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:

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

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	Total order	$O(n \log n)$	$O(n)$	Yes	No
HeapSort				No	Yes
QuickSort				Yes/No*	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: Fall 2022

Programming Assignment 0: Expanding 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 (but do check out the challenge problem at the end for extra credit points).

The Expanding Stack: In Lecture 2, we introduced a simple mechanism for implementing an array-based stack that automatically expands as needed by repeated size-doubling. In this assignment, you will implement a simple version of this data structure for storing string objects, called `ExpandingStack`. This data structure supports the following public functions.

`ExpandingStack(int initialCapacity)`: This creates an empty stack as an array of type `String` containing `initialCapacity` elements and sets `top` to -1. We assume that `initialCapacity` ≥ 1, and if not, it throws an `Exception` with the error message "`Invalid capacity`".

`void push(String x)`: This pushes the string `x` onto your stack. If there is not sufficient space in the current array, this allocates a new array of twice the current capacity, copies the elements of the old array into this new array, and then makes this new array the current one. In either case, you can now increment the top and a store `x` at this position in your array.

For example, suppose that your current array had capacity 8 and `top == 7`, meaning that the array is entirely filled. You would allocate an array of size 16, copy the 8 existing elements over, and make this new array the current one. You can now increment top to 8, and store `x` at this index.

`String pop()`: Assuming that `top` ≥ 0, this pops the element at index `top` off the stack, returning its value and decrementing `top`. Otherwise, it throws an `Exception` with the message "`Pop of empty stack`".

`String peek(int idx)`: Assuming that $0 \leq$ `idx` $\leq$ `top`, this returns the element at index `top` $-$ `idx` in your array. Thus, `peek(0)` returns the element at the top of the stack. If `idx` is not in this range, it throws an `Exception` with the message "`Peek index out of range`".

`int size()`: Returns the number of elements currently in the stack (or equivalently `top`$+1$).

`int capacity()`: Returns the current size of your array.

`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 $\langle$`"pig", "dog", "cat"`$\rangle$.

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, `ExpandingStack.java`, with all the function prototypes, and you just need to fill them in.

```
package cmsc420_f22; // Don't change this line

import java.util.ArrayList;

public class ExpandingStack {

    public ExpandingStack(int initialCapacity) throws Exception { ... }
    public void push(String x) { ... }
    public String pop() throws Exception { ... }
    public String peek(int idx) throws Exception { ... }
    public int size() { ... }
    public int capacity() { ... }
    public ArrayList<String> list() { ... }
}
```

Sample input/output: Here is an example of what the input and output might look like. Let us assume that the initial capacity is set to 4. Notice that when `"frog"` is pushed (resulting in 5 elements in the stack), we allocate a new array with capacity $2 \cdot 4 = 8$.

Input:	**Output:**
push:ladybug	push(ladybug): successful
list	list: ladybug
pop	pop: ladybug
list	list:
push:cat	push(cat): successful
push:dog	push(dog): successful
push:pig	push(pig): successful
push:cow	push(cow): successful
list	list: cow pig dog cat
size	size: 4
capacity	capacity: 4
peek:1	peek(1): pig
peek:-1	peek(-1): Failure due to exception: "Peek index out of range"
push:frog	push(frog): successful
list	list: frog cow pig dog cat
size	size: 5
capacity	capacity: 8

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 `ExpandingStack`. 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_f22`". Note that all must begin with the statement "`package cmsc420_f22`".

`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 `ExpandingStack.java`. It invokes the functions in your `ExpandingStack` class and outputs the results. It also catches and processes any exceptions.

▷ *You should not modify this file.*

Requirements: Because this is a very primitive data structure, your implementation should involve similarly primitive data structures. In particular, your stack contents should be stored in a simple Java array of type `String`. You should not use any additional Java data types, such as `ArrayList` or `LinkedList`. The only exception is the `list` operation (which is just there for debugging and testing purposes), which is allowed to use an `ArrayList` for storing its results.

When grading for efficiency, we require that all the operations run in constant time except for `list` and `push` (but only whenever a reallocation occurs). In both of these exceptional cases, the running time should be $O(n)$, where n is the current number of elements in the stack.

Style and Efficiency: Part of your grade (usually 5%) is based on having clean programming style and implementing the operations in an efficient manner. We have no formal requirements for what constitutes good style. Mostly, it involves that your code is clear and understandable to the grader. The efficiency requirements are mentioned above.

Challenge Problem: (Challenge problems are not graded separately from the assignment. After final grades have been computed, I may "bump-up" a grade that is slightly below a cutoff threshold based on these extra points.)

Ignoring the `list` operation (which is just there for debugging and testing purposes), all the stack operations run in constant time with the exception of when a `push` operation results in the stack overflowing. This operation runs in time proportional to the number of elements in the stack, which may be arbitrarily large. However, the amortized analysis in class shows that on average, each operation requires only *constant time*.

Implement the expanding stack so that all the operations run in constant *worst-case time* (irrespective of the number of elements in the stack). Our strict way of enforcing this is that (ignoring the `list` operation), your program *may not use any looping constructs of any kind*. That is, you may not use `for` or `while` loops, and recursive function calls are not allowed. For example, the following is allowed:

```
A[0] = B[0];
A[1] = B[1];
A[2] = B[2];
```

But this is not:

```
for (int i = 0; i < 3; i++) A[i] = B[i];
```

You are **not** allowed to cheat by invoking Java's built-in functions which do copying behind the scenes (such as `ArrayList` or `java.util.Arrays.copyOf` or `System.arraycopy`.)

Note that there is no need for trickery, but some relaxation to the assignment specifications is needed. Rather than waiting until the array capacity is exceeded, you are allowed to allocate additional array storage any time you like, and you may (prematurely) copy elements into this array prior to the event when the reallocation is actually required. You cannot use any loops, recursion, or any Java functions to assist you with the copying process.

You may assume that allocating a new array of any size runs in constant time. (This is a tiny lie, because Java automatically initializes arrays to zero. But you don't need to rely on this to solve this problem.)

```
String[] A = new String[100000]; // this runs in constant time
```

Also, assigning one array to another (a so-called "shallow copy") also takes constant time. But note that this does not copy individual elements. It just results in two variables that point to the same block of memory.

```
String[] B = A; // now B and A both point to the same array (constant time)
```

If you attempt the challenge, please insert a comment at the top of your `ExpandingStack.java` file explaining that you did this and briefly (in a few sentences) how you did it:

```
// I did the challenge problem. Here's how. ...

package cmsc420_f22;
import java.util.ArrayList;

public class ExpandingStack {
...
}
```

CMSC 420: Fall 2022

Programming Assignment 1: Leftist Heaps

Overview: In this programming assignment you will implement a leftist heap, the mergeable heap structure presented in Lecture 5. Your implementation will support all the basic functions of a mergeable priority queue, namely `insert`, `extract-min`, and `merge`. There will also be a couple of additional operations including an operation to list the contents of your tree structure so we can check its correctness.

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.

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

`boolean isEmpty()`: Returns `true` if the current heap has no entries and `false` otherwise.

`void clear()`: This resets the structure to its initial state, removing all its existing contents.

`void insert(Key x, Value v)`: This inserts the key-value pair $(\texttt{x}, \texttt{v})$, where `x` is the key and `v` is the value. (**Hint:** This can be done with a single call to the utility function `merge`, without the need of loops or recursion.)

`void mergeWith(LeftistHeap<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.

`Key getMinKey()`: This returns the smallest key in the heap, but makes no changes to the heap's contents or structure. If the heap is empty, it returns `null`.

`Value extractMin()`: If the heap is empty, this throws an `Exception` with the error message `"Empty heap"` Otherwise, this locates the entry with the minimum key value, deletes it from the heap, and returns its associated value. (**Hint:** This can be done with a single call to the utility function `merge`, without the need of loops or recursion.)

`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.npl + "]"` with the node's key, value, and npl value to the end of the array-list. (The symbol "␣" is a space.) Then recursively visit `u.right` and then `u.left`.

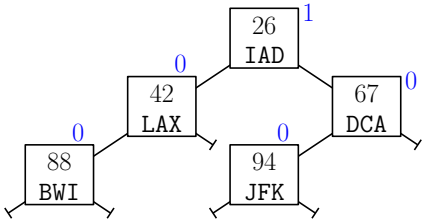


Figure 1: A leftist heap.

An example of the output on the tree shown in Fig. 1 is shown below.

Index	Array-List Contents
0:	`(26, IAD) [1]`
1:	`(67, DCA) [0]`
2:	`[]`
3:	`(94, JFK) [0]`
4:	`[]`
5:	`[]`
6:	`(42, LAX) [0]`
7:	`[]`
8:	`(88, BWI) [0]`
9:	`[]`
10:	`[]`

This format has been chosen for a particular reason. It is very easy to produce a nicely formatted output based on this. Given the above output, our program will generate the following structured output. If you rotate it 90° clockwise, it looks quite similar to the tree structure of Fig. 1.

```
Formatted structure:
  | (67, DCA) [0]
  | | (94, JFK) [0]
  (26, IAD) [1]
  | (42, LAX) [0]
  | | (88, BWI) [0]
```

Split: The operations described above follow directly from the code given in class. We would like you to implement one more operation. This is more challenging. It is worth 10 points. So, if you do not implement it correctly, you will still get most of the credit for the assignment.

`LeftistHeap<Key, Value> split(Key x)`: Given a key `x`, this splits the heap into two, the current one contains all the entries whose keys are less than or equal to `x` and all the entries whose keys are strictly greater than `x` are moved into a new heap, which is returned.

For the sake of efficiency (and so we can test your output), the process should be implemented as follows. First, we create an empty list (e.g., a Java `ArrayList`) of nodes. Next, perform a left-to-right preorder traversal of the current tree. When we visit a node `u`, if `u.key` $\leq$ `x`, then leave the node unchanged and apply the traversal recursively,

first to the left subtree and then to the right subtree. On the other hand, if `u.key` $>$ `x`, unlink this node from the current tree and append it to the end of the list. By heap ordering, we know that all the nodes of this subtree will be placed in the new heap.

