4. (25 points) Alice and Bob continue to test their study of the performance of standard (unbalanced) binary search trees. They test a new insertion pattern. First, they pick a positive integer k, they fill the entries of an upper triangular $k \times k$ matrix row-by-row with

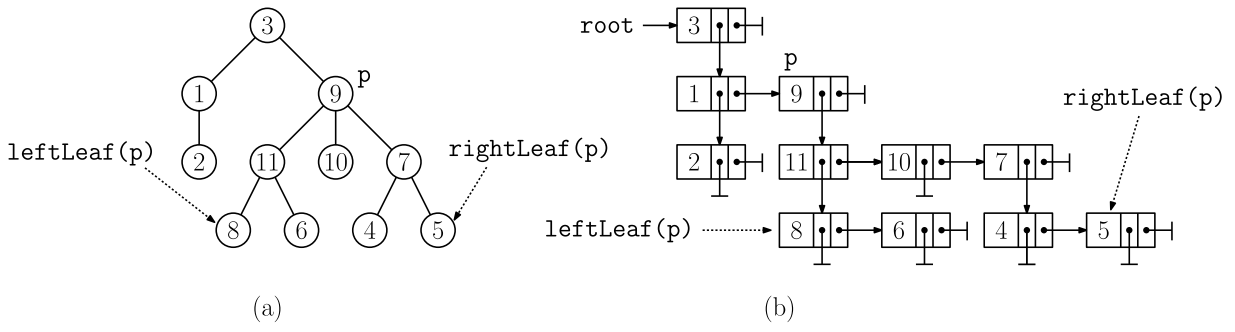


Figure 2: Leftmost and rightmost leaves in a multi-way tree.

numbers $\{1, 2, 3, \ldots\}$, and then they insert the keys into an unbalanced binary search tree by reading down the columns, working from right to left. An example for $k = 5$ is shown below.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ & 6 & 7 & 8 & 9 \\ & & 10 & 11 & 12 \\ & & & 13 & 14 \\ & & & & 15 \end{bmatrix}$$

They would insert $n = 15$ keys in the order $\langle 5, 9, 12, 14, 15, 4, 8, 11, 13, 3, 7, 10, 2, 6, 1 \rangle$.

Answer the following questions based on Alice and Bob's insertion process.

(a) (5 points) Given $k \geq 1$, what is the total number of keys n inserted? (Express your answer as a function of k.) **Hint:** For any $m \geq 0$, $\sum_{i=1}^{m} i = m(m + 1)/2$.

(b) (5 points) Draw the final binary search tree that results for the above example when $k = 5$. **Hint:** It has a very regular structure.

(c) (5 points) What is the height of the resulting tree? (Express your answer as a function of k and/or n.)

(d) (5 points) Letting h denote the height of the final tree. Give a formula $d(i)$, which for $0 \leq i \leq h$ is the number of nodes at depth i in the resulting tree. Express your answer as a function of k, n, and/or i.

(e) (5 points) Assuming that the time needed to insert a node at depth i is $i + 1$, what is the total time needed to insert all n keys in the tree? For full credit, express your answer in big-O notation as an tight asymptotic function of n in closed form—no summations or recurrences. **Hint:** For any $m \geq 0$, $\sum_{i=1}^{m} i^2 = (2m^3 + 3m^2 + m)/6$.

Problem 5. (15 points) In this problem we will analyze the amortized complexity of a new variant of the expanding stack. Rather than doubling, the size of the array m grows as a **perfect square**, that is, $m = k^2$ for some $k \geq 1$. We start with an array of size 1, and then subsequent overflows result in arrays of sizes 4, 9, 16, 25, and so on. Each operation costs +1 work unit, unless it causes the array to overflow. If so, we allocate a new larger array, copy the old contents into it. *The actual cost of the expansion is the number of elements copied.*

See Fig. 3 for an example. A run starts when we overflow an array of size $(k-1)^2$, resulting in an array of size $m = k^2$. When this array overflows, we allocate an array of size $(k+1)^2$ and copy all m elements to it.

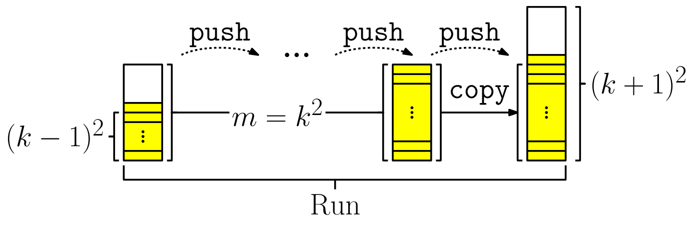


Figure 3: Perfect-square expanding array.

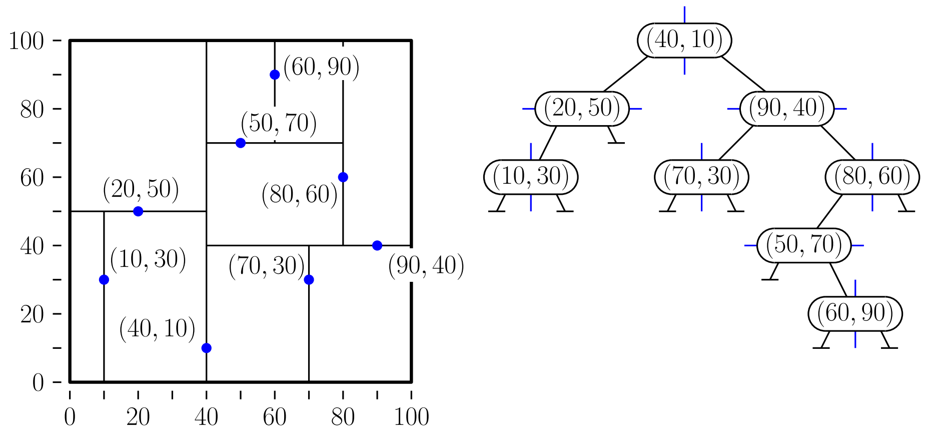
(a) (10 points) Consider a single run involving an array of size $m = k^2$ for some large k Fig. 3. Derive the worst-case amortized cost of this single run, including costs of the operations to fill the array and the expansion cost. Express your answer as a function of m and/or k. Since k is large, you may ignore $O(1)$ additive terms.

(b) (5 points) Suppose that we start with an empty stack and an array of size $m = 1$ and perform n operations, for some very large number n. What is the worst-case asymptotic amortized cost of this data structure as a function of n? Express your answer as a function of n using big-O notation. (A brief explanation suffices. We don't need a formal proof.)

CMSC 420: Spring 2023

Homework 3: kd-Trees, Splay Trees, and More

Problem 1. (10 points) Consider the kd-tree shown in the figure below. Assume a “standard” kd-tree where the cutting dimensions alternates between x and y with each level. (Intermediate results are not required, but may be given for partial credit.)

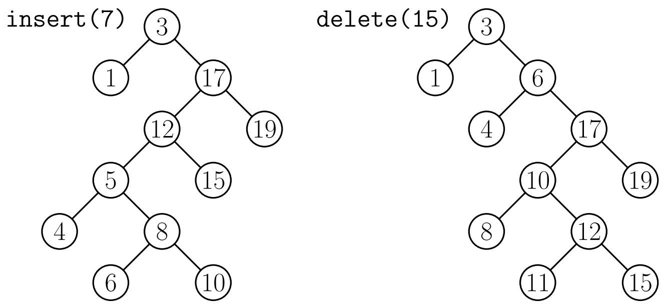


(a) (5 points) Show the final tree after the operation `insert((80,20))`. You need only show the tree, not the spatial subdivision.

(b) (5 points) Starting with the *original tree*, show the final tree after `delete((40,10))`. Indicate which nodes were used as replacement nodes.

Problem 2. (10 points) Consider the splay trees shown in the figure below. In both cases, apply the exact algorithms described in the lecture notes (Lecture 13).

(a) (5 points) Show the steps involved in operation `insert(7)` for the tree on the left.



(b) (5 points) Show the steps involved in operation `delete(15)` for the tree on the right.

In both cases, in addition to the final tree, show the result after each splay. Indicate which node was splayed on and the tree that resulted after splaying. (Intermediate results may be shown for partial credit.)

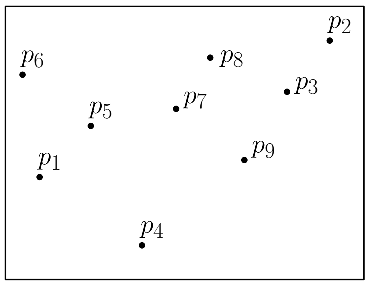
Problem 3. (10 points) Suppose you are given a kd-tree storing a set of points in $\mathbb{R}^2$. Recall from Lecture 10 that the `delete` operation on standard kd-trees makes use of the helper function, `findMin(Node p, int i)`, which returns the point in node `p`'s subtree that has the smallest ith coordinate ($i = 0$ for x and $i = 1$ for y). Let us assume that your kd-tree satisfies the "standard assumptions", namely that the cutting dimensions alternate between x and y with each level of the tree, and for each internal node `p`, there are an equal number of points in its left and right subtrees.

Under these assumptions, prove that for any node `p`, the running time for `findMin(Node p, int i)` is $O(\sqrt{m})$, where m denotes the number of points in `p`'s subtree.

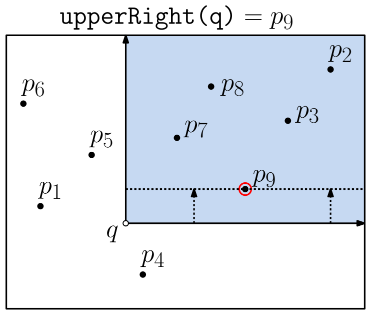
Hints: Your analysis is asymptotic, meaning that you should focus on the case when m is very large. You may use fact that, given any positive constants a, b, and c, the recurrence $T(n) = aT(n/b) + c$ solves to $O(n^{\log_b a})$, for all sufficiently large n.

Problem 4. (10 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ stored in a "standard" point kd-tree, as given in Lecture 10.

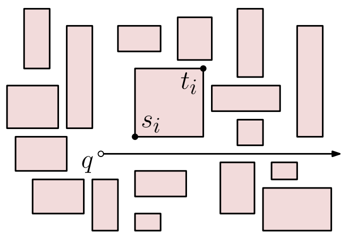
You are given a query point $q = (q_x, q_y)$, and the objective of an *upper-right query* is to compute the the bottommost point (that is, the point having the minimum y-coordinate) among all the points of P whose x-coordinates are greater than or equal to q_x and whose y-coordinates are greater than or equal to q_y (see the figure below). If there is no such point, the query returns `null`. To avoid messy edge cases, you may assume that there are no duplicate x- or y-coordinates among the points (including q).



(a)



(b)



(c)

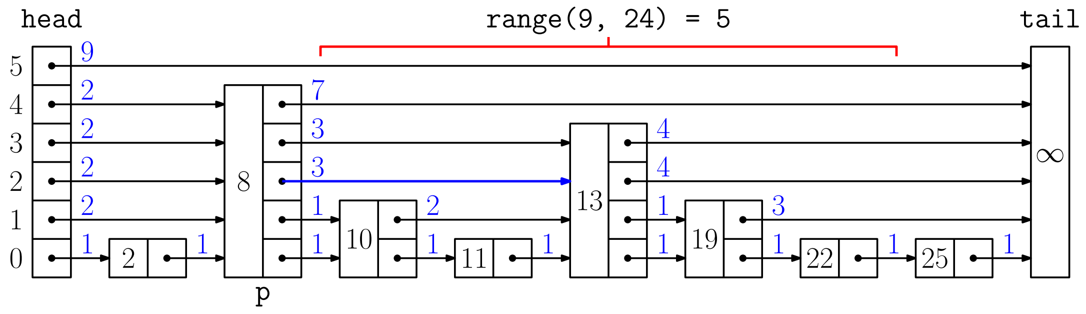
Present pseudo-code for an algorithm, `Point upperRight(Point q)` for answering upper-right queries given a kd-tree storing the points of P.

For full credit, your query algorithm should run in $O(\sqrt{n})$ time, but you do not need to prove this.

Hint: As usual, create a recursive helper function and explain how it is initially invoked. This is similar to nearest-neighbor searching in the sense that you will keep track of each node's `cell` and the `best` point seen so far. For efficiency, you should visit subtrees in an efficient order and avoid visiting nodes that cannot possibly contribute to the final answer.

Problem 5. (10 points) In this problem we consider an enhanced version of a skip list. As usual, each node `p` stores a key, `p.key`, and an array of next pointers, `p.next[]`. To this we add an array `p.span[]` which is parallel to `p.next[]`. This array is defined as follows. If `p.next[i]` refers to a node `q`, then `p.span[i]` contains the number of nodes from `p` to `q` (at level 0) of the skip list.

For example, in the figure below, the node `p` (with key "8") the link `p.next[2]` (shown in blue) jumps 3 nodes forward to the node with key 13, and so `p.span[2] = 3`.



Assuming this enhanced structure, present pseudo-code for a function `int range(Key lo, Key hi)`, which returns a count of the number of nodes in the entire skip list whose key values are greater than or equal to `lo` and less than or equal to `hi`. For example, in the figure above, the operation `range(9,24)` would return 5, since there are five items in this interval (namely, 10, 11, 13, 19, and 22).

Assuming that the skip list contains a total of n keys, your procedure should run in time expected-case time $O(\log n)$ (over all random choices), irrespective of the number of elements that lie within the range. Briefly explain how your function works.

Hint: It may help to approach this by first solving the simpler problem to answering semi-infinite range counting queries, namely, counting all the points whose key is $\leq x$.

Challenge Problem: You are given a set $R = \{r_1, \ldots, r_n\}$ of n disjoint rectangles in the $\mathbb{R}^2$. Each rectangle r_i is represented by a pair of points s_i is the lower left corner and t_i is the upper right corner. You may store this information in any data structure you choose with the purpose of answering the following horizontal ray-shooting queries efficiently.

You are given a query point $q = (q_x, q_y)$ which is guaranteed to lie outside of all of these rectangles. You shoot an infinite ray horizontally to the right of q. Your objective is to determine whether this ray hits any of the rectangles. To avoid messy edge cases, you may assume that there are no duplicate x- or y-coordinates among the points (including q).

Describe your data structure (what is stored? how is it organized?), and explain how to answer these horizontal ray-shooting queries from it. For full credit, your data structure should use $O(n)$ storage, and queries should be answered in $O(\sqrt{n})$ time. You may express your answer either in plain English or using pseudo-code (your choice), as long as it is clear and unambiguous how your algorithm can be implemented. Justify the correctness of your solution.

Hint: It is possible to reduce this problem to the kd-tree data structure from Problem 4, but you will need to augment the data structure with additional information.

CMSC 420: Spring 2023

Practice Problems for Midterm 2

Problem 1. Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(a) Given any binary tree T, define its *reversal* to be the tree that results by flipping the left and right children at every node in the tree. A class of trees is said to be *symmetrical* if (ignoring keys) its structural properties are invariant under reversals. That is, given any valid tree T in the class, its reversal is also a valid member of the class. Which of the following classes of binary trees are symmetrical? (Select all that apply.)

(1) Leftist heaps	(3) Red-black trees	(5) Scapegoat trees
(2) AVL trees	(4) AA trees	(6) Splay trees

(b) Suppose you know that a very small fraction of the keys in an ordered dictionary data structure are to be accessed most of the time, but you do not know which these keys are. Among the various data structures we have seen this semester, which would be best for this situation? Explain briefly.

(c) What is the maximum number of points that can be stored in a 3-dimensional point quadtree of height h? Express your answer as an exact (not asymptotic) function of h. (**Hint:** It may be useful to recall the formula for any $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1})-1)/(c-1)$.)

(d) We have n uniformly distributed points in the unit square, with no duplicate x- or y-coordinates. Suppose we insert these points into a kd-tree in *random* order, where the cutting dimension alternates between x and y. As a function of n what is the expected height of the tree? (You may express your answer in asymptotic form.)

(e) Same as the previous problem, but suppose that we insert points in *ascending* order of x-coordinates, but the y-coordinates are *random*.

(f) Both scapegoat trees and splay trees provide O(log n) amortized time for standard dictionary operations (insert, delete, and find). Suppose that your application involves many more find operations than insertions or deletions. Which of these two structures would you prefer and why?

(g) Unbalanced search trees and skip lists both support dictionary operations in $O(\log n)$ "expected time." What difference is there (if either) in the meaning of "expected time". Which would be preferred from the perspective of performance?

(h) You are given a skip list storing n items. What is the expected number of nodes that are at levels 3 and higher in the skip list? (Express your answer as a function of n. Assume that level 0 is the lowest level, containing all n items. Also assume that the coin is fair, return heads half the time and tails half the time.)

Problem 2. Recall that in an extended binary search tree, the keys are stored only in the external nodes, and each internal node stores a splitter key. Keys smaller than the splitter are stored in the left subtree and keys greater than or equal to the splitter are stored in the right subtree.

We say that an extended binary search tree is *geometrically-balanced* if the splitter value stored in each internal node p is midway between the smallest and largest keys of its external nodes. More formally, if the smallest external node in the subtree rooted at p has the value $x_{\min}$ and the largest external node has the value $x_{\max}$, then p's splitter is $(x_{\min} + x_{\max})/2$ (see Fig. 1).

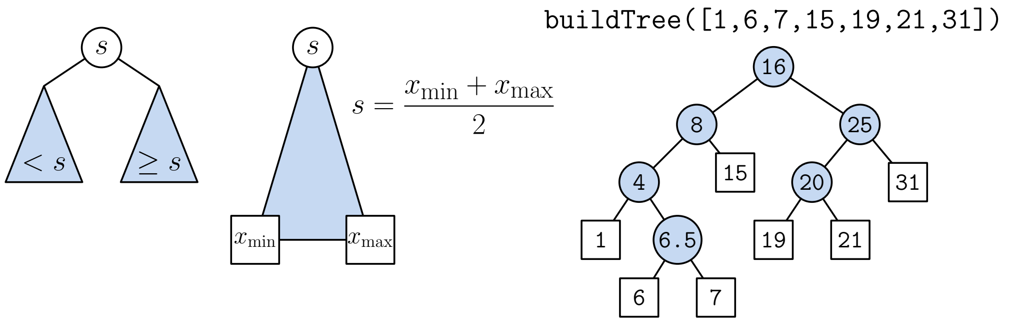


Figure 1: Geometrically balanced tree

Given a sorted array $A[0 \ldots n-1]$ containing $n \geq 1$ numeric keys, present pseudo-code for a function that builds a geometrically-balanced extended binary search tree, whose external nodes are the elements of A. (You may assume that you have access to a function for extracting a sublist of an array, but explain them.)

Problem 3. We usually like our trees to be balanced. Here we will consider *unbalanced* trees. Given a node p, recall that `size(p)` is the number of nodes in p's subtree. A binary tree is *left-heavy* if for each node p, where `size(p)` ≥ 3, we have

$$\frac{\texttt{size(p.left)}}{\texttt{size(p)}} \geq \frac{2}{3}$$

(see the figure below). Let T be a left-heavy tree that contains n nodes.

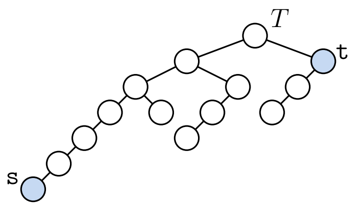


Figure 2: A left-heavy tree.

(a) Consider any left-heavy tree T with n nodes, and let s be the leftmost node in the tree (the first node in an inorder traversal). Prove that `depth(s)` $\geq (\log_{3/2} n) - c$, for some constant c. (The term c is just a small correction term. Let's assume that n is very large, so c may be ignored.)

(b) Consider any left-heavy tree T with n nodes, and let t be the rightmost node in the tree (the last node in an inorder traversal). Prove that `depth(t)` $\leq \log_3 n$.

Problem 4. In ordered dictionaries, a *finger search* is one where the search starts at some node of the structure, rather than the root. In this problem, we will consider how to perform finger searches in a skip list.

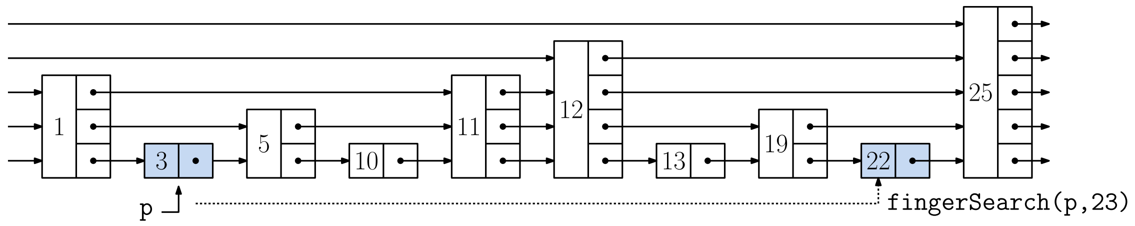


Figure 3: The operation `fingerSearch(p,23)`.

You are given a pointer `p` to a node in a skip list. You are also given a key `y`, where `p.key <= y`. Present pseudocode for a function `fingerSearch(p, y)` that performs a find operation on `y`, but rather than starting at the head node of the skip list, it starts at `p`. It returns a pointer to the node `q` that contains the largest key that is less than or equal to `y` (see Fig. 3).

Of course, we could crawl along level 0, but this is too slow. Suppose that there are m nodes between `p` and `q` in the skip list. We want the expected search time to be $O(\log m)$, not $O(m)$ and not $O(\log n)$.