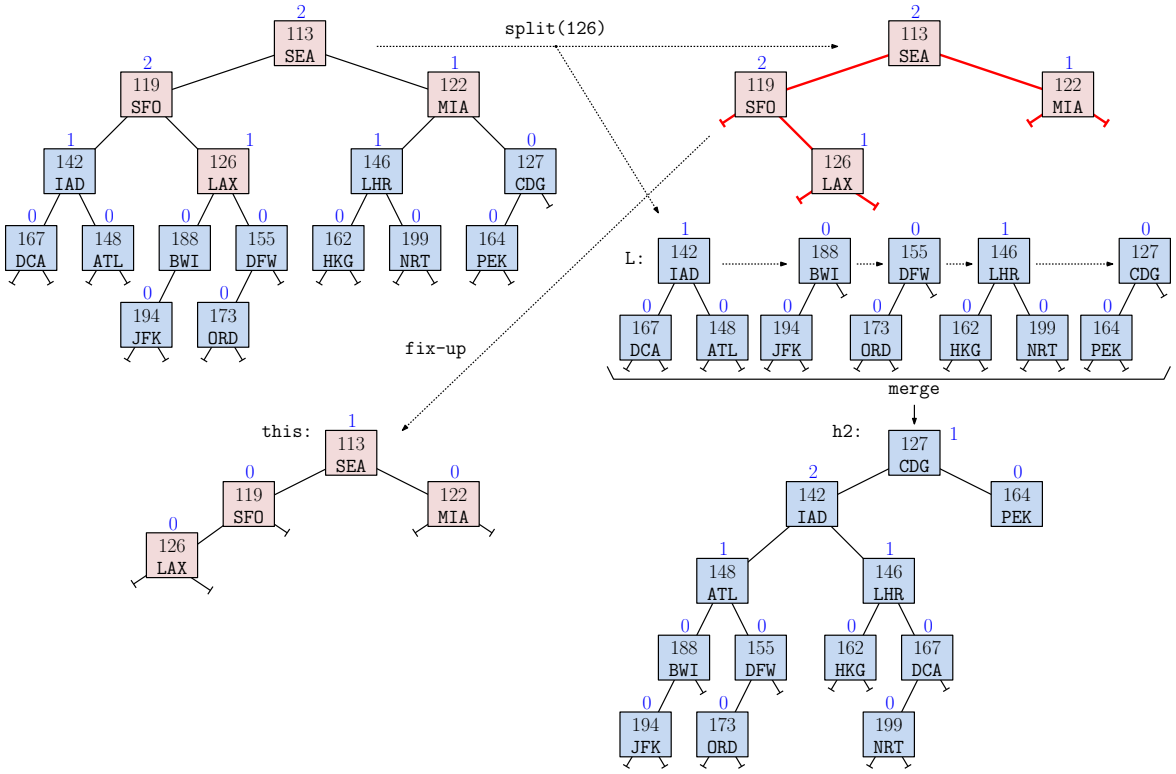


Figure 2: Splitting a leftist heap. We traverse the tree, unlinking all subtrees whose key value strictly exceeds `x = 126`. We merge these trees (from left to right) to form the final result `h2`. We traverse the current tree, update the npl values and swap subtrees so that the leftist property holds.

When this process returns, we have a list $L = \langle u_1, \ldots, u_k \rangle$ of maximal subtrees whose nodes are to be placed in the new heap. These are ordered from left to right. We create a new empty leftist heap, called `h2`, and we merge each of the elements of L into this new heap, from left to right. That is, we perform `h2.mergeWith`(u_i) for $i = 1, 2, \ldots, k$. (Unfortunately, this statement is not kosher because u_i is not a heap, it is just a node. But this is effectively what your program should perform.)

Finally, the original tree has had a number of subtrees removed from it. As a result, the npl values may be wrong and the leftist property may be violated. (Note that the other tree, `h2` will be valid, because it was formed through merges.) To fix it, perform a traversal of the tree, compute the proper npl values, and perform left-right child swaps whenever needed to enforce the leftist property.

Now, both trees have their proper contents and satisfy the required properties. Finally, return a reference to the leftist heap `h2` as the final result.

Note that there are many different valid leftist heaps containing a given set of nodes, and

if your implementation differs from the one described above, you will obtain a different tree and your results will not match ours.

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 `LeftistHeap.java`. Remember that you must use the package "`cmsc420_f22`" 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. Below is a short summary of the contents of `LeftistHeap.java`.

Class Structure: The high-level `LeftistHeap` class structure is presented below. 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 `LHNode`, 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 LeftistHeap<Key extends Comparable<Key>, Value> {

    class LHNode {                      // recommended node (you may change)
        Key key;                        // key (priority)
        Value value;                    // value (application dependent)
        LHNode left, right;             // children
        int npl;                        // null path length
        // ... any utility functions you want to define
    }

    // ... any private and protected members you need

    public LeftistHeap() { ... }        // constructor
    public boolean isEmpty() { ... }    // is the heap empty?
    public void clear() { ... }         // clear its contents
    public void insert(Key x, Value v) { ... }                // insert (x,v)
    public void mergeWith(LeftistHeap<Key, Value> h2) { ... } // merge with h2
    public Key getMinKey() { ... }                            // get min key
    public Value extractMin() throws Exception { ... }        // extract min
    public ArrayList<String> list() { ... }                   // list entries
}
```

Efficiency requirements: The functions `insert`, `mergeWith`, and `extractMin` should all run in $O(\log n)$ time. The function `getMinKey()` should run in $O(1)$ time. The function `list()` should run in time proportional to the size of the tree. A portion of your grade will depend on the efficiency of your program.

The function `split` should run in time $O(k \log n)$, where k is the number of subtrees that need to be merged together. (In the worst case, k may be as high as $O(n)$, but your function should be more efficient when k is small. If you follow the outline that we have given for `split`, you should achieve this running time.)

The public interface should match what you are given in the skeleton code. You are free to add whatever private and protected members (both data and functions, subject to these efficiency requirements.)

Testing/Grading: Submissions will be made through Gradescope (you need only upload your modified `LeftistHeap.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.

The total point value is 80 points. Of these, 60 points will be for the heap operations excluding `split`. Correctly implementing `split` is worth 10 points, and additional 10 points is reserved for clean programming style and the above efficiency requirements.

CMSC 420: Fall 2022

Programming Assignment 2: Extended kd-Trees

Overview: In this assignment we will implement a variant of the kd-tree data structure, which will call an *extended kd-tree* (or `XkdTree`) to store a set of points in 2-dimensional space. This data structure involves a number of practical extensions over the standard kd-tree covered in class. First, the tree will be extended, meaning that we distinguish between *internal nodes*, whose only function is to subdivide space, and *external nodes* (or *leaves*), where the points are actually stored. Second, we allow each external node to a small number of points, depending on a parameter called the *bucket size*.

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. 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.

Each point to be stored in the data structure will have an associated value, called its *label*. In our case, the label is just a Java `String`. The resulting object is called a *labeled point*. Rather than impose a particular class structure, a labeled point is any class that supports a Java interface, which we call `LabeledPoint2D.java`. Here is the 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)
}
```

As the implementer of the data structure, you do not need to worry about the actual objects being stored, as long as you access the object through the interface functions. Again, all of these will be provided to you in our skeleton code.

Extended kd-Tree: The `XkdTree` data structure will be templated with the labeled-point type. Since this is any object that implements the `LabeledPoint2D` interface, we will call it `LPoint`. Your class declaration (which we will provide you) looks like this:

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

An *extended kd-tree* involves two modifications to the standard kd-tree as discussed in lecture (see Fig. 1). First, the tree is extended, meaning that there are two types of nodes, *internal nodes* and *external nodes*. Second, rather than holding a single point, each external node stores a small set of points, called a *bucket*. The actual number ranges from 0 up to some

maximum number, called the *bucket size*. The bucket size is specified by the user when the data structure is first constructed. You should think of it as a small positive integer, ranging from perhaps 1 up to 10.

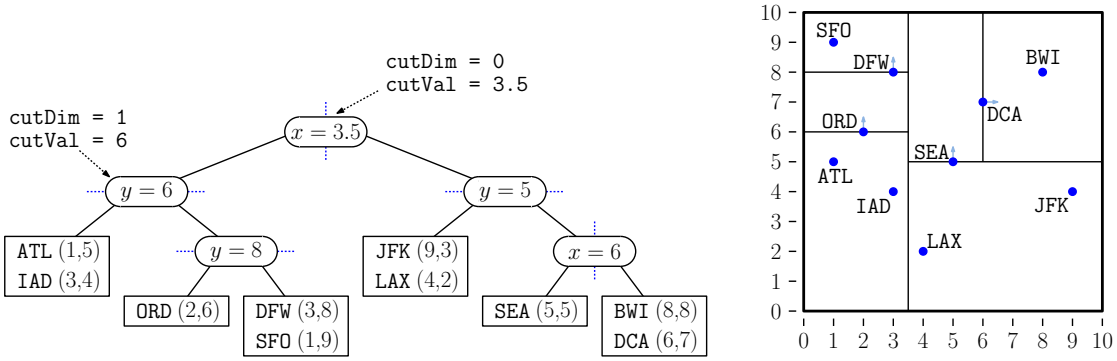


Figure 1: An example of an extended kd-tree with bucket size 2 and bounding box $[0, 10] \times [0, 10]$.

Internal Nodes: Each internal node stores the splitting information, consisting of a *cutting dimension* and a *cutting value*. The cutting dimension (or `cutDim`) indicates which axis (0 for x and 1 for y) is used to determine which subtree the points belong to. The cutting value (or `cutVal`) indicates where the cut occurs along this axis (see Fig. 1).

For example, if the cutting dimension is 0 (for x) and the cutting value is 5, then a point $p = (p_x, p_y)$ will be put in the left subtree if $p_x < 5$ and in the right subtree if $p_x \geq 5$. Note that the cutting value does *not* need to be the coordinate of any point in the tree.

External Nodes: Each external node stores a list (e.g., a Java `ArrayList`) of the labeled points that lie within this node. The size of this list can vary between zero and the data structure's bucket size.

Note that there are no `null` pointers. Every internal node has two non-null children, which may be external nodes, and external nodes have no children by definition. What if the tree is empty? An empty tree is represented setting the `root` to point to a single external node whose bucket is empty.

Requirements: Your program will implement the following functions for the `XkdTree`. While you can implement the data structure internally however you like (subject to the style and efficiency requirements given below), the following function signatures should not be altered. Recall that `Point2D` is a 2-dimensional point, and an `LPoint` is any object that implements `LabeledPoint2D`.

`XkdTree(int bucketSize, Rectangle2D bbox)`: This constructs a new (empty) `XkdTree` with the given bucket size and bounding box.

`void clear()`: This removes all the entries of the tree.

`int size()`: Returns the number of points in the tree. For example, for the tree of Fig. 1, this would return 10. For efficiency, you should store this value in your structure, rather than compute it by traversing the tree.

`LPoint find(Point2D pt)`: Determines whether a point coordinates `pt` occurs within the tree, and if so, it returns the associated `LPoint`. Otherwise, it returns `null`. (Note that the query is a standard 2-dimensional point, while the output is a labeled point.)

Unlike standard kd-trees, if `pt` lies directly on the cutting line (that is, `pt[cutDim] == cutVal`), this point might lie in either the left subtree or the right subtree. (In the standard kd-tree presented in class, it always lies in the right subtree.) Whenever you visit a node where this is the case, you will need to check *both* subtrees of an internal node. If it is found in either, then return the associated labeled point. If it is found in neither, return `null`.

`void insert(LPoint pt) throws Exception`: Inserts a single labeled point `pt` in the tree. You may assume that there are no duplicate points, but there may be duplicate coordinate values. Inserting a single point is functionally equivalent to performing a bulk insertion (see below) with a point set of size one.

`void bulkInsert(ArrayList<LPoint> pts) throws Exception`: Inserts a set of one or more labeled points, `pts`, in the tree. You may assume there are no duplicate points. If any point lies outside the bounding box, you should throw an `Exception` with the error message `"Attempt to insert a point outside bounding box"`.

The insertion is performed as follows. First, we determine the external nodes into which each of the points falls. (If the point lies on the splitting line of an internal node, insert it into the right subtree.) Next, consider each of the external node that has received at least one new point. We merge all the new points together with the existing points of this node's bucket. (The built-in Java function `addAll` is useful for doing this.) If the number of total number of points is not greater than the bucket size, we are done. Otherwise, we need to split the bucket, as described next.

Splitting (Big View): The splitting process works as follows. First, we compute the smallest bounding axis-aligned rectangle containing all the points. (The function `expand`, which is provided in `Rectangle2D` is useful for doing this.) If this rectangle's width is greater than or equal to its height, the cutting dimension is set to 0 (that is, x), and otherwise it is set to 1 (that is, y).

Next, compute the median coordinate along the cutting dimension and partition the set about this median element into two sets, call them `L` and `R`. We create a new internal node having this cutting dimension and set its cutting value to be the median coordinate. The elements of `L` are then recursively inserted into its left subtree and the elements of `R` are recursively inserted into the right subtree. Whenever the number of remaining points is less than or equal to the bucket size, we create a new external node and store the points there.

Splitting (Details): Care must be taken with the splitting process to avoid issues that arise when points have duplicate coordinates along the cutting dimension. If multiple points share the same coordinate value as the median, our usual convention would put all of these points into the right subtree. But this may result in a highly imbalanced tree (and even infinite looping if you are not careful). In order to ensure that the tree is balanced, we want every partition to be as balanced as possible.

Here is how to implement this. (For the sake of uniformity in grading, it is a *requirement* that you partition your points in this way.) Once the cutting dimension

has been determined, sort the points according to cutting dimension with ties broken by other coordinate. Suppose that there are n points, and let `points[n]` denote the sorted sequence of points (this is probably an `ArrayList`). Let $m = \lfloor n/2 \rfloor$. If n is odd, the cutting value is taken to be the unique median element, `points[m]`. If n is even, the cutting value is taken to be the mean of the lower and upper medians, that is, (`points[m - 1]` + `points[m]`)/2.

Next, define the left list `L` to be the sublist consisting of the first m elements. Let's use the notation `points[i,j]` to denote the subarray from `points[i]` to `points[j-1]`. We have `L = points[0,m]`. Define the right list `R` to be the sublist consisting of the last $n - m$ elements, that is, `R = points[m,n]`. (Note that Java provides a handy function, called `sublist`, which can be used to partition the list.)

Sorting Labeled Points: You might be wondering, "Do I need to write my own sorting function?" (Answer: No, you should *never* write your own sorting algorithm.) Java provides a flexible method sorting based on various criteria. There is a 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. (A Google search for "`Java sorting with a Comparator`" will reveal more examples.)

```
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 */
    }
}
```

`void delete(Point2D pt) throws Exception:` (There is no deletion function required for this assignment.)

`ArrayList<String> list():` This operation generates a right-to-left preorder traversal of the nodes in the tree. (Recall that `right-to-left` means that we visit the right subtree before the left.) 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 + ")"` or `"(y=" + cutVal + ")"`

External nodes: This generates list of points in this bucket surrounding by square brackets, `"[" + ... + "]"`. The "`...`" is a list of the labeled points in the bucket,

each enclosed in curly braces `"{" + ... + "}"`. The points should be sorted by their string labels (another `Comparator`!). To output each labeled point, you can invoke the function `point.toString()`, which is defined in `Airport.java`.

For example, here is the result for the tree of Fig. 1. (Take note of spaces.)

```
(x=3.5)
(y=5.0)
(x=6.0)
[ {BWI: (8.0,8.0)} {DCA: (6.0,7.0)} ]
[ {SEA: (5.0,5.0)} ]
[ {JFK: (9.0,3.0)} {LAX: (4.0,2.0)} ]
(y=6.0)
(y=8.0)
[ {DFW: (3.0,8.0)} {SFO: (1.0,9.0)} ]
[ {ORD: (2.0,6.0)} ]
[ {ATL: (1.0,5.0)} {IAD: (3.0,4.0)} ]
```

Note that our autograder is sensitive to both case and whitespace. Our command-handler program will convert this into a formatted tree structure:

```
Tree structure:
  | | | [ {BWI: (8.0,8.0)} {DCA: (6.0,7.0)} ]
  | | (x=6.0)
  | | | [ {SEA: (5.0,5.0)} ]
  | (y=5.0)
  | | [ {JFK: (9.0,3.0)} {LAX: (4.0,2.0)} ]
  (x=3.5)
  | | | [ {DFW: (3.0,8.0)} {SFO: (1.0,9.0)} ]
  | | (y=8.0)
  | | | [ {ORD: (2.0,6.0)} ]
  | (y=6.0)
  | | [ {ATL: (1.0,5.0)} {IAD: (3.0,4.0)} ]
```

`LPoint nearestNeighbor(Point2D center)`: This function is given a query point (regular point, not a labeled point), and it computes the closest point. If the tree is empty, it returns `null`. Otherwise, it returns a reference to the closest `LPoint` in the kd-tree. For example, in Fig. 2, the nearest neighbor for `center` $= (6, 4)$ would return a reference to the point labeled `LAX`.

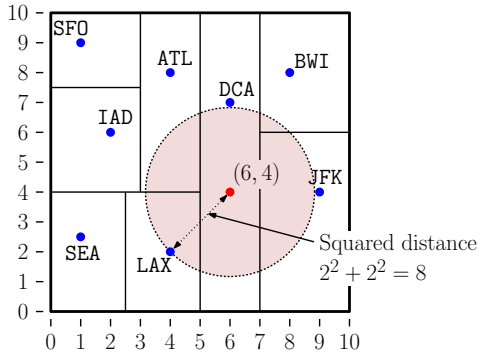


Figure 2: Nearest-neighbor query using squared distances.

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`.

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.)

If there are multiple nearest neighbors of the query point, you may return any one of them. (We will engineer the test data so this never happens, so there should be no variance with our expected outputs.)

Java Node Structure: Because we have two types of nodes, we will need a more sophisticated node structure. We use good object-oriented principles by defining a parent node type, called `Node`. This is abstract, which means that we never create such a node. The actual nodes are called `InternalNode` and `ExternalNode`. (You may name them whatever you like.) Internal nodes store splitting information (cutting dimension and cutting value) and subtrees. External nodes stores the bucket, which might be stored as a Java `ArrayList`.

An example of how this might look is shown below. Since this is all internal, you are free to implement however you like. But, I would strongly encourage you to use object inheritance, since this is just good programming style.

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

    private abstract class Node { // generic node (purely abstract)
        abstract LPoint find(Point2D pt); // find helper - abstract
        // ... other helper functions omitted
    }

    private class InternalNode extends Node {
        int         cutDim;      // the cutting dimension (0 = x, 1 = y)
        double      cutVal;      // the cutting value
        Node        left, right; // children

        LPoint find(Point2D q) { /* find helper for internal nodes */ }
        // ... other helper functions omitted
    }

    private class ExternalNode extends Node {
        ArrayList<LPoint> points; // the bucket

        LPoint find(Point2D pt) { /* find helper for external nodes */ }
        // ... other helper functions omitted
    }
```

```
    // ... the rest of the class
}
```

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.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You will need to fill in the implementation of the `XkdTree.java`. We also provide some utility classes (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, if you wanted 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_f22`" 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) For the sake of partitioning, your bulk insert function is allowed to sort the points being inserted at each node of the tree. (You are encouraged to think about more efficient methods. It is obviously inefficient to sort the points by x at one node, then resort them by y at its child, and then resort them again by x at its grandchild, but this is the sort of thing that the above construction algorithm will do.) As mentioned above, you should use Java's `Collection.sort` (or some other built-in) sorting function, as opposed to implementing your own bubble sort algorithm (ugh!) or the like.

You should have a variable that tracks the number of points in the tree in order to answer the `size` operation efficiently.

The nearest-neighbor algorithm should follow the structure given in the lecture notes. Minor variations are allowed, but your implementation should have the two major features as the one given in class. First, it should prioritize visiting on the side of the tree that is closer to the query point, and second, it should not visit subtrees that cannot possibly contain the nearest neighbor.

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. 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 1, on the class Project Page.)

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. You only need to submit your `XkdTree.java` file. If you created any additional files for utility objects, you will need to upload those as well.

Challenge Problem: (Remember, challenge problems count for extra-credit points, which are only taken into consideration after the final cutoffs have been set.)

Implement a deletion operation. This is a public function with the following function declaration:

```
public void delete(Point2D pt) throws Exception
```

If the point is not found in the tree, the deletion function throws an `Exception` with the error message `"Deletion of nonexistent point"`. Otherwise, it removes this point from the kd-tree. If the deletion results in an external node becoming empty, and this external node is not the root, then the tree should be restructured to eliminate this empty external node. This will induce other changes in the tree. Since an internal node cannot have a `null` child, removing an external node results in the removal of its parent. The external node's sibling will now be *promoted*, in the sense that it will become a child of its former grandparent. Part of the challenge is for you to figure out these changes.

Note that repeated deletions can cause the tree to become unbalanced, but there is no requirement that you do anything about rebalancing the tree. (When we study scapegoat trees, we'll see that there is an easy way to do this.)

CMSC 420: Fall 2022

Programming Assignment 2: Tips on Bulk-Insertion

The principal differences between the data structure in Programming Assignment 2 and the standard kd-tree are (1) it is based on an extended binary tree (with points stored only in the leaves, not the internal node), (2) leaf nodes can hold multiple points (based on the bucket size), (3) points can be inserted "in-bulk" and the splitting process depends on the distribution of these points.

Bulk-insertion is the most complicate of the operations. In this handout, we will discuss the tree's node structure and how to perform bulk insertion with an example.

Node Structure: As mentioned in the assignment handout, the easiest way to implement an extended binary tree in Java is to use an inner class for the nodes, where there is a parent class `Node` and two subclasses `InternalNode` and `ExternalNode` derived from this. You should expect to have helper functions for each of your major operations (e.g., `find`, `bulkInsert`, `list`, and so on). The internal-node helpers mostly serve to direct points down to the appropriate external nodes, and the external node helpers do most of the real work.

These are abstract member functions, which means that Java will invoke the appropriate function depending on the node type. For example, given a node pointer `p`, the call `p.find(q)` will invoke the `InternalNode` find function if `p` is an internal node and the `ExternalNode` find function if `p` is an external node. (If you do this properly, you should not need to resort to checking a node's type using "`instance of`".)

Bulk Insertion: Let's consider this in the general case, from the perspective of a tree that already contains some points. Consider the bulk insertion of five points into the tree shown in Fig. 1(a), and suppose that the bucket size is two.

Helpers: You will have helper functions for both internal and external nodes. Both will take a list of points (actually, a list of type `LPoint`) as the argument.

Internal node: The helper for the internal node takes the list of points and splits it into two sublists consisting of the points to be placed in the left subtree and those for the right subtree. This is based on the cutting dimension and cutting value. There are many ways to perform this partition. I believe that the easiest (even if not the fastest) is to sort the points according to the cutting dimension, determine the index where to split the list, and then use Java's `subList` function to do the actual partition. (Remember that our convention is that points that fall on the splitting line are placed in the right subtree.)

For example, in Fig. 1(b), we start by sorting the input along the cutting dimension of the root, which is x, to obtain the list `[SFO, ORD, DFW, SEA, DCA]`. We partition this about $x = 5$ into the sublists `[SFO, ORD, DFW]`, which we recursively insert to the left subtree and `[SEA, DCA]`, which we recursively insert to the right.

This continues at all the internal nodes. For example, at the internal node "$y = 6$" the list is split based on the y-coordinate into the sublist `[SEA]`, which is sent to the left subtree and `[DCA]`, which is sent to the right.

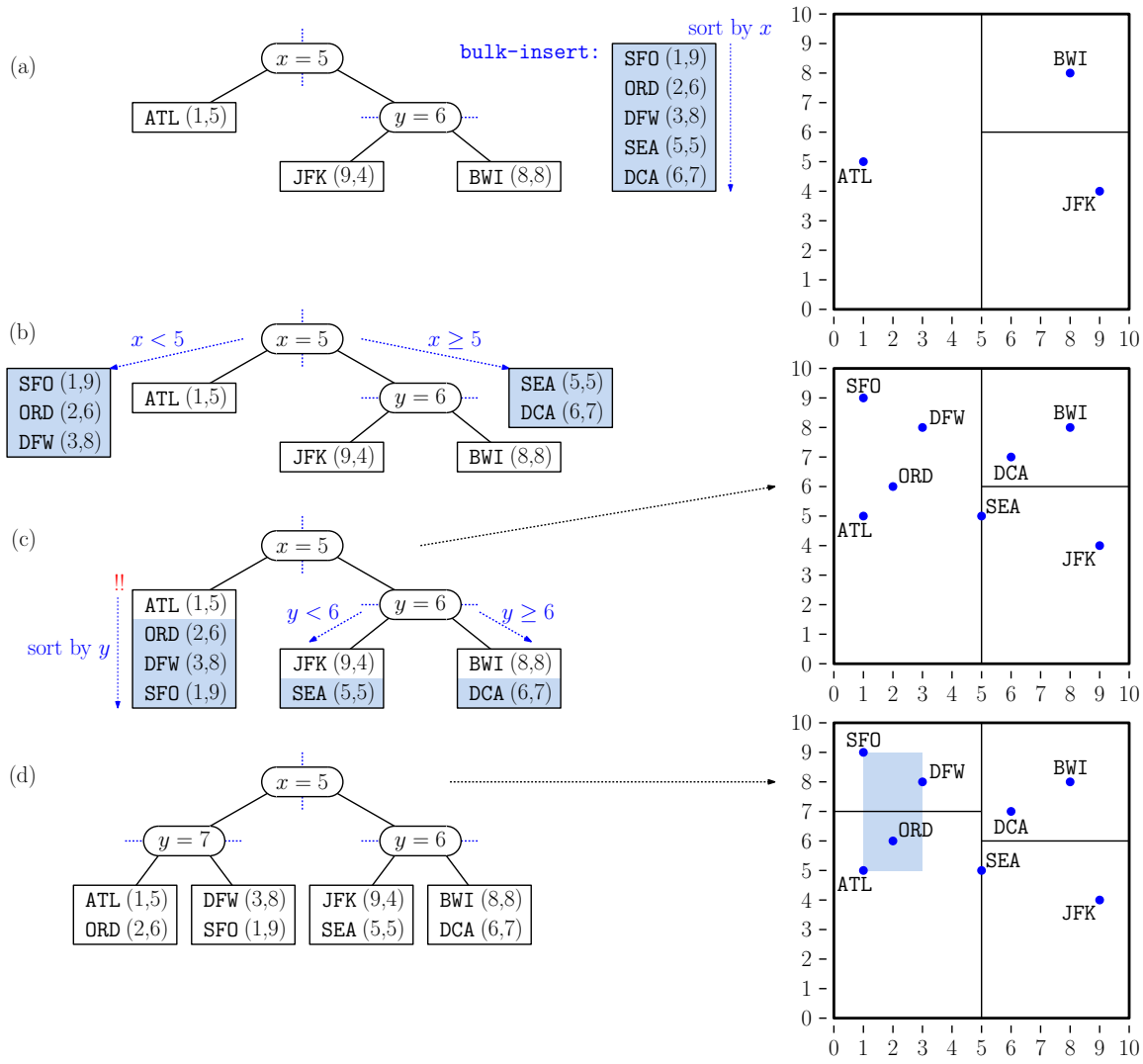


Figure 1: Bulk-insertion of five points in a tree with bucket size two.

External node: When the points arrive at an external node, they are added to the associated bucket list in this node. If the number of points does not exceed the bucket size, then we are done. (See the external nodes containing `JFK` and `BWI` in Fig. 1(c).)

Otherwise, we need to split this node. To do so, we first compute the bounding box for all the points, both new and old. (See the shaded blue rectangle in Fig. 1(d).) Depending on whether it is wider or taller, we split based on the x- or the y-axis. (In this case the rectangle is taller, so the cutting dimension is set to y (1)).

We sort the points along this dimension and select the median coordinate. (In this case it is midway between y-coordinates of `ORD` and `DFW`, which is $y = 7$.) We create an internal node having this cutting dimension and cutting value, and we partition the points to its left and right subtrees. (In this case, the sublist `[ORD, ATL]` is sent to the left subtree and `DFW, SFO]` is sent to the right. Note that in this case, both lists with within a single bucket, so we create a single external node for each and we are done.)

CMSC 420: Fall 2022

Programming Assignment 3: Capacitated Facility Location

Background: In this assignment, we will ask you to combine the various data structures we have implemented this semester to solve an important optimization problem from the field of operations research. This falls within a broad category of optimization problems known as *facility location*. To motivate the problem, imagine that you are in charge of deciding where to put a set of *service centers* for an organization. For example, this might be the locations of cell towers in city, the locations of retail outlets like coffee shops, or the locations of neighborhood facilities like post offices and schools. You want to distribute your service centers close to where your customers/clients are located, but there is a maximum limit, or *capacity*, to the number of clients a given center can serve. It is customary to refer to the client locations as *demand points*, since we think of each of them as demanding service which we need to provide. We don't want our customers to travel too far, so we wish to keep the travel distance from each demand point to its assigned service center to be small.

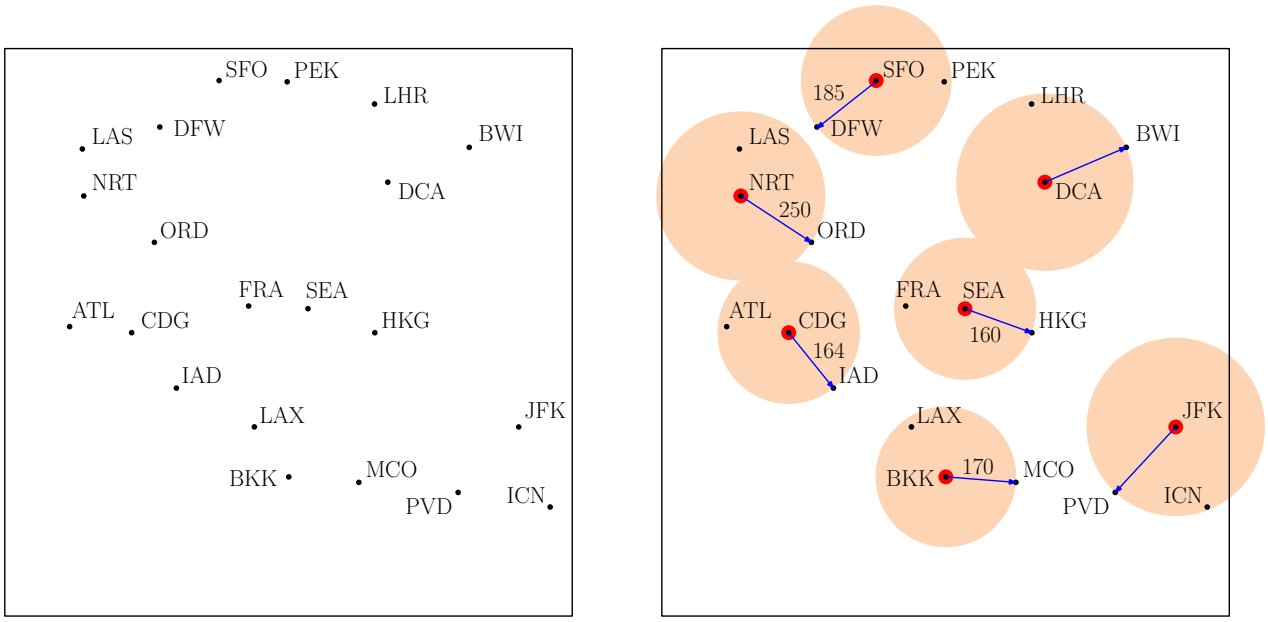


Figure 1: (a) A set of 21 demand points and (b) a possible solution to the discrete k-capacitated facility location problem, for $k = 3$. (This test case can be found in `test04-input.txt`)

Discrete Capacitated Facility Location: This problem can be modeled mathematically as follows. Let $P = \{p_1, \ldots, p_n\}$ denote the set of *demand points* in $\mathbb{R}^2$ (see Fig. 1(a)), and the integer k denote the maximum capacity of any center. The objective is to determine the locations of a set of *service centers* $C = \{c_1, c_2, \ldots\}$, and an assignment of each demand point

to a one of these centers, so that each center is assigned to serve at most k demand points (see Fig. 1(b)). For each center c_j, define *service radius* r_j to be the maximum distance to any of the demand points assigned to it. Define the *cost* of the solution, denoted $\text{cost}(C) = \sum_j r_j$. The objective is to compute a set of centers satisfying the capacity constraints and minimizing the cost. This is called the *discrete k-capacitated facility location problem.*

Unfortunately, there is a trivial solution to the problem, namely to make *every* point of P a service center. To prevent this, let us add the additional constraint that every service center must be assigned to *exactly* k demand points (including itself). As a consequence, we will need to add the condition that the capacity k evenly divides the total number of points.

Greedy Heuristic: This problem is NP-hard, so we are not aiming to find an optimal solution. Instead, we will implement a common heuristic solution, called the *greedy solution.* This is computed as follows:

- Initialize the set of service centers to the empty set, that is, $C \leftarrow \emptyset$.
- While $P \neq \emptyset$, repeat the following steps:
  - For each $p_i \in P$ compute its k-nearest neighbors (including itself) from P. Let r_i denote the distance from p_i to its kth nearest neighbor, called its *radius.*
  - Find the point p_i that has the smallest radius value r_i. Add p_i to C. Remove p_i and its $k-1$ other nearest neighbors from P.
- Return C and the associated assignments as the final result.

Implementing the Greedy Heuristic: Unfortunately, a direct implementation of the above algorithm will not be very efficient. First off, computing nearest neighbors without the aid of a data structure is very slow. Also, with each iteration, we delete k points from P, which will affect the nearest-neighbor radii of the other points.

Instead, we will use the data structures we have implemented this semester to help us out. First, we will store the points in a spatial index (the extended kd-tree from Project 2) so that k-nearest-neighbor queries can be answered efficiently. Second, we will store service centers c_j and the associated radius values r_j in a priority queue (the leftist heap from Project 1). To extract the next center, we extract the minimum key from the priority queue.

We still have the problem that whenever we delete a point from P, it might affect the result of an earlier k-nearest-neighbor queries. To handle this, we will store not only the service center and radius, but all c_j's k-nearest-neighbors in the priority queue. When we extract a candidate cluster center, we will search the tree to see that all the points assigned to it still exist. If so, this cluster of points is valid, and add c_j to C and delete all the points in its cluster. If not, we recompute c_j's k-nearest neighbors, and recompute its new radius r_j, and put this entry back in the priority queue. The algorithm is outlined below. (Also see the description of `extractCluster` below.)

- Initialize the set of service centers to the empty set, that is, $C \leftarrow \emptyset$. Create a new (extended) kd-tree and add all the points of P in bulk to this tree. Create an new (leftist) priority queue.

- For each point $p_i \in P$, use the kd-tree to compute its k-nearest neighbors (including itself). Let L_i be a list storing these k points, and let r_i denote the distance to the farthest of these points. Store the key-value pair (r_i, L_i) in the priority queue.
- While $P \neq \emptyset$, repeat the following steps:
  - Use the extract-min operation to remove the pair (r_i, L_i) from the queue having the smallest radius. Let c_i denote the associated cluster center. (Note: In our implementation, we will assume that the elements of L_i are listed in increasing order of distance from the center point. This means that c_i will always be the first element in the list.)
  - For each $q_i \in L_i$, perform a find operation on the kd-tree to see whether q_i is still in the tree. If not, break out of the loop in failure.
  - If we exited the loop with success, delete all the points of L_i from the kd-tree. Add c_i to our list of service centers and the points of L_i (including c_i itself) are the demand points assigned to it.
  - If we exited the loop with failure, there are two cases. If c_i is still in the kd-tree, we perform a k-nearest neighbor search to compute its new list L_i' of nearest neighbors. Let r_i' be the distance to the farthest point in the new list. (Note: Since we assume that L_i' is sorted by distance, r_i' will be the distance from c_i to the last point in the list.) Insert the pair (r_i', L_i') into the priority queue. Otherwise, (if c_i is not in the kd-tree) do nothing. In either case, we stay in the loop until we find a valid cluster.
- Return C and the associated assignments as the final result.

In order to implement the above algorithm, you will need to make two enhancements to the kd-tree from Project 2. First, you will need to implement a delete operation, and second you will need to implement a k-nearest-neighbor operation. We will discuss these in further detail below.

Deletion and k-Nearest Neighbors

What are the main changes to your kd-tree implementation from Project 2? These are the operations of deletion and k-nearest neighbors (k-NN)queries. Along the way we will implement a useful utility data structure.

`void delete(Point2D pt) throws Exception`: This deletes the point `pt` from the tree. It begins by searching for the point. If it is not found, it throws an `Exception` with the error message `"Deletion of nonexistent point"`. Otherwise, it removes this point from the external-node containing it. (Note that this is very different from the deletion operation for standard kd-trees. We do not need to find a replacement node, because all deletions occur from the leaf level.)

If the deletion results in an empty bucket, we may need to take additional action. If the external node is the root of the tree, we just leave the external node empty. (This only happens when we delete the last point, so the tree is now empty.) Otherwise, the tree needs to be modified to avoid having an empty bucket. Let `q` be the external node where

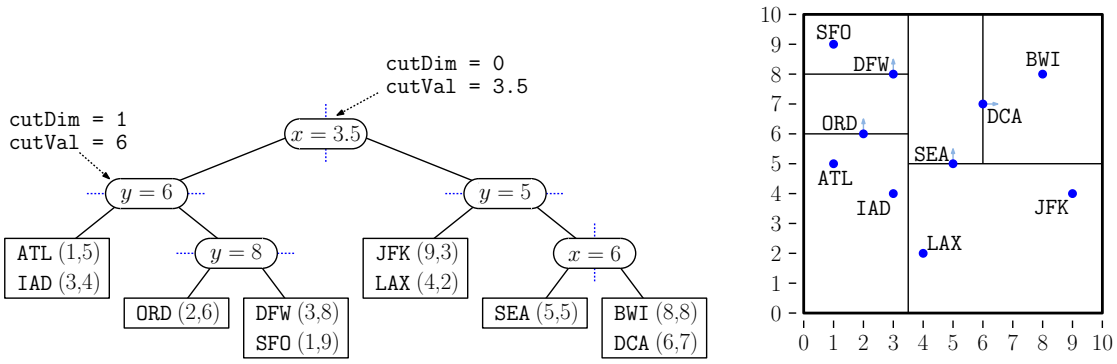


Figure 2: An example of an extended kd-tree with bucket size 2 and bounding box $[0, 10] \times [0, 10]$.

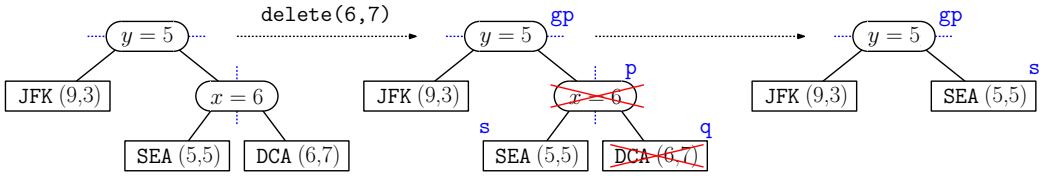


Figure 3: Deleting $(6, 7)$ results in an empty bucket. We remove this external node `q` and its parent `p`, replacing them with the sibling node `s`.

the deletion took place, let `p` be its parent, let `gp` be `q`'s grandparent, and let `s` be `q`'s sibling (see Fig. 3). We unlink both `q` and `p` from the tree, and `s` replaces `p` as the child of `gp`. Note that if `p` is the root of the tree, `s` becomes the new root node. In Fig. 4 we present an example of a series of deletions resulting eventually in an empty tree.

`ArrayList<LPoint> kNearestNeighbor(Point2D q, int k)`: Computes the k points that are closest to the query point `q`. The result is a Java `ArrayList` of `LPoint`. If the tree is empty, it returns an empty `ArrayList`. If k exceeds the number of points in the tree, the list contains only as many points as there are in the tree. The list should be sorted in increasing order of (squared) distance from the query point (see Fig. 5).

How to implement `kNearestNeighbor`? The k-NN algorithm is quite similar to that of the single nearest-neighbor. In the that algorithm, we maintained a variable `best`, which stored the closest point seen so far in the search. To generalize this for k-nearest neighbors, we will maintain the k closest points seen so far. To do this, you will implement a useful utlity data structure, called `MinK`. Each time we encounter a new point, we add it to the `MinK` structure, but no matter how many entries are inserted, this structure *only retains the smallest k items.* In our case, elements in `MinK` will be sorted by their squared distance from the query point, so we only save the k closest points.

Given this data structure, we can answer k-nearest neighbor queries as follows. First, we get the kth (that is, maximum) entry from the `MinK` structure (using an operation `getKth` described below). If the cell is farther away relative to `q`, we know that nothing in this subtree can provide a better point than one of the closest k, so we skip this node. Otherwise, we add a key-value pair consisting of this node's point and its distance from `q` to the `MinK` structure.

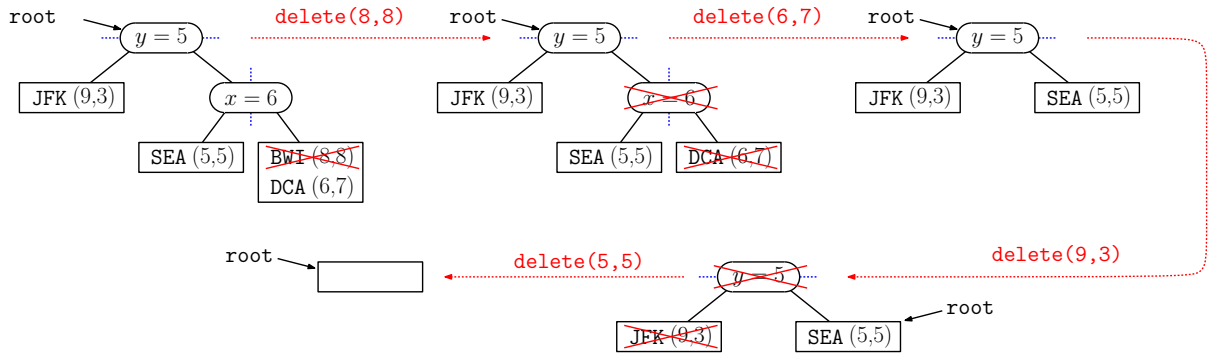


Figure 4: The operation `delete(8,8)` removes `BWI` from its bucket. Next, `delete(6,7)` removes `DCA` from its bucket, resulting in an empty bucket, causing `SEA` to replace its parent. Similarly, `delete(9,3)` removes both `JFK` and its parent, making `SEA` the new root. Finally, deleting `(5,5)` results in an empty tree with a single empty external node.

(Remember, this only saves the k closest points.) We then recursively visit the two children, giving priority to the one that is closer to `q`. The code block below shows how to do this for a standard kd-tree. Note that, unlike the code given in class, there is no need to return `minK`, because it is passed as a reference parameter. You will need to convert this to work with your extended kd-tree.

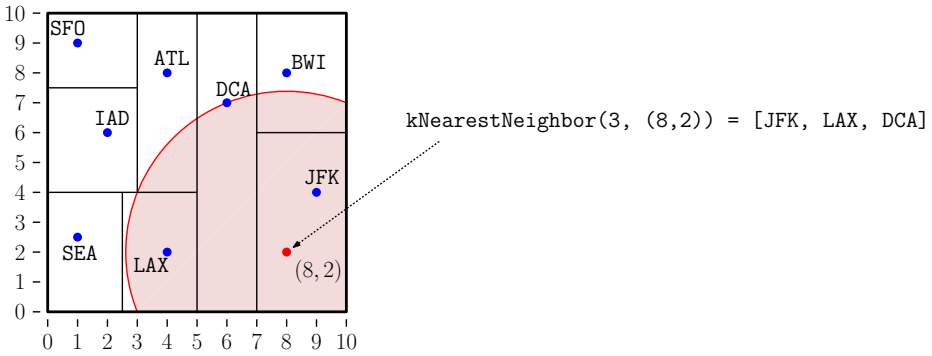


Figure 5: k-NN query for $k = 3$ and $q = (8, 2)$ returns the sorted point list `[JFK, LAX, DCA]`.

The `MinK` data structure: All that remains is to explain how `MinK` works. It is defined generically, based on the key and value pairs it stores. Here is the declaration:

```
public class MinK<Key extends Comparable<Key>, Value> { ... }
```

When we use it for k-NN queries, the keys will be squared distances to the query point (of type `Double`) and the values will be points of type `LPoint`. It has the following public members:

`MinK(int k, Key maxKey)`: This is the constructor, which is given the number of items `k` to maintain and the maximum possible key value. For our puroposes, the value `k` will just be the value k in the k-NN query, and since we are sorting by squared distances,

Helper for k-NN search in a standard kd-tree

```
void kNNHelper(Point2D q, KDNode p, Rect2D cell, MinK minK) {
    if (p == null) return                              // fell out of tree?
    if (cell.distTo(q) > minK.getKth()) return         // cell is too far away?
    minK.add(p.point.distTo(q), p.point)               // add this point
    int cd = p.cutDim                                  // cutting dimension
    Rectangle leftCell = cell.leftPart(cd, p.point)    // get child cells
    Rectangle rightCell = cell.rightPart(cd, p.point)

    if (q[cd] < p.point[cd]) {                         // q is closer to left?
        kNNHelper(q, p.left,  leftCell,  minK)
        kNNHelper(q, p.right, rightCell, minK)
    } else {                                           // q is closer to right?
        kNNHelper(q, p.right, rightCell, minK)
        kNNHelper(q, p.left,  leftCell,  minK)
    }
}
```

the value of `maxKey` can be set to `Double.POSITIVE_INFINITY`. (Save this value, since it will be needed for `getKth` below.)

`int size()`: This returns the current number of elements in the structure. Initially the size is zero, and as elements are added the size increases up to a maximum of k.

`void clear()`: This removes all entries, reseting the structure to its initial empty state.

`Key getKth()`: If the structure has k elements, this returns the maximum key value among these elements. Otherwise, it returns `maxKey` value given in the constructor. (Why `maxKey` and not `null`? The reason is that this is what works most conveniently for the k-NN helper. If we have not yet seen k points, we will never decline the opportunity to visit a node.)

`ArrayList<Value> list()`: Create a `ArrayList` of the values in the structure, sorted in increasing order by their key values. (Note: We will not modify the contents of the values stored in the returned list, so it is not necessary to perform a "deep copy" of the values. You can just copy the references into the resulting `ArrayList`.)

`void add(Key x, Value v)`: This adds the given key-value pair to the current set. If the structure has fewer than k element, this entry is added, increasing the size by one. If the structure has k elements and x is greater than or equal to the largest, the operation is ignored. If the structure has k elements and x is less than the largest, the pair (x, v) is added, and the previous largest is removed. Thus, the size of the structure remains k.

An example is shown in Fig. 6.

How to implement `MinK`? You have two choices on how to implement the `MinK` structure:

Simple and Slow: (Efficiency penalty of 10 points.) Just store the key-value pairs in an array (or `ArrayList`) sorted by key values (similar to Fig. 6). When a new entry is added, simulate one step of insertion-sort, by sliding all the larger elements down one position until finding the slot where the new item fits. If there were already k elements, the largest element just falls off the end of the array. (This takes $O(k)$ time per insertion.)

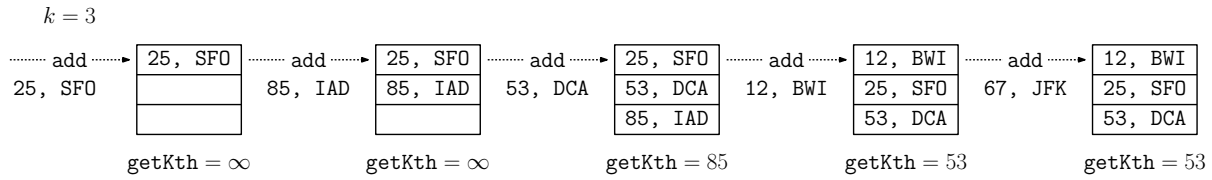


Figure 6: Example of `MinK` operations for $k = 3$ and `maxKey` $= \infty$.

Smart and Speedy: The second method is more efficient, and you will get full credit. Maintain a standard binary heap, as used in heap-sort. (Recall Lecture 5 on Heaps). It will be max-heap ordered with the largest element stored at the root. When a new entry is added, there are two cases. First, if there are currently fewer than k elements in the heap, add this element to the next available position in the heap array and sift it up to its proper location. Otherwise, if there are k elements in the heap, compare the newly added element to the root. (The root has the largest entry.) If it is greater than or equal, ignore the insertion. If it is less than the root's key, replace the root with the new item, and sift it down to its proper location. (This takes $O(\log k)$ time per insertion.) Note that the operation `list` will need to copy the contents of structure to a new `ArrayList` and then sort them by key. If you like, you may do this by implementing heap-sort. But it is fine for full credit to simply invoke `Collections.sort`.)

Back to Facility Location

In order to test your facility location algorithm, you will implement a class called `KCapFL`, for *k-capacitated facility locator*. It will implement the greedy facility location algorithm, and we will add a few additional functions for grading purposes to allow us to inspect how your function works. This class is parameterized by the point type, which like our kd-tree, will just be labeled points. The declaration is `public class KCapFL<LPoint extends LabeledPoint2D>`. It will store the following pieces of private data:

`int capacity`: This is the maximum capacity of any service center, which we have called k up to now

`XkdTree<LPoint> kdTree`: A kd-tree for storing the points

`LeftistHeap<Double, ArrayList<LPoint>> heap`: A leftist heap for storing key-value pairs. Each pair is of the form (r_i, L_i), where r_i is the squared radius of this cluster of points and L_i is the set of points in the cluster. The heap ordered by the (squared) service radii, that is, the squared distance to the kth nearest neighbor. (As with the programming assignment, this is min-heap ordered, with the smallest radius at the root.)

It supports the following public functions:

`KCapFL(int capacity, int bucketSize, Rectangle2D bbox)`: This is the constructor. It sets the capacity, creates a new kd-tree with the given bucket size and bounding box, and creates a new leftist heap.

`void clear()`: This clears the data structure by clearing both the kd-tree and the leftist heap.

`void build(ArrayList<LPoint> pts) throws Exception`: This initializes the structure. First, it checks whether the number of points is strictly greater than zero and is evenly divisible by the capacity. If not, it throws an `Exception` with the error message `"Invalid point set size"`. Otherwise, it performs a bulk-insertion of the points into your kd-tree. Finally, it creates the initial radii. Do do this, it enumerates all the points. For each point p_i it computes its k-nearest neighbors using the kd-tree. et L_i be the `ArrayList` of labeled points returned by the k-nearest neighbor procedure. Let c_i be the center point (the first point in L_i). Let r_i be the squared distance to the farthest point (the kth point of L_i). Insert the key-value pair (r_i, L_i) in your leftist heap. (There is not need to perform a deep copy. Just copy the reference to L_i.)

`ArrayList<LPoint> extractCluster()`: This performs one step of the greedy algorithm. First, if the kd-tree is empty, return `null` as a signal that there are no more clusters. Otherwise, repeat the following steps until we are successful in finding a cluster. Extract the next cluster (r_i, L_i) from your priority queue (the leftist heap). Using the kd-tree `find` operation, check whether every point of L_i is still in the kd-tree. If so, we have successfully found a cluster, and otherwise we haven't. Here is how to process each of these cases:

Success: Delete all the points of L_i from the kd-tree, and return L_i as the answer.

Failure: Let c_i be the first point of L_i (the service center). If c_i is still in the kd-tree, compute a new list L_i' of its k-nearest neighbors, let r_i' be its new radius, and add the pair (r_i', L_i') back into the leftist heap. (Note that c_i will still be the first element of L_i', so we have effectively replaced an old damaged cluster for c_i with a new one.) Otherwise, (c_i is not in the kd-tree) do nothing. In either case, continue extracting clusters from the priority queue until we succeed.

Note that the leftist heap will throw an `Exception` if you attempt to extract when there are no more elements left. In theory, this should not happen, since if you still have points in your kd-tree, you should still have clusters in your leftist heap. Nonetheless, you will need to create a `try-catch` block to keep the compiler happy, but if you ever reach the `catch` section, there is something wrong in your program.

`ArrayList<String> listKdTree()`: This just invokes the `list` operation on your kd-tree tree (for debugging).

`ArrayList<String> listHeap()`: This just invokes the `list` operation on your leftist heap tree (for debugging).

More Information

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You will need to fill in the implementation of the `KCapFL.java`, `XkdTree.java`, `LeftistHeap.java`, and `MinK.java`. As before, we will provide `Point2D`, `Rectangle2D`), and so on. We will also provide the testing programs, `Part3Tester.java` and `Part3CommandHandler.java`. As with the previous assignment, the package "`cmsc420_f22`" is required for all your source files.

What if I cannot finish? Give priority to working on point deletion and k-nearest neighbors. If you do just those, you will get roughly 50% credit. The remainder will be for the `KCapFL` functionality and the usual efficiency/style points. But, don't be intimidated by the lengthy description. The lengthy description above is almost a line-for-line presentation of the `KCapFL` class. My class was only about 1/5 as long as my leftist heap and kd-tree implementations.

What if my kd-tree/leftist heap didn't work? We'll make minimal versions of these programs available to you. You will still need to implement deletion and k-nearest neighbors.

Efficiency requirements: (20% of the final grade) 10% for efficiently implementing k-NN and 10% for efficiently implementing `MinK`.

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. 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 1, on the class Project Page.)

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. You need to submit `KCapFL.java`, `XkdTree.java`, `LeftistHeap.java`, and `MinK.java` files. If you created any additional files for utility objects, you will need to upload those as well.

CMSC 420: Fall 2022

Homework 1: Trees and More

Problem 1. (15 points) Answer the following questions involving the rooted trees shown in Fig. 1.

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

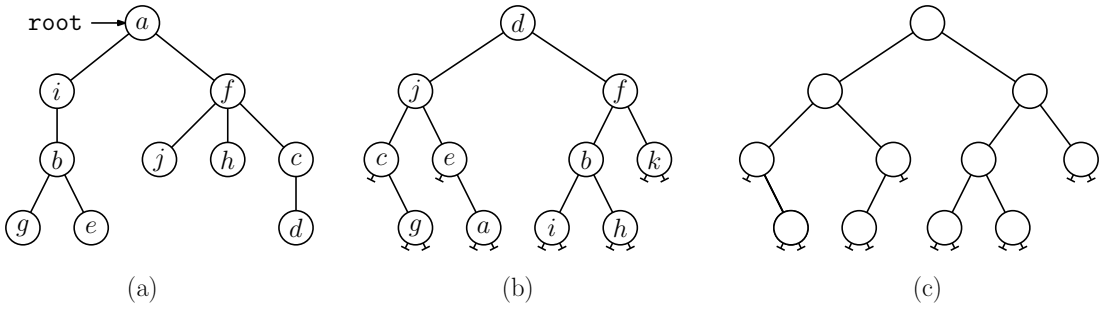


Figure 1: Rooted trees.

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

(c) (2 points) Repeat (b) but for *inorder*.

(d) (2 points) Repeat (b) but for *postorder*.

(e) (3 points) Draw a figure showing the tree of Fig. 1(c) with inorder threads. (As an example, see Fig. 7 from the Lecture 3 notes. Be sure to include any `null` threads.)

(f) (3 points) Draw a figure showing the tree of Fig. 1(c) where each node is labeled with its null path length value. (As an example, see Fig. 5(a) from the Lecture 5 notes).

Problem 2. (5 points) An alternative to using rank in the disjoint-set union-find data structure is to use the size of the tree. Suppose that we modify the union algorithm as follows. When we union two trees T' and T'' having n' and n'' nodes respectively, if $n' \leq n''$, then we link T' as a child of T'', and otherwise we link T'' as a child of T' (see Fig. 2).

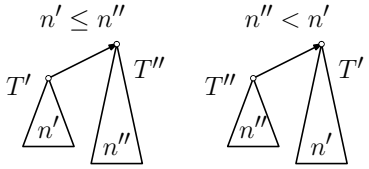


Figure 2: Size-based union.

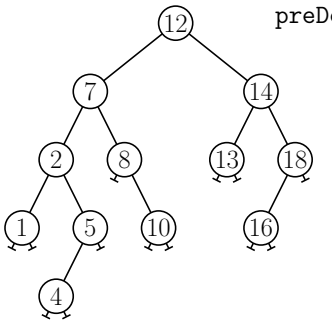
Prove the following lemma, which shows that this also yields height-balanced trees.

Lemma: If the above size-based merging process is used, a tree of height h has at least 2^h elements.

Give a formal proof of this lemma. (Hint: The proof is similar to the one given in the lecture notes.)

Problem 3. (12 points) You are given a binary search tree as given in the class `BinarySearchTree` from Lecture 6 (page 9). (In this problem, we will not be using the `value` fields of the nodes.) In answering the following you may add additional utility functions to this class, but you should not modify the node class, `BSTNode`, nor add additional data to the `BinarySearchTree` class. In addition to your pseudocode, provide a short description of how your function works.

(a) (4 points) Present pseudocode for a member function `preDepth()`, which traverses the nodes in preorder and prints each node's key and its depth (see Fig. 3). (Hint: I would suggest creating a recursive helper function `preDepthHelper(BSTNode u, ...)`, which is given a specific node `u` and, optionally, additional parameters of your choosing. The initial call is `preDepthHelper(root, ...)`. You may create additional helper functions if you like.)