Present pseudo-code for an algorithm for an efficient function. You do not need to analyze the running time.

Problem 5. You are given a set P of n points in the real plane stored in a kd-tree, which satisfies the *standard assumptions*. A *partial-range max query* is given two x-coordinates `lo` and `hi`, and the problem is to find the point $p \in P$ that lies in the vertical strip bounded by `lo` and `hi` (that is, `lo` $\leq$ `p.x` $\leq$ `hi`) and has the maximum y-coordinate (see Fig. 4).

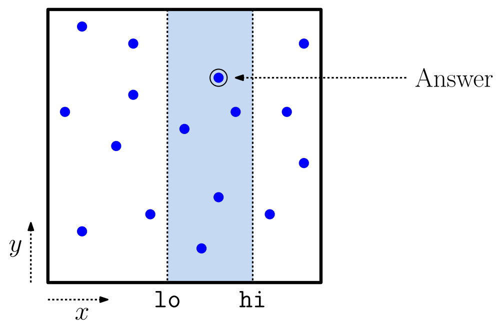


Figure 4: Partial-range max query.

(a) Present pseudo-code for an efficient algorithm to solve partial-range max queries, assuming that the points are stored in a point kd-tree. You may make use of any primitive operations on points and rectangles (but please explain them). **Hint:** A possible signature for your helper:

```
Point partialMax(double lo, double hi, KDNode p, Rectangle cell, Point best)
```

(b) Show that your algorithm runs in time $O(\sqrt{n})$.

Problem 6. In this problem we will see how to use kd-trees to answer a common geometric query, called *ray shooting*. You are given a collection of vertical line segments in 2D space, each starts at the x-axis and goes up to a point in the positive quadrant. Let $P = \{p_1, \ldots, p_n\}$ denote the upper endpoints of these segments (see Fig. 5). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.

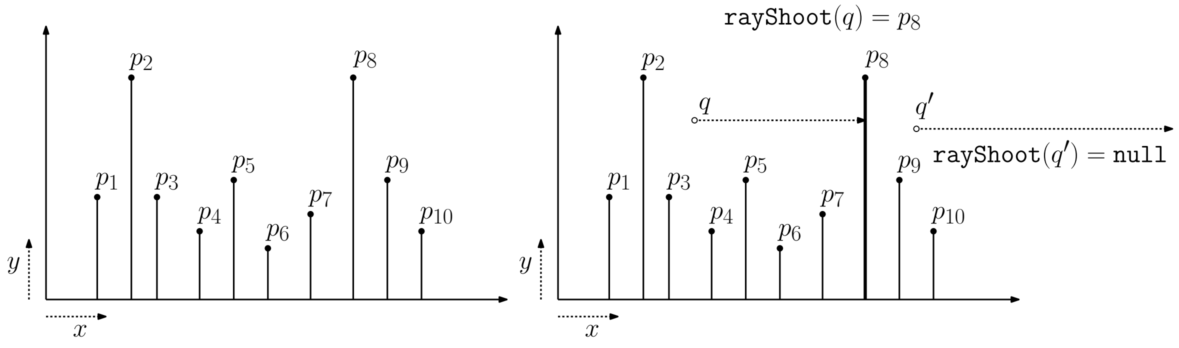


Figure 5: Ray shooting in a kd-tree.

Given a point q, we shoot a horizontal ray emanating from q to the right. This ray travels until it hits one of these segments (or perhaps misses them all). For example, in the figure above, the ray shot from q hits the segment with upper endpoint p_8. The ray shot from q' hits nothing.

In this problem we will show how to answer such queries using a standard point kd-tree for the point set P. A query is given the point $q = (q_x, q_y)$, and it returns the upper endpoint $p_i \in P$ of the segment the ray first hits, or `null` if the ray misses all the segments.

Suppose you are given a kd-tree of height $O(\log n)$ storing the points of P. (It does *not* store the segments, just the points.) Present pseudo-code for an efficient algorithm, `rayShoot(q)`, which returns an answer to the horizontal ray-shooting query (see the figure above, right).

You may assume the kd-tree structure given in class, where each node stores a point `p.point`, a cutting dimension `p.cutDim`, and left and right child pointers `p.left` and `p.right`, respectively. You may make use of any primitive operations on points and rectangles (but please explain them). You may assume that there are no duplicate coordinate values among the points of P or the query point.

Hint: You need only store points in your kd-tree. Here is a suggested helper.

```
Point rayShoot(Point q, KDNode p, Rectangle cell, Point best),
```

What is the initial call to the helper?

Problem 7. In class we showed that for a balanced kd-tree with n points in the real plane (that is, in 2-dimensional space), any *axis-parallel line* intersects at most $O(\sqrt{n})$ cells of the tree.

Show that for every n, there exists a set of points P in the real plane, a kd-tree of height $O(\log n)$ storing the points of P, and a line ℓ, such that *every* cell of the kd-tree intersects this line.

Problem 8. It is easy to see that, if you splay twice on the same key in a splay tree (`splay(x); splay(x)`), the tree's structure does not change as a result of the second call.

Is this true when we alternate between two keys? Let T_0 be an arbitrary splay tree, and let x and y be two keys that appear within T_0. Let:

- T_1 be the result of applying `splay(x); splay(y)` to T_0.
- T_2 be the result of applying `splay(x); splay(y); splay(x); splay(y)` to T_0.

Question: Irrespective of the initial tree T_0 and the choice of x and y, is $T_1 = T_2$? (That is, are the two trees structurally identical?) Either state this as a theorem and prove it or provide a counterexample, by giving the tree T_0 and two keys x and y for which this fails.

Problem 9. Throughout this problem, we start with a large, perfectly balanced binary search tree (see Fig. 6(a)). The only operations we perform are `delete` and `find`.

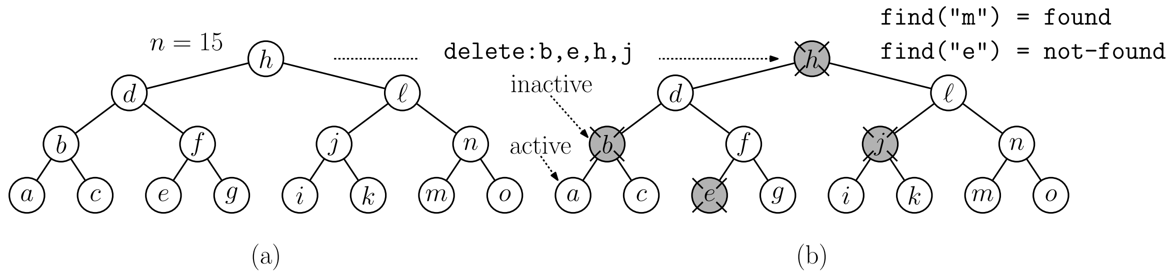


Figure 6: (a) A balanced binary search tree and (b) lazy deletion.

An alternative to the standard `delete` operation is called *lazy deletion*. We do not actually remove any nodes. Instead, to delete a node, we mark this node as *inactive*. When we perform a `find` operation, if the node found is *active*, we return `found`. But, if it is not found or *inactive*, we return `not-found` (see Fig. 6(b)).

As we perform deletions, the tree becomes burdened with many inactive nodes—not good. Whenever the number of inactive nodes exceeds the number active nodes, we *rebuild* the tree as follows. We first traverse the tree keeping only active nodes. We then build a perfectly balanced binary search tree containing only the active nodes (Fig. 7). Let n denote the total number of nodes in the tree (both active and inactive).

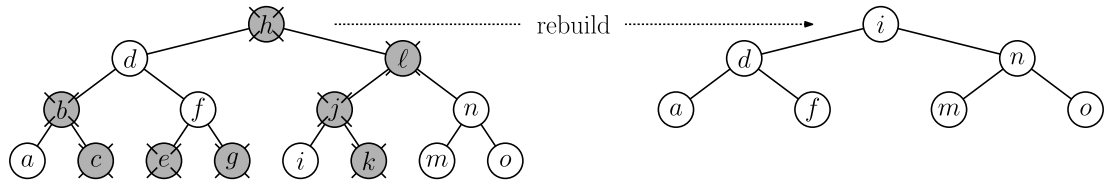


Figure 7: Rebuilding (with 8 inactive nodes and 7 active nodes).

We will analyze the time for `find` and `delete`. If rebuild does not occur, both `delete` and `find` take actual time $O(\log n)$ where n is the total number of nodes (both active and inactive). The actual time for rebuilding is $O(n)$.

(a) Show that the *worst-case time* for any `find` operation is $O(\log n_a)$, where n_a is the current number of *active nodes* in the tree. (Note that $n_a \leq n$, where n is the total number of nodes.)

(b) Derive the *amortized time* of lazy deletion. Express your answer asymptotically as a function of n (e.g., $O(1)$, $O(\log n)$ or $O(n)$.)

CMSC 420: Spring 2023

CMSC 420 (0201) - Midterm Exam 2

Problem 1. (10 points) Consider the kd-tree shown in Fig. 1. Assume a "standard" kd-tree where the cutting dimensions alternate between x and y with each level.

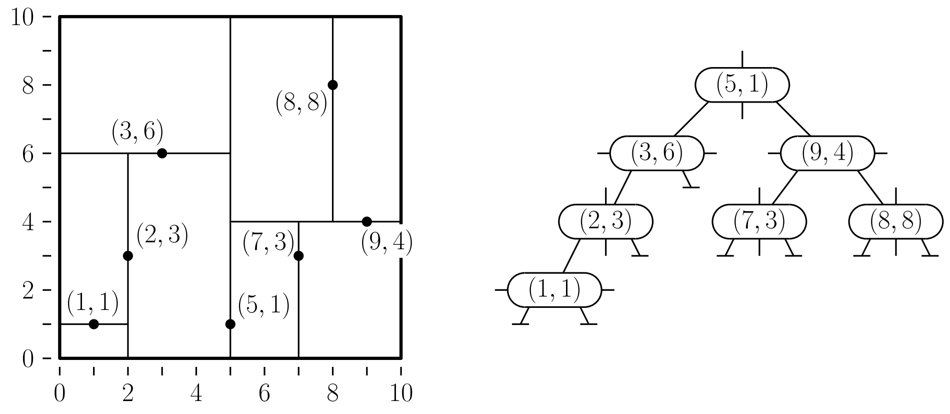


Figure 1: kd-Tree operations.

(a) (5 points) Show the final tree after the operation `insert((6,6))`. You need only show the tree, not the spatial subdivision.

(b) (5 points) Starting with the *original tree*, show the final tree after `delete((3,6))`. Indicate which nodes were used as replacement nodes. (Intermediate results are not required, but may be given for partial credit.)

Problem 2. (35 points) Short answer questions. No explanations required, but can be given for partial credit.

(a) (7 points) Consider a 2-dimensional point quadtree with m nodes. As an exact function of m, how many `null` pointers does it have? (Partial credit given depending on how close.)

(b) (7 points) You have a scapegoat tree, but you make two changes. First, a rebuild is triggered when an inserted node's depth exceeds $\log_{10/9} n$ (instead of $\log_{3/2} n$), and second a node `p` is declared a scapegoat if `size(p.child)/size(p)` $> 9/10$ (instead of $2/3$). Compared to the standard scapegoat tree, what changes? (Select all that apply.)

(1) The tree's height will tend to be *larger*

(2) The tree's height will tend to be *smaller*

(3) Subtrees will tend to be rebuilt *more often*

(4) Subtrees will tend to be rebuilt *less often*

(c) (3 points) What is the *maximum* number of subtrees that may need to be rebuilt as a result of a single insertion into a scapegoat tree? (Select the best option.)

(1) 1
(2) More than one, but a constant number
(3) $O(d)$, where d is the depth of the inserted node
(4) $O(h)$, where h is the overall height of the tree (even if the node is inserted at a much smaller depth)
(5) Larger than $O(h)$

(d) (7 points) You have a skip list with n nodes. Suppose that rather than using a fair coin to decide a node's height, you instead use a coin that comes up heads with probability 3/5 and tails with probability 2/5. All nodes start at level 0, and a node survives to the next higher level if the coin toss comes up heads. As a function of n, what is the expected number of nodes that survive to level 2 and higher?

(e) (3 points) You have a skip list containing n keys, where n is a large number. Suppose you perform a find operation. The search algorithm visits one or more nodes at each level of the structure. How many nodes do you *expect to visit* at level 4 of the search structure?

(1) None of them
(2) $O(1)$
(3) $O(\log n)$
(4) $O(n/(2^4))$
(5) All of them

(f) (5 points) Splay trees operate by performing most rotations in groups of two (zig-zag and zig-zig). Why is this necessary? In particular, why not just perform single rotations from the bottom up to bring the node up to the root?

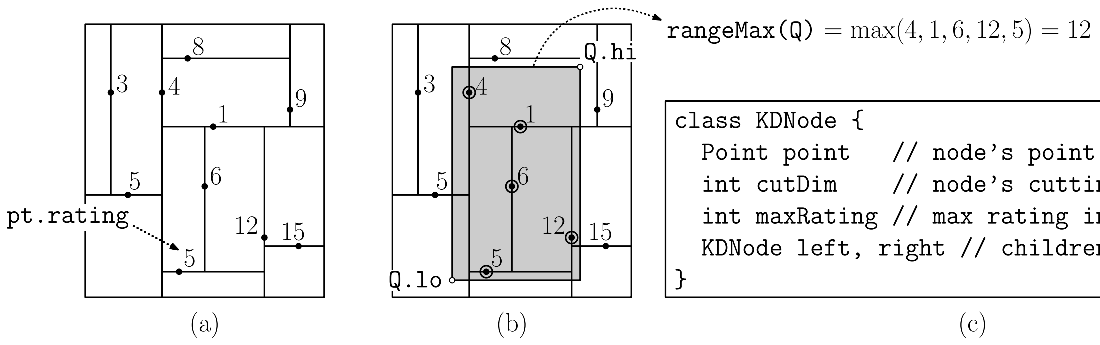
(g) (3 points) You have a splay tree with a large number n of keys, and you perform a long series of m `find` operations (where m is much larger than n). Suppose that one key is extremely popular, say `"Taylor Swift"`. Indeed, *every third* `find` is made to this popular key. What can you say about the time needed for the `find` operations on this popular key?

(1) They will have an amortized cost $O(1)$
(2) They will have an amortized cost of $O(\log n)$, but not $O(1)$
(3) They will have an amortized cost of $O((\log n)^3)$, but not $O(\log n)$
(4) They will have an amortized cost of $O(n)$, but not $O((\log n)^3)$
(5) They will have an amortized cost exceeding $O(n)$

Problem 3. (15 points) You are given a 2-dimensional point set stored in a standard kd-tree (cutting dimensions alternate). Let `root` denote its root, and let `rootCell` denote the root's rectangular cell.

Our objective is to answer queries of the form "What is the top rated restaurant in a given region?" To do this, in addition to its coordinates, each point pt $\in P$ stores a positive integer rating, `pt.rating` (see Fig. 2(a)). We are given a query rectangle Q, with lower-left and upper-right corner points, `Q.lo` and `Q.hi`, respectively. The function `rangeMax(Q)` returns the *maximum rating* among the points of P that lie within Q (see Fig. 2(b)). If there are no points in the range, it returns 0.

To help, each node `p` in the tree stores a field `maxRating`, which is the maximum rating over all the points in `p`'s subtree (see Fig. 2(c)).



```
class KDNode {
  Point point    // node's point
  int cutDim     // node's cutting dim
  int maxRating // max rating in subtree
  KDNode left, right // children
}
```

(c)

Figure 2: Range max queries.

(a) (10 points) Present pseudo-code for the kd-tree function `rangeMax(Rectangle Q)` that efficiently answers these queries. For full credit, it should run in $O(\sqrt{n})$ time.

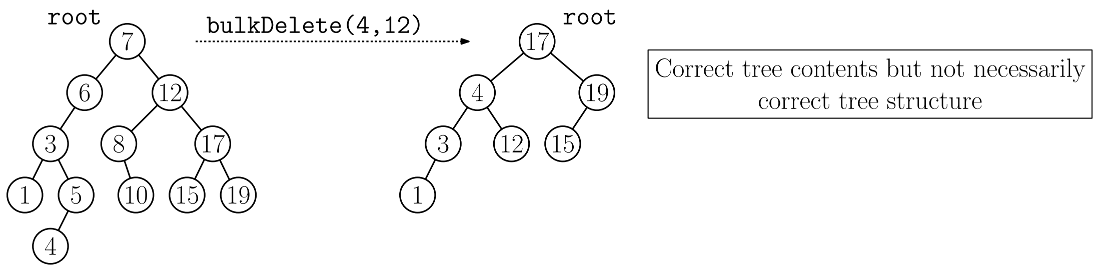
Hint: You may use whatever helper you like. Here is a suggestion.

```
int rangeMax(Rectangle Q, KDNode p, Rectangle cell)
```

You may assume that any geometric primitive involving a constant number objects (e.g., "is `Q` disjoint from `cell`") can be computed in constant time.

(b) (5 points) How do we update node `maxRating` values with each insertion? Edit/Modify the insertion helper for the kd-tree so that it both inserts a point `pt` with rating `pt.rating` and efficiently *updates* the `maxRating` values for the affected nodes of the tree.

Problem 4. (10 points) Suppose you are given a splay tree storing the keys $X = \{x_1, x_2, \ldots, x_n\}$. Design a new splay-tree operation called `bulkDelete(a, b)`. It is given two keys $a, b \in X$, where $a < b$, and it *deletes* all the keys between a and b, *exclusive*, that is, it deletes $\{x \in X : a < x < b\}$. (The elements a and b are *not* deleted.) For example, in Fig. 3(b), the operation `bulkDelete(4, 12)` deletes the keys $\{5, 6, 7, 8, 10\}$.



root
bulkDelete(4,12)
root
Correct tree contents but not necessarily correct tree structure

Figure 3: The `bulkDelete` operation in splay trees.

Present an efficient algorithm for this operation. As with other splay-tree operations, you are allowed to perform splay operations, either on the entire tree or on subtrees, and you can access and modify nodes. However, you are *not* allowed to iterate through the tree or apply recursive functions to the tree (other than calling `splay`).

You may present your algorithm either in pseudo-code or in English. You may assume that $a < b$, and both keys appear in tree. **Hint:** It is possible to do this with a *constant* number of splays, no matter how many entries are deleted.

Problem 5. (10 points) In this problem we consider an enhanced version of a skip list. As usual, each node `p` stores a key, `p.key`, and an array of next pointers, `p.next[]`. To this we add a parallel array `p.span[]`, where `p.span[i]` stores the number of nodes that `next[i]` skips.

Present pseudo-code for a function `Key getKth(int k)`, which returns the kth smallest key in the entire skip list. For example, in Fig. 4, the call `getKth(6)` would return 19, since 19 is the sixth smallest key. You may assume that $1 \leq k \leq n$, where n is the total number of nodes in the skip list.

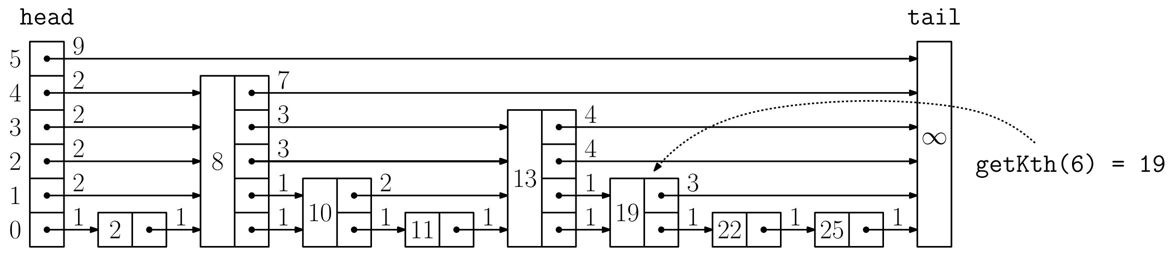


Figure 4: Skip list with span values.

Your procedure should run in time expected-case time $O(\log n)$ (over all random choices), but you don't need to prove this.

Problem 6. (20 points) Recall that an extended binary search tree consists of internal nodes, which have exactly two children, and external nodes, which have no children. A node's *weight* is defined to be the number of external nodes its subtree. (Internal nodes are not counted.) For example, in Fig. 5 each node is labeled with its weight.

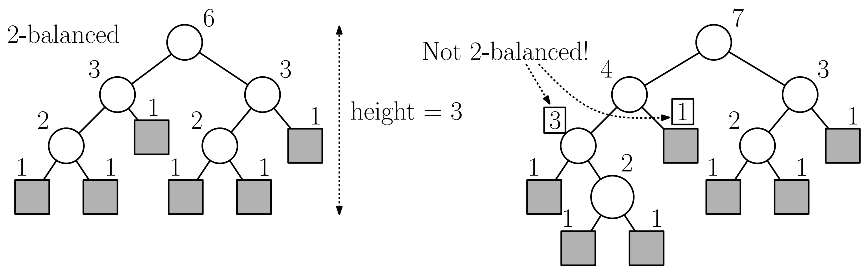


Figure 5: Weight-balanced extended trees.

Given $\alpha \geq 1$, an external tree is *α-balanced* if for every internal node `u`,

$$\frac{1}{\alpha} \leq \frac{\text{weight(u.left)}}{\text{weight(u.right)}} \leq \alpha.$$

In Fig. 5, the tree on the left is 2-balanced. But the tree on the right is not because there are two siblings with weight ration $3:1$, exceeding the allowed ratio of $2:1$.