```
preDepth: 12, 0   postNPL: 1, 0   postStats: 1, 3, 0
          7, 1              4, 0              4, 4, 0
          2, 2              5, 0              5, 3, 1
          1, 3              2, 1              2, 2, 2
          5, 3              10, 0             10, 3, 0
          4, 4              8, 0              8, 2, 1
          8, 2              7, 1              7, 1, 3
          10, 3             13, 0             13, 2, 0
          14, 1             16, 0             16, 3, 0
          13, 2             18, 0             18, 2, 1
          18, 2             14, 1             14, 1, 2
          16, 3             12, 2             12, 0, 4
```

Figure 3: Depth and null path lengths in postorder.

(b) (4 points) Present pseudocode for a member function `postNPL()`, which traverses the nodes in postorder and prints each node's key and its null path length (see Fig. 3). (Hint: As in part (a), create a recursive helper function.)

(c) (4 points) Present pseudocode for a member function `postStats()`, which traverses the nodes in postorder and prints each node's key, its depth, and its height (see Fig. 3).

Problem 4. (10 points) You have just invented a new data structure, called a *dual stack*, which stores two stacks in a single array. It works as follows. Given an array `A` of length m, one of the stacks starts at index 0 and grows upwards and the other starts at index $m-1$ and grows downwards (see Fig. 4). More formally, there are two stack tops `top1` and `top2`. Initially, `top1 = -1` and `top2 = m`. Assuming that `top1` $<$ `top2`, when an object is pushed on the first stack, we store it in `A[++top1]`. When an object is pushed on the second stack, we store it in `A[--top2]`. Each operation costs $+1$ unit.

If a push occurs where `top1` $=$ `top2`, we need to expand the arrays (see Fig. 4). Let $n_1 =$ `top1` $+1$ denote the number of elements in stack 1, and let $n_2 = m -$ `top2` denote the number of elements in the second stack. We allocate a new array of size $m' = 3\max(n_1, n_2)$, and then we copy the elements from the current array to the new array. (The stack-2 elements are copied to the top of the new array.) The actual cost of the expansion is equal to the total

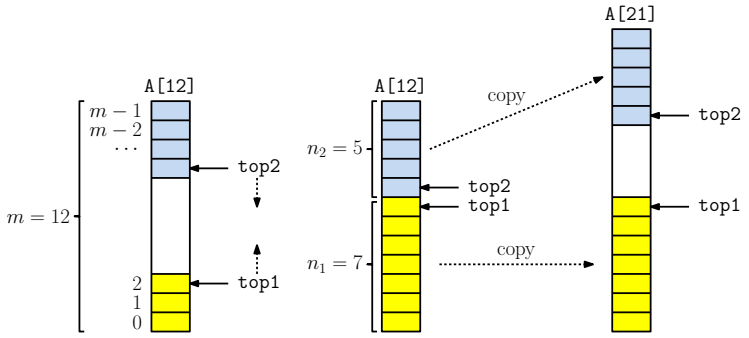


Figure 4: Expanding dual stack.

number of elements copied, that is, $n_1 + n_2$. After the expansion is finished, we have sufficient space to insert the new element, and we perform the push, which costs +1 unit of work.

For example, in Fig. 4, the current array has 12 elements, and when $n_1 = 7$ and $n_2 = 5$ and someone attempts to push an additional element on one of the two stacks we allocate a new array of size $m' = 3\max(n_1, n_2) = 21$ and we copy the elements of the two arrays into the new array. The cost of the expansion is $n_1 + n_2 = 12$ plus an addition +1 unit for the actual push.

In this problem, we will prove that the dual stack has constant amortized cost. Initially, the array has space for two entries ($m = 2$), and both stacks are empty.

(a) (4 points) Suppose that we have just performed a reallocation. Our current array of size $n_1 + n_2 = m$, and our new array is of size $m' = 3\max(n_1, n_2)$. Explain why at least $m'/3$ pushes can be performed until the new array needs to expand again. (Hint: The actual number depends on the relative values of n_1 and n_2. What is the worst case?)

(b) (2 points) Explain why the cost of the next expansion is m'.

(c) (4 points) Using parts (a) and (b) and the fact that cheap stack operations (which do not cause an expansion) cost +1 unit, derive the smallest constant c such that the amortized cost of our expanding dual stack is at most c and prove the correctness of your answer.

Problem 5. (8 points) Consider the two leftist heaps with roots u and v shown in Fig. 5. In this problem, you will trace the execution of the merge function `merge(u, v)` on this input.

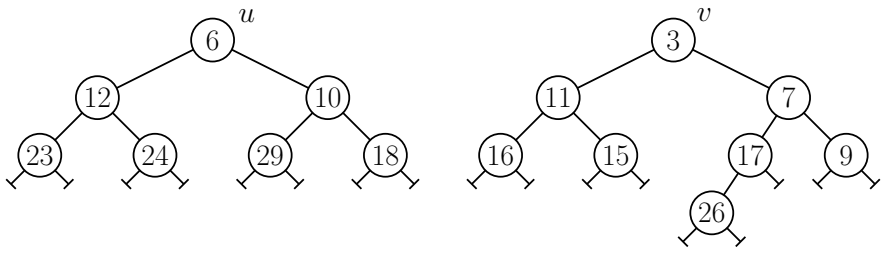


Figure 5: Leftist heap merge.

(a) (4 points) List all the recursive calls made in the order in which they are made. (Use the key value in each node to identify it. For example, the first recursive call made is `merge(6,3)`. You should follow the code rigorously. Do not rely on the intuitive "two-phase" approach used to illustrate the algorithm.)

(b) (4 points) Draw the final merged tree structure (after all the recursive calls terminate) and indicate the NPL values for each node.

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: In class, we showed that it is possible to merge two leftist heaps of sizes n' and n'' in time $O(\log n)$, where $n = n' + n''$. We never explained how to perform the operations insert and extract-min, however. Present pseudocode to implement these two operations in $O(\log n)$ time. (Hint: Use the merge helper function.)

Challenge Problem 2: In class, we showed that the rightmost path in any leftist heap has length $O(\log n)$. The analysis presented in the Lecture 5 notes (page 6) shows that if a leftist heap has n nodes, it has at most $\lg(n+1)$ nodes along its rightmost path. Prove that it is generally the case that in any binary tree with $n \geq 1$ nodes, there exists a path from the root to a node with a null child, such that this path has at most $\lg(n+1)$ nodes.

CMSC 420: Fall 2022

Homework 2: Search Trees

Problem 1. (10 points) Consider the AVL trees shown in Fig. 1.

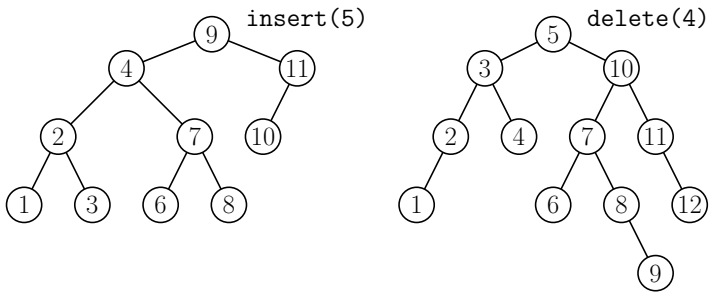


Figure 1: AVL-tree operations.

(a) (5 points) Show the result of executing the operation `insert(5)` to the tree on the left.

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

In each case, show the final tree and list (in order) all the rebalancing operations performed (e.g., "`rotateLeftRight(7)`"). Intermediate results may be shown for the sake of assigning partial credit. Draw the final tree as in Fig. 1(b) from Lecture Lecture 7. Show the balance factors at each node. (Don't bother to give the heights.)

Problem 2. (10 points) Consider the AA trees shown in Fig. 2.

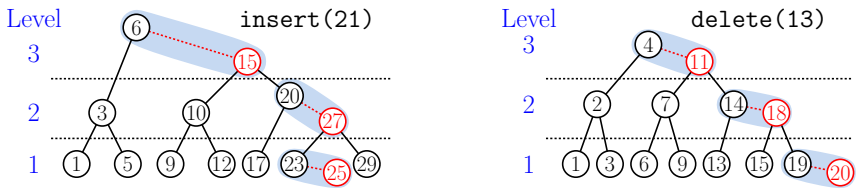


Figure 2: AA-tree operations.

(a) (5 points) Show the result of executing the operation `insert(21)` to the tree on the left.

(b) (5 points) Show the result of executing the operation `delete(13)` 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. (10 points) Recall from Lecture 9 the classical red-black tree (not the AA tree) effectively models a generalization of the 2-3 tree, called the *2-3-4-tree*. Recall that this is a straightforward generalization of 2-3 trees, but nodes may have 2, 3, or 4 children (and 1, 2, or 3 keys, respectively).