(a) (10 points) Prove that if an extended binary tree of total weight $n \geq 1$ is 2-weight balanced, then its height is at most $\log_{3/2} n$.

(b) (5 points) Generalize the result from (a). Given an extended tree that is α-balanced for some $\alpha \geq 1$, its height is at most $\log_\beta n$ for some β that depends on α. What is the value of β as a function of α? (You do not not need to give the proof, just the formula).

(c) (5 points) **True or False**: Given a weight balanced tree with total weight n, if there is a node at depth greater than $\log_{3/2} n$, then some ancestor of this node is *not* 2-weight balanced. Give your answer and a brief (1–2 sentence) justification.

CMSC 420: Spring 2023

Homework 4: Hashing, B-Trees, and Tries

Problem 1. (15 points) In this problem, you will show the result of inserting a sequence of three keys into a hash table, using linear and quadratic probing and double hashing. For each operation (9 in total) show contents of the hash table after the insertion. If successful, indicate the number of *probes*, that is, the number of array elements accessed. (Note that the initial access counts as a probe, so if there is no collision, the number of probes is 1.) If the sequence fails, i.e., loops infinitely, indicate that the insertion *fails*.

See Figs. 3 and 4 from Lecture 15 for an example.

(a) (5 points) Show the results of inserting the keys "X" then "Y" then "Z" into the hash table shown in Fig. 1(a), assuming *linear probing*. (Insert the keys in *sequence*, so each inserted key remains for the later insertions.)

(a) Linear probing

```
insert("X")  h("X") = 2
insert("Y")  h("Y") = 13
insert("Z")  h("Z") = 11
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
▨	A	M	L	▨	H	▨	C	I	▨	▨	J	F	B

(b) Quadratic probing

```
insert("X")  h("X") = 3
insert("Y")  h("Y") = 12
insert("Z")  h("Z") = 4
```

0	1	2	3	4	5	6	7	8	9	10	11	12
F	W	▨	P	L	J	▨	Q	▨	▨	N	▨	▨

Figure 1: Hashing with linear and quadratic probing.

(b) (5 points) Repeat (a) using the hash table shown in Fig. 1(b) assuming *quadratic probing*.

(c) (5 points) Repeat (a) using the hash table shown in Fig. 2 assuming *double hashing*, where the second hash function g is shown in the figure.

(c) Double hashing

```
insert("X")  h("X") = 4; g("X") = 2
insert("Y")  h("Y") = 9; g("Y") = 3
insert("Z")  h("Z") = 3; g("Z") = 5
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E	▨	▨	▨	M	▨	K	▨	▨	F	▨	▨	B	Q	▨

Figure 2: Hashing with double hashing.

Problem 2. (12 points) Consider the B-trees of order 4 shown in Fig. 3 below. Let us assume two conventions. First, key rotation (when possible) has precedence over splitting/merging. Second, when splitting a node, use the convention given in the lecture, that the left node has $\lceil m/2 \rceil$ children and the right node has the remaining children.

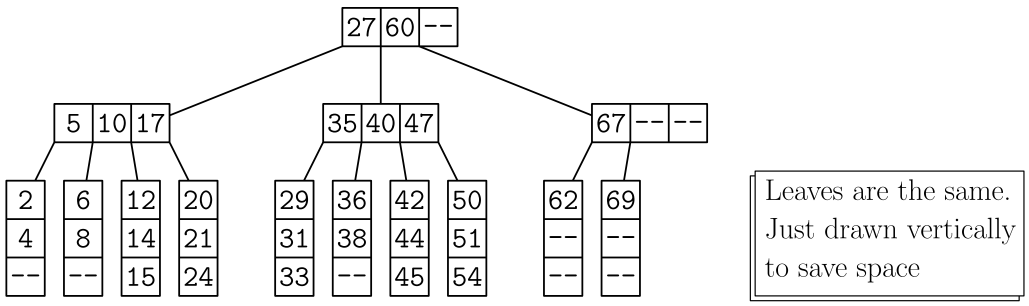


Figure 3: B-tree operations.

(a) (4 points) Show the final tree that results after inserting the key `23` into the tree of Fig. 3.

(b) (4 points) Show the final tree that results after inserting the key `55` into the (original) tree of Fig. 3.

(c) (4 points) Show the final tree that results after deleting the key `62` from the (original) tree of Fig. 3.

(Intermediate results are not required, but may be given to help assigning partial credit.)

Problem 3. (12 points) In this problem we will build a suffix tree for $S =$ `"abbabbaba$"`.

(a) (4 points) List the 10 substring identifiers of S as shown in Fig. 7 of Lecture 18.

(b) (8 points) Draw the suffix tree for S. Use the same form as shown in Fig. 7 of Lecture 18. When drawing your tree, order the children of each node from left to right in the order `a` < `b` < `$`. (**Hint:** Be sure to compress paths wherever possible. Note that there are online suffix tree generators, but we strongly encourage you to use these only to verify your final answer. They generally do not match the format we are looking for, and you won't be able to use them on the final exam.)

Problem 4. (11 points) In class, we discussed the de la Briandais representation of a trie, in which the trie is stored using the first-child/next-sibling tree representation (see Fig. 4). We assume that the strings stored in the tree are *prefix free*, meaning that no string is a prefix of any other.

Let us assume our tree is an extended tree. Each internal node `p` store a single character as the `p.key` and its first-child and next-sibling pointers, `p.firstChild` and `p.nextSibling`, respectively. External nodes store the final string as `p.key`. To distinguish between node types, each node stores a boolean, `p.isExternal`, which is `true` for external nodes and false otherwise. Finally, each node contains a value `p.weight` which indicates the number of external nodes in the subtree rooted at `p`. Let `root` denote the root of the tree.

(a) (5 points) Present pseudocode for an efficient function `int prefixCount(String pattern)`, which returns a count of the number of strings having `pattern` as a prefix. If no string has this prefix, return 0.

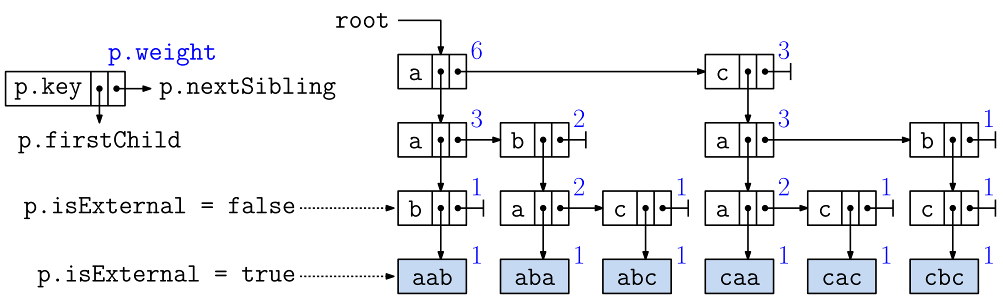


Figure 4: Pattern searching in a de la Briandais trie.

For example, in Fig. 4, the call `prefixCount("ca")` would return 2, for the strings `"caa"` and `"cac"`. The call `prefixCount("abacus")` would return 0, since no string has this as a prefix.

Your algorithm should run in time proportional to the product of the length of the pattern string and the alphabet size. (**Hint:** Your algorithm may be recursive or iterative. You may use any standard string processing utilities you like.

(b) (6 points) Present pseudocode for an efficient function `int wildCount(String pattern)`, which is given a pattern string containing *wildcard symbols*. If the pattern string contains the symbol `"*"`, this will match any single character at this position. In order to be counted, an entire string of the dictionary must match the entire pattern.

For example, in Fig. 4:

`wildCount("*a*")`: Matches all 3-character strings having an `"a"` as the middle character, and so would return 3 (for the strings `"aab"`, `"caa"`, and `"cac"`).

`wildCount("*bb")`: Matches all 3-character strings ending in `"bb"`, and so returns 0.

`wildCount("c")`: Matches the string `"c"`, and so would return 0.

`wildCount("c**")`: Matches all 3-character strings that start with `"c"`, and so would return 3 (for the strings `"caa"`, `"cac"`, and `"cbc"`).

Challenge Problem. You are given a very, very long linked list, where each node contains a single member, `next`, which points to the next element in the linked list. The variable `head` points to the head of the linked list. There are two possible forms that the list might take:

Open: The linked list eventually ends with a `null` pointer (see Fig. 5(a)).

Closed: The linked list loops back on itself, with one `next` pointer that points to an earlier node in the list (see Fig. 5(b)).

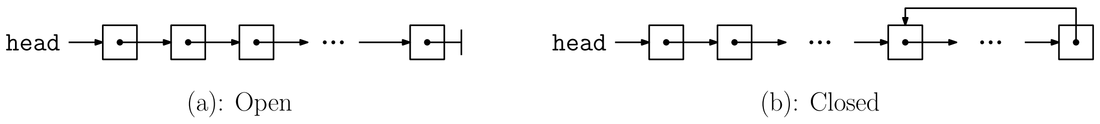


Figure 5: Open or closed linked list?

Describe an efficient function that determines whether the list is open or closed. Here is the catch:

- You do not know how many elements are in the list
- You cannot modify the nodes of the list, you do not know how many nodes there are in the list
- You cannot use more than a constant amount of additional working storage

This means that you cannot create an array, you cannot allocate any linked structures, and you cannot make recursive calls (since recursive calls implicitly use the system's recursion stack, which is an array).

Your function should run in $O(n)$ time, where n is the number of elements in the linked list.

CMSC 420: Spring 2023

Practice Problems for the Final Exam

Problem 1. Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(a) You have an inorder-threaded binary tree with n nodes. Let u be an arbitrary non-leaf node in this tree. **True or False**: There must be at least one thread that points into u.

(b) Let T be an extended binary search tree (that is, one having internal and external nodes). You visit the nodes of T according to one of the standard traversals (preorder, postorder, or inorder). Which of the following statements is necessarily true? (Select all that apply.)

(1) *Preorder traversal*: All the internal nodes appear *before* any of the external nodes

(2) *Inorder traversal*: Internal and external nodes *alternate* with each other

(3) *Postorder traversal*: The *first* node visited is an *external node*

(4) *Postorder traversal*: The *last* node visited is an *internal node*

(c) Given a binary max-heap with n entries ($n \geq 3$), you want to return the *third largest* item in the heap (without modifying its contents). What is the minimum number of heap entries that you might need to inspect to be certain that the third largest item is among them?

(d) When we delete an entry from a simple (unbalanced) binary search tree, we sometimes need to find a replacement key. Suppose that `p` is the node containing the deleted key. Which of the following statements are true? (Select all that apply.)

(1) A replacement is needed whenever `p` is the root

(2) A replacement is needed whenever `p` is a leaf

(3) A replacement is needed whenever `p` has two non-null children

(4) It is best to take the replacement exclusively from `p`'s right subtree

(5) At most one replacement is needed for each deletion operation

(e) You have an AVL tree containing n keys, and you insert a new key. As a function of n, what is the maximum number of rotations that might be needed as part of this operation? (A double rotation is counted as two rotations.) Explain briefly.

(f) Repeat the previous question in the case of deletion. (Give your answer as an asymptotic function of n.)

(g) What are the min and max number of nodes in a 2-3 tree of height 2? (Remember, *height* is the number of edges from the root to the deepest leaf.)

(h) You have just performed a deletion from a 2-3 tree of height h. As a function of h, what is the maximum number of key-rotations (adoptions) that might be needed as a result?

(i) The AA-tree data structure has the following constraint: "*Each red node can arise only as the right child of a black node.*" Which of the two restructuring operations (`skew` and `split`) enforces this condition?

(j) You have just inserted a key into an AA tree having L levels. As a function of L, what is the maximum number of skew operations that might be needed as a result? (Here we are only counting skew operations that have an effect on the structure, in the sense that a rotation is performed.)

(k) Splay trees are known to support efficient finger search queries. What is a "finger search query"?

(l) In class, we mentioned that when using double hashing, it is important that the second hash function $g(x)$ should not share any common divisors with the table size m. What might go wrong if this were not the case?

(m) Hashing is widely regarded as the fastest of all data structures for basic dictionary operations (insert, delete, find). Give an example of an operation that a tree-based search structure can perform *more efficiently* than a hashing-based data structure, and explain briefly.

(n) In the (unstructured) memory management system discussed in class, each available block of memory stored the size of the block both at the beginning of the block (which we called `size`) and at the end of the block (which we called `size2`). Why did we store the block size at both ends?

(o) You build a suffix tree for a string having m characters (including the `"$"` at the end). What is the maximum number of internal nodes in the tree?

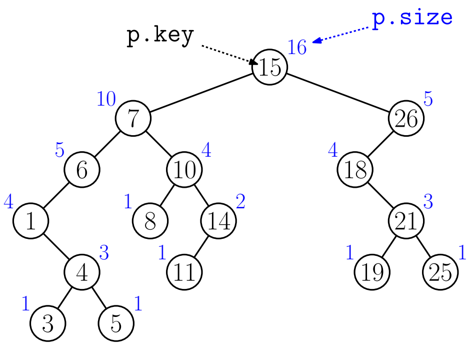
(p) You build a Bloom filter for a set X that uses k hash functions. Each hash function can be computed in $O(1)$ time. You query the data structure on an element x. Which of the following hold? (Select all that apply.)

(1) If $x \in X$, the data structure will correctly report this
(2) If $x \in X$, the data structure may or may not correctly report this
(3) If $x \notin X$, the data structure will correctly report this
(4) If $x \notin X$, the data structure may or may not correctly report this
(5) The query is answered in $O(k)$ worst-case time
(6) The query is answered in $O(k)$ expected time (but it might take longer)

Problem 2. This problem involves an input which is a binary search tree having n nodes of height $O(\log n)$. You may assume that each node `p` has a field `p.size` that stores the number of nodes in its subtree (including `p` itself). Here is the node structure:

```
class Node {
    int key             // key
    Node left, right    // children
    int size            // number of entries in this subtree
}
```

(a) Present pseudocode for a function `printMaxK(int k)`, which is given $0 \leq k \leq n$, and prints the values of the k largest keys in the binary search tree (see, for example, Fig. 1). You should do this in a single pass by traversing the relevant portion of the tree. It would be considered cheating to store all the elements of in a list, and then just print the last k entries of the list.



`printMaxK(2)` $= \langle 25, 26 \rangle$

`printMaxK(8)` $= \langle 11, 14, 15, 18, 19, 21, 25, 26 \rangle$

`printOdd()` $= \langle 1, 4, 6, 8, 11, 15, 19, 25 \rangle$

Figure 1: The functions printMaxK and printOdd.

For fullest credit, the keys should be printed in *ascending order*, and your algorithm should run in time $O(k + \log n)$ (see part (b) below). Briefly explain your algorithm.

Hint: I would suggest using the helper function `printMaxK(Node p, int k)`, where `k` is the number of keys to print from the subtree rooted at `p`.

(b) Derive the running time of your algorithm in (a).

(c) Give pseudocode for a function `printOdd()`, which does the following. Let $\langle x_1, x_2, \ldots, x_n \rangle$ denote the keys of the tree in ascending order, this function prints every other key, namely $\langle x_1, x_3, x_5, \ldots \rangle$ (see Fig. 1).

Again, you should do this in a single pass by traversing the tree. (For example, it would be considered cheating to traverse the tree and construct a list with all the entries, and then only print the odd entries of your list.) Your function should run in time $O(n)$. Briefly explain your algorithm.

Problem 3. In this problem you will write a program to check the validity of an AVL tree. The node structure is given below. All members are public.

```
class AVLNode {
    public int key                // key
    public int height             // height of this subtree
    public AVLNode left, right    // left and right children
}
```

In order to be valid, every node `p` of the tree must satisfy the following conditions (see Fig. 2):

- `p.height` is correct given the heights of its children. (Recall: `height(null) == -1`.)
- The absolute height difference between `p`'s left and right subtrees is at most 1
- An inorder traversal of the tree encounters keys in *strictly* ascending order

Problem 4. Assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$ that satisfies the *standard assumptions*. (That is, the cutting dimension alternates between x and y, subtrees are balanced, and the tree stores a bounding box `bbox` containing all the points of P.)

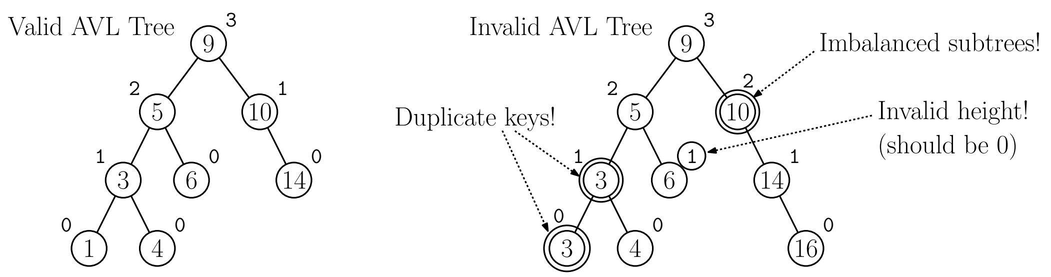


Figure 2: Valid and invalid AVL trees.

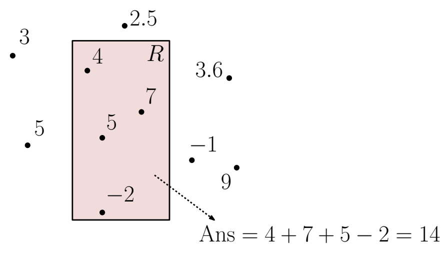


Figure 3: Weighted range query.

(a) In a standard range-counting query, we want to count the number of points in the range. Suppose that each point $p_i \in P$ has an associated real-valued weight w_i. In a *weighted orthogonal range query*, we are given a query rectangle R, given by its lower-left corner r_{lo} and upper-right corner r_{hi}, and the answer is the sum of the weights of the points that lie within R (see Fig. 3(b)). If there are no points in the range, the answer is 0.

Explain how to modify the kd-tree (by adding additional fields to the nodes, if you like) so that weighted orthogonal range queries can be answered efficiently. Based on your modified data structure, present an efficient algorithm in pseudo-code for answering these queries and explain. (For full credit, your algorithm should run in $O(\sqrt{n})$ time).

You may handle the edge cases (e.g., points lying on the boundary of R) however you like. **Hint:** You may use whatever helper function(s) you like, but I would suggest using:

```
double weightedRange(Rectangle R, KDNode p, Rectangle cell)
```

where `p` is the current node in the kd-tree, `cell` is the associated cell.

(b) Briefly analyze the running time of your algorithm, assuming that the tree is balanced. (You may apply/modify results proved in class.)

Problem 5. Assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$ that satisfies the *standard assumptions*. (That is, the cutting dimension alternates between x and y, subtrees are balanced, and the tree stores a bounding box `bbox` containing all the points of P.)

In a *fixed-radius nearest neighbor query*, we are given a point $q \in \mathbb{R}^d$ and a radius $r > 0$. Consider a circular disk centered at q whose radius is r. If no points of P lie within this disk, the answer to the query is `null`. Otherwise, it returns the point of P within the disk that is closest to q (see Fig. 4). Present (in pseudo-code) an efficient kd-tree algorithm that answers such a query.

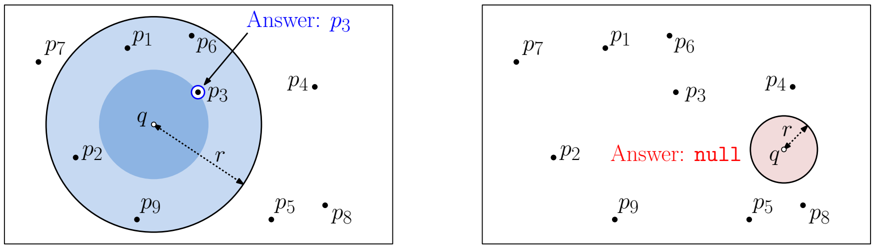


Figure 4: Fixed-radius nearest-neighbor query.

Hint: You may use whatever helper function(s) you like, but I would suggest using:

```
Point frnn(Point q, double r, KDNode p, Rectangle cell, Point best)
```

where `p` is the current node in the kd-tree, `cell` is the associated cell, and `best` is the best point seen so far.

You *do not* need to analyze your algorithm's running time, but explain it briefly. Your algorithm should not waste time visiting nodes that cannot possibly contribute to the answer.

Problem 6. Define a new treap operation, `expose(Key x)`. It finds the key `x` in the tree (throwing an exception if not found), sets its priority to $-\infty$ (or more practically `Integer.MIN_VALUE`), and then restores the treap's priority order through rotations. (Observe that the node containing `x` will be rotated to the root of the tree.) Present pseudo-code for this operation.

Problem 7. Suppose that you are given a treap data structure storing n keys. The node structure is shown in Fig. 5. You may assume that *all keys and all priorities are distinct.*

```
class TreapNode {
    Key key            // key
    int priority       // priority
    TreapNode left     // left child
    TreapNode right    // right child
}
```

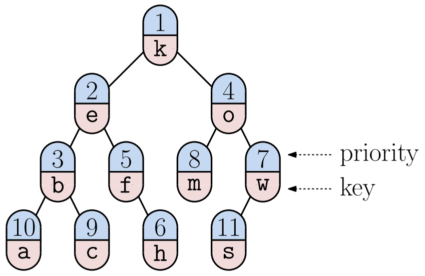


Figure 5: Treap node structure and an example.

(a) Present pseudocode for the operation `int minPriority(Key x0, Key x1)`, which is given two keys x_0 and x_1 (which may or may not be in the treap), and returns the lowest priority among all nodes whose keys x lie in the range $x_0 \leq x \leq x_1$. If the treap has no keys in this range, the function returns `Integer.MAX_VALUE`. Briefly explain why your function is correct.

For example, in Fig. 5 the query `minPriority("c", "g")` would return 2 from node `"e"`, since it is the lowest priority among all keys x where `"c"` $\leq x \leq$ `"g"`.

(b) Assuming that the treap stores n keys and has height $O(\log n)$, what is the expected-case running time of your algorithm? (Briefly justify your answer.)

Problem 8. In this problem we will build a suffix tree for the string $S =$ `baabaabababaa$`.

(a) List the substring identifiers for the 14 suffixes of S. For the sake of uniformity, list them in order (either back to front or front to back). For example, you could start with "`$`" and end with the substring identifier for the entire string.

(b) List the substring identifiers again together with their indices (0 through 13), but this time in alphabetical order (where `"a"` < `"b"` < `"$"`).

(c) Draw a picture of the suffix tree for S. For the sake of uniformity, when drawing your tree, use the convention of Fig. 7 in the Lecture 17 LaTeX lecture notes. In particular, label edges of the final tree with substrings, index the suffixes from 0 to 13, and order subtrees in ascending lexicographical order.

Problem 9. Suppose you have a large span of memory, which starts at some address `start` and ends at address `end-1` (see Fig. 6). (The variables `start` and `end` are generic pointers of type `void*`.) As the dynamic memory allocation method of Lecture 15, this span is subdivided into blocks. The block starting at address `p` is associated with the following information:

- `p.inUse` is 1 if this block is in-use (allocated) and 0 otherwise (available)
- `p.prevInUse` is 1 if the block immediately preceeding this block in memory is in-use. (It should be 1 for the first block.)
- `p.size` is the number of words in this block (including all header fields)
- `p.size2` each available block has a copy of the size stored in its last word, which is located at address `p + p.size - 1`.

(For this problem, we will ignore the available-list pointers `p.prev` and `p.next`.)

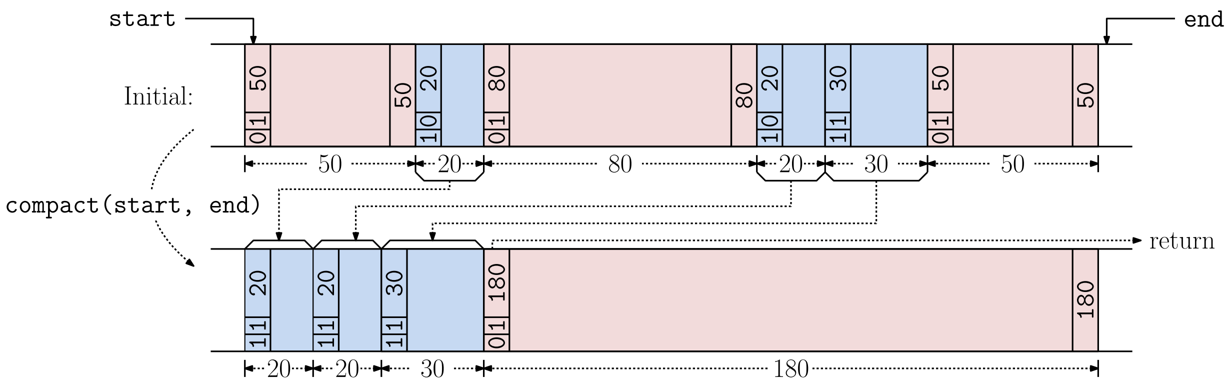


Figure 6: Memory compactor.

In class, we said that in real memory-allocation systems, blocks cannot be moved, because they may contain pointers. Suppose, however, that the blocks are movable. Present pseudo-code for a function that compacts memory by copying all the allocated blocks to a single contiguous span of blocks at the start of the memory span (see Fig. 6). Your function `compress(void* start, void* end)` should return a pointer to the head of the available block at the end. Following these blocks is a single available block that covers the rest of the memory's span.

To help copy blocks of memory around, you may assume that you have access to a function `void* memcpy(void* dest, void* source, int num)`, which copies `num` words of memory from the address `source` to the address `dest`.

Problem 10. This problem involves a data structure called an *erasable stack*. This data structure is just a stack with an additional operation that allows us to "erase" any element that is currently in the stack. Whenever we pop the stack, we skip over the erased elements, returning the topmost "unerased" element. The pseudocode below provides more details be implemented.

```
class EStack {      // erasable stack of Objects
    int top         // index of stack top
    Object A[HUGE] // array is so big, we will never overflow
    Object ERASED  // special object which indicates an element is erased

    EStack() { top = -1 }  // initialize

    void push(Object x) {  // push
       A[++top] = x
    }

    void erase(int i) {    // erase (assume 0 <= i <= top)
        A[i] = ERASED
    }

    Object pop() {         // pop (skipping erased items)
        while (top >= 0 && A[top] == ERASED) top--
        if (top >= 0) return A[top--]
        else return null
    }
}
```

Let n = `top` + 1 denote the current number of entries in the stack (including the `ERASED` entries). Define the *actual cost* of operations as follows: `push` and `erase` both run in 1 unit of time and `pop` takes $k + 1$ units of time where k is the number of `ERASED` elements that were skipped over.

(a) As a function of n, what is the *worst-case running time* of the `pop` operation? (For fullest credit, make your bound as tight as possible.) Justify your answer.

(b) Starting with an empty stack, we perform a sequence of m `push`, `erase`, and `pop` operations. Give an upper bound on the *amortized running time* of such as sequence. You may assume that all the operations are valid and the array never overflows. (For fullest credit, make your bound as tight as possible.) Justify your answer.

(c) Given two (large) integers k and m, where $k \leq m/2$, we start from an empty stack, push m elements, and then erase k elements *at random*, finally we perform a single `pop` operation. What is the *expected running time* of the final pop operation. You may express your answer asymptotically as a function of k and m.

In each case, state your answer first, and then provide your justification.

Problem 11. You are designing an expandable hash table using open addressing. Let m denote the current table size. Initially $m = 4$. Let us make the ideal assumption that each hash operation takes exactly 1 time unit. After each insertion, if the number of entries in the table is greater than or equal to $3m/4$, we expand the table as follows. We allocate a new table of size $4m$, create a new hash function, and rehash all of the elements from the current table into the new table. The time to do this expansion is $3m/4$.

(a) Derive the amortized time to perform an insertion in this hash table (assuming that m is very large). State your amortized running time and explain how you derived it. (For fullest credit, your running time should as tight as possible.)

Hint: The amortized time need not be an integer.

(b) One approach to decrease the amortized time is to modify the table expansion factor, which in this case is 4. In order to reduce the amortized time, should we *increase* or *decrease* this factor? If you make this adjustment, what negative side effect (if any) might you observe regarding the space and time performance of the data structure? Explain briefly.

‘

CMSC 420: Spring 2023

CMSC 420 (0201) - Final Exam

This exam is closed-book and closed-notes. You may use three sheets of notes (front and back). Write all answers on the exam paper. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 120 points. Good luck!

Problem 1. (30 points) Short answer questions. Unless requested, explanations are not required but may be given to help with partial credit.

(a) (2 points) You have an inorder-threaded binary tree with $n \geq 1$ nodes. As a function of n, how many threads are there? (Include the null threads in the leftmost and righmost nodes.)

(b) (2 points) You are given a leftist max-heap containing $n \geq 3$ keys (priorities), where all the keys are distinct. **True or False**: The node with the smallest key must be a leaf of the tree.

(c) (4 points) What are the min and max number of keys in a 2-3 tree of height 2? (Remember, *height* is the number of edges from the root to the deepest leaf. Note that we are asking about *keys*, not *nodes.*)

(d) (4 points) You have just performed an insertion into a 2-3 tree of height $h \geq 0$. As a function of h, what are the minimumm and maximum number of splits that might be needed as a result?

(e) (6 points) You have just inserted a large number n of keys into a scapegoat tree (no deletions). The next operation may be a find, insert, or delete. Which of the following statements is true? (Select all that apply.)

(1) The next operation is *find*, the worst-case (not expected-case) time is $O(\log n)$

(2) The next operation is *insert*, the worst-case (not expected-case) time is $O(\log n)$

(3) The next operation is *delete*, the worst-case (not expected-case) time is $O(\log n)$

(f) (4 points) In the memory management system described in class, which of the following elements are stored in each *allocated block*? (Select all that apply)

(1) The header, containing the `inUse` bit, `prevInUse` bit, and `size`

(2) Links to the `prev` and `next` elements in the available-block list

(3) The `size2` field at the end of the block

(4) The `time` field indicating when the block was most recently accessed

(g) (3 points) You have a splay tree with n keys. You perform a splay on an arbitrary key x and then immediately following this, you perform a splay on the very next key in ascending order. Which of the following is most accurate? (Select one.)

(1) The second splay takes $O(1)$ time in the worst case.

(2) The second splay takes $O(\log n)$ time in the worst case.

(3) One of the two splays takes $O(\log n)$ time, but we cannot predict which.

(4) The second splay takes $O(n)$ time in the worst case.

(h) (5 points) You have a Bloom filter that stores a set with a large number n of elements. The table has m entries and there are k hash functions. Which of the following is true? (Select all that apply.)

(1) The worst-case insertion time is (most accurately) $O(1)$.

(2) The worst-case insertion time is (most accurately) $O(m)$.

(3) The worst-case insertion time is (most accurately) $O(k)$.

(4) Assuming k is fixed, increasing m (generally) decreases the chance of a false positive.

(5) Assuming m is fixed, increasing k (generally) decreases the chance of a false positive.

Problem 2. (10 points) Recall that the *depth* of a node in a binary tree is the number of edges between this node and the root. In any binary tree, there are at most 2^d nodes at depth d. Given any AVL tree T and an integer $d \geq 0$, we say that T is *full at depth d* if it has the maximum possible number of nodes (namely, 2^d) at depth d.

Prove that for any $h \geq 0$, an AVL tree of height h is full at all depths from 0 up to $\lfloor h/2 \rfloor$. (For example, the AVL tree in Fig. 1 has height 4, and it is full at levels 0, 1, and 2, but it is not full at levels 3 and 4.)

Hint: Prove this by induction on the height of the tree.

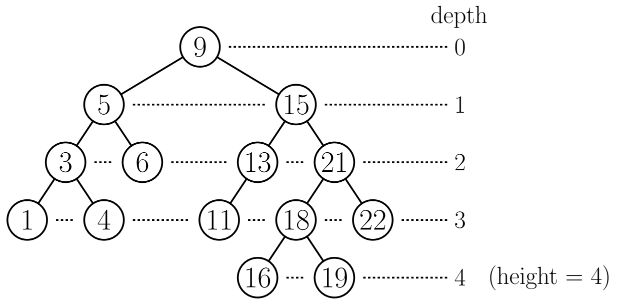


Figure 1: Full at depth.

Problem 3. (10 points) Consider the B-trees of order $m = 4$ shown in Fig. 2. When answering the questions below, assume the following conventions. Key rotation/adoption (when possible) takes precedence over splitting/merging. Second, when splitting a node, use the convention given in the lecture, that the left node has $\lceil m/2 \rceil - 1$ keys and the right node has $\lfloor m/2 \rfloor$ keys. (**Hint:** To save time, you need only redraw the portion of the tree that is affected by the operations.)

(a) (5 points) Show the final tree that results after inserting the key `13`.

(b) (5 points) Starting with the *original tree*, show the final tree that results after deleting the key `42`.

Problem 4. (15 points) In this problem we will build a suffix tree for $S =$ `"ababab$"`.

(a) (5 points) List the seven substring identifiers of S (from the last to the first).

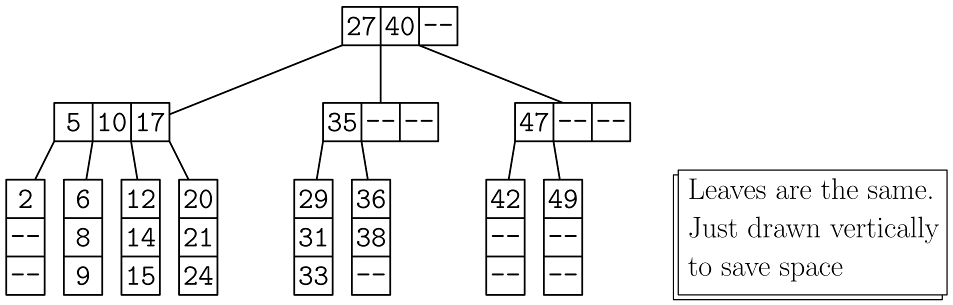


Figure 2: B-Tree Operations.

(b) (10 points) Draw the suffix tree for S. Use the same form as in the lectures. When drawing your tree, order the children of each node from left to right in the order `a` $<$ `b` $<$ `$`. (**Hint:** Be sure to compress paths wherever possible.)

Problem 5. (15 points) You are given a collection of vertical line segments, where the lower endpoints all lie on the x-axis and the upper endpoints lie in the first quadrant. This is presented to you as a set $P = \{p_1, \ldots, p_n\}$ of the upper endpoints of these segments, where each point `p` has coordinates `(p.x,p.y)`.

In a *segment-hitting query*, we are given a horizontal ray that is shot to the left of a query point $q = (q.x, q.y)$, and the problem is to count the number of segments that are hit by this ray (see Fig. 3).

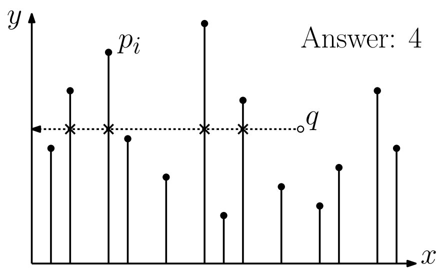


Figure 3: Segment-hitting queries.

(a) (3 points) You are asked to design an efficient kd-tree-based data structure for answering these queries. Explain what information is stored in the nodes of the tree, and what assumptions you make about the tree's structure (e.g., the choice of cutting dimensions and cutting values, tree balance, etc.)

(b) (8 points) Present pseudocode for an efficient function that answers segment hitting queries for a given query point `(q.x q.y)`. **Hint:** Here is a suggested helper:

```
int segHit(Point q, KDNode p, Rect cell)
```

where `q` is the query point, `p` is the current node of the kd-tree, `cell` is the rectangular cell associated with `p`.

(c) (4 points) What is your algorithm's running time? Explain briefly. (You may use facts proven in class about kd-trees.)

Problem 6. (20 points) Consider strings over the alphabet $\Sigma = \{0, 1, 2\}$. We say that a string is *monotone* if its characters (in left to right order) are monotonically non-decreasing. That is, given a string "$s_0 s_1 \ldots s_{k-1}$", we have $s_i \geq s_{i-1}$, for $1 \leq i \leq k-1$.

For example, the strings "012", "02", and "22222" are all monotone because, when read from left to right, the digit values never decrease. However, the string "00102" is not (because the fourth character "0" is smaller than its predecessor "1").

The objective of this problem is, given a trie containing a set of strings, count the number monotone strings in the set. Throughout, you may assume that the set of strings is *prefix free*, that is, no string is a prefix of any other.

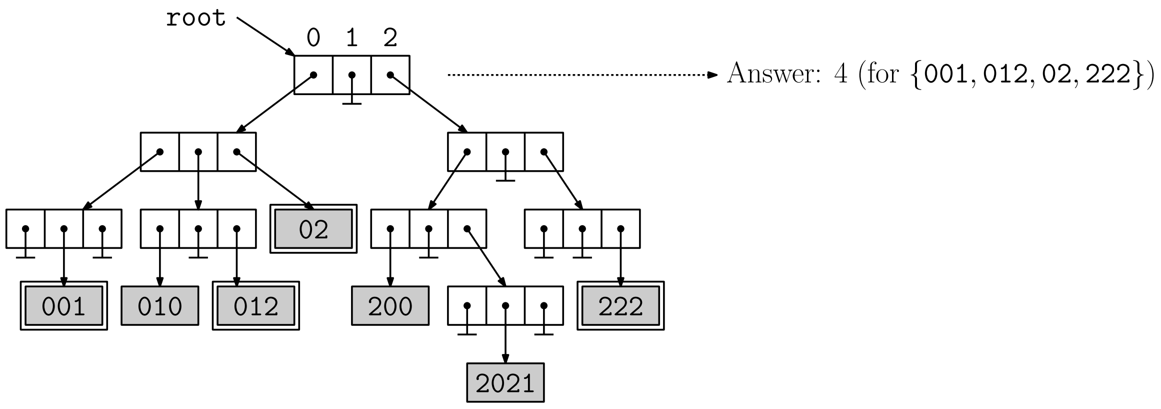


Figure 4: Monotone Strings: Standard Trie.

(a) (10 points) Assume that the trie is represented in standard (uncompressed) form as an extended tree. (See Fig. 4.) Each internal node `p` stores an array `p.child[0..2]`. (For example `p.child[2]` points to the subtree of strings whose next character is "2".) External nodes store the final string as `p.key`. To distinguish between node types, each node stores a boolean, `p.isExternal`, which is `true` for external nodes and `false` otherwise. Let `root` denote the root of the tree.

Present pseudocode for an efficient function, `countMonotone(root)`, which returns a count of the number of monotone strings in the trie.

Hint: You may create any helper functions you need. You may use any standard string utility functions. Your answer may be expressed either iteratively or recursively, your choice.)

(b) (10 points) Repeat part (a), but now assuming that the trie is represented using a de la Briandais trie (see Fig. 5). Each internal node `p` stores a single character as the `p.key` and its first-child and next-sibling pointers, `p.firstChild` and `p.nextSibling`, respectively. External nodes store the final string as `p.key`. To distinguish between node types, each node stores a boolean, `p.isExternal`, which is `true` for external nodes and `false` otherwise.

Problem 7. (20 points) In this problem, we will consider a variant of double hashing, called *priority hashing* (or "prashing"). As with double hashing, we are given a hash table with m

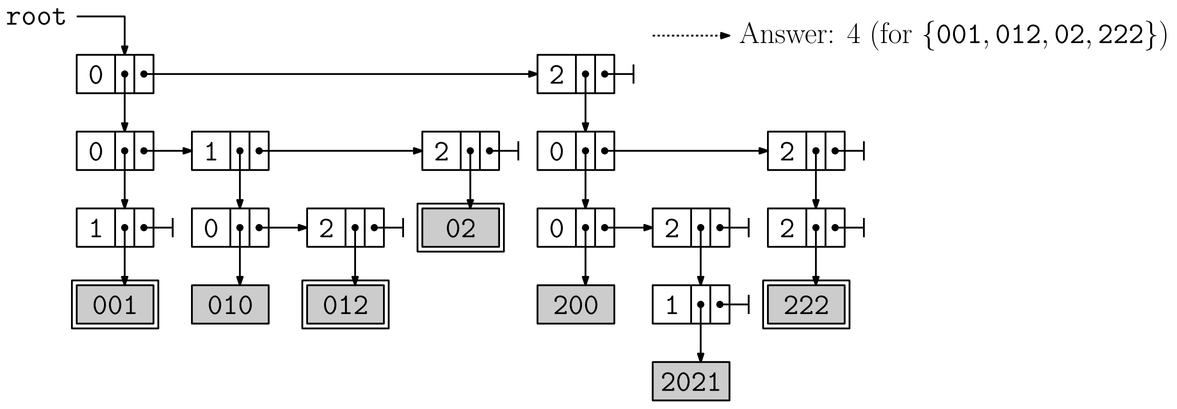


Figure 5: Monotone Strings: de la Briandais Trie.

entries, denoted $T[0..m-1]$. Each key x is associated with two hash functions, $h(x)$ and $g(x)$, and the probe sequence for key x consists of the indices

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

where all indices are taken modulo m. This is exactly the same as standard double hashing.

The new twist is that each key x is also associated with a numeric *priority function*, $p(x)$. Whenever a collision occurs, we compare the priorities of the two keys. The key with the higher priority gets to stay, and the other is "bumped", that is, it must move to the next position in its probe sequence. To simplify matter, assume that *no two keys have the same priority.* Here is the insert function:

```
void insert(Key x) {                   // insert key x in table T[0..m-1]
    insertHelper(x, h(x))
}

void insertHelper(Key x, int i) {      // try to insert x at T[i]
    i = i % m                          // wrap around
    if (T[i] == null) {                // empty?
        T[i] = x                       // yes, insert here
    } else {
        Key y = T[i]                   // collision with y
        if (p(x) < p(y)) {             // x has lower priority?
            insertHelper(x, i + g(x))  // bump x to its next position
        } else {                       // x has higher priority?
            T[i] = x                   // x takes y's spot
            insertHelper(y, i + g(y))  // bump y to its next position
        }
    }
}
```

(a) (10 points) Assume that we perform insertions, but *no deletions.* Present pseudocode for an efficient function `find(Key x)`, which searches for a (non-null) key x in the table. If it is in the table, return `true`, and otherwise return `false`.

(b) (5 points) Assume that for all keys x, the values of $g(x)$ and m are relatively prime (that is, they share no common factors other than 1).

True or False: If the hash table has at least one empty entry, the insert function will successfully insert x in the table (as opposed to looping infinitely).

Justify your answer. If true, give a clear argument as to why. If false, give a counterexample showing that it can go into an infinite loop.

(c) (5 points) Repeat part (b), but this time $g(x)$ and m need *not* be relatively prime.

Justify your answer. If true, give a clear argument as to why. If false, give a counterexample showing that it can go into an infinite loop.