(a) (5 points) Fig. 3 shows a (classical) red-black tree. Draw the associated 2-3-4 tree. (Draw your tree using the same format as in Fig. 1(c) from Lecture 8.)

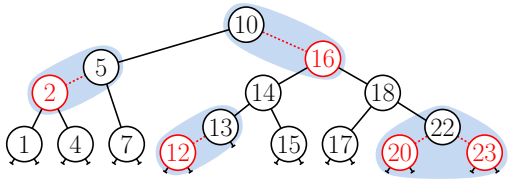


Figure 3: Converting a red-black tree to a 2-3-4 tree.

(b) (5 points) Explain what sequence of AA rebalancing operations (skews and splits) would be needed to convert this tree into an AA tree (e.g., "`skew(13), split(18), ...`"). Draw the final AA tree.

Problem 4. (10 points) Consider the splay trees shown in Fig. 4. In both cases, apply the exact algorithms described in the Lecture 12 notes.

(a) (5 points) Show the steps involved in operation `insert(9)` for the tree in Fig. 4 (left).

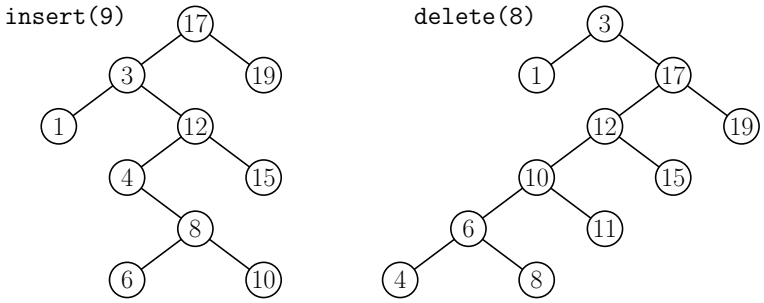


Figure 4: Splay-tree operations.

(b) (5 points) Show the steps involved in operation `delete(8)` for the tree in Fig. 4 (right).

In both cases, indicate what splays are performed and what additional alterations are performed beyond splaying. (When doing splaying, we only need to see the tree after splaying is complete, but intermediate results may be shown for partial credit). Also show the final tree after the insert/delete operation is finished.

Problem 5. (10 points) Can you determine the structure of a binary tree based solely on a preorder enumeration of its nodes? In general, the answer is no, since there can be multiple trees that have the same preorder listing of nodes. In this problem, we will see that there are instances where this is possible.

Recall that a binary tree is said to be *full* if every node has either two (non-null) children or no children at all (both are null). An example is shown in Fig. 5.

Suppose that you have a full tree with n nodes, which have been enumerated according to a preorder traversal and stored in an array called `preorder[n]`. Suppose as well that you have a parallel boolean array `isLeaf[n]`, which tells you which nodes are leaves and which are not. In particular, `isLeaf[i]` is true if `preorder[i]` is a leaf node and false otherwise.

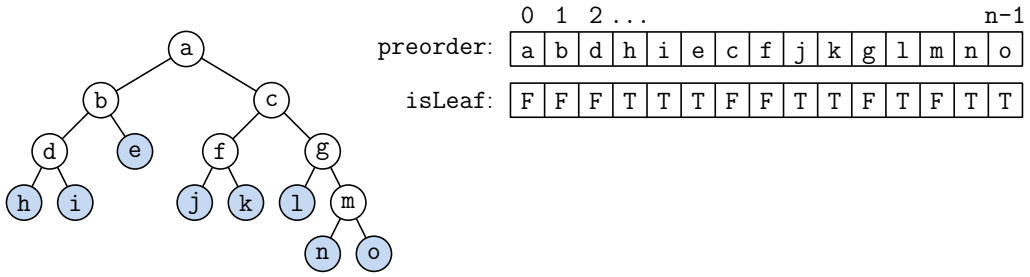


Figure 5: A full binary tree and the arrays `preorder` and `isLeaf`.

(a) (5 points) Suppose that the arrays `preorder` and `isLeaf` have been generated from some full binary tree T. Prove that T is uniquely determined from the contents of these two arrays. (Hint: Use strong induction on the number of nodes in the tree.)

(b) (5 points) Present pseudocode for a function, which given two valid arrays, `preorder` and `isLeaf`, constructs the unique tree that they represent. You may assume that the arrays are valid in the sense that they define an actual full binary tree.

Hint: Write a recursive function `buildTree(int i)` which is given an index `i`, constructs the subtree rooted at node `preorder[i]`, and returns a reference to the root of this subtree. It may be helpful to assume that `i` is passed in as a reference parameter, and as the procedure runs, it advances `i` to the first node following the generated subtree. What is the initial call to this function?

Challenge Problem: Recall that each node of an AA tree stores a key, its level, and pointer to its left and right children. (For this problem, we will ignore the node values. Also, rather than using the sentinel node nil, let's assume that we just use standard null pointers in the leaves.)

Suppose that you have been given a valid AA tree, but all the level information is missing! Knowing that the tree is a valid AA tree, is it possible to reconstruct the level information for the tree? Take a position and justify it. If you believe that it is not possible, give an example of a binary tree which can be interpreted as two different valid AA trees (where at least one node has a different level number in the two trees). If you believe it is possible, present a proof that the assignment of level numbers is unique. (Your proof might take the form of a procedure that reconstructs the level information for the AA tree.)

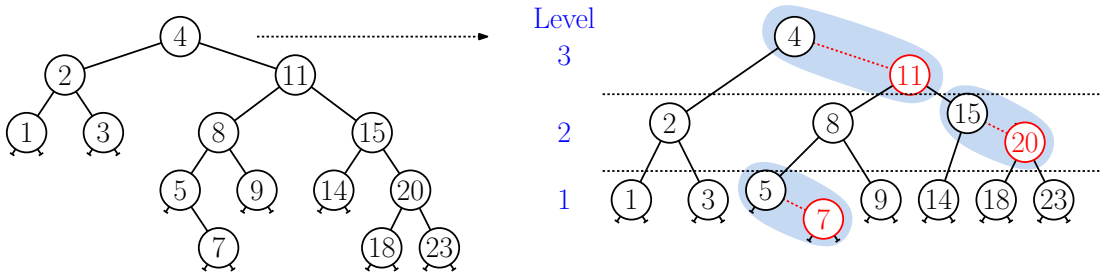


Figure 6: Can you recover the level numbers?

CMSC 420: Fall 2022

Practice Problems for Midterm 1

Problem 0. Expect at least one question of the form "apply operation X to data structure Y," where X is a data structure that has been presented in lecture. (Likely targets: Union-Find, leftist heaps, AVL trees, 2-3 trees, AA trees, treaps, skiplists, and splay trees).

Hint: Intermediate results can be included for partial credit, but don't waste too much time showing intermediate results, since they can steal time from later problems.

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

(a) 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 a standard (unbalanced) binary search tree storing the consecutive odd keys $\{1, 3, 5, 7, 9, 11, 13\}$ (which may have been inserted in any order). Into this tree you insert the consecutive keys $\{0, 2, 4, 6, 8, 10, 12, 14\}$ (also inserted in any order). Which of the following statements hold for the resulting tree. (Select all that apply.)

(i) It is definitely a full binary tree

(ii) It is definitely a complete binary tree

(iii) Its height is larger than the original by exactly 1

(iv) Its height is larger than the original, but the amount of increase need not be 1

(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. (No justification needed.)

(d) 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).)

(e) In a leftist heap containing $n \geq 1$ elements, what is the minimum possible NPL value of the root? What is the maximum? (It is okay to be off by an additive error of $\pm O(1)$.)

(f) What are the minimum and maximum number of levels in a 2-3 tree with n nodes. (Define the number of levels to be the height of the tree plus one.) Hint: It may help to recall the formula for the geometric series: $\sum_{i=0}^{m-1} c^i = (c^m - 1)/(c - 1)$.

(g) 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? Briefly explain.

(h) Unbalanced search trees, treaps and skiplists all support dictionary operations in $O(\log n)$ "expected time." What difference is there (if any) in the meaning of "expected time" in these contexts?

(i) 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? (Briefly explain)

(j) 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. 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).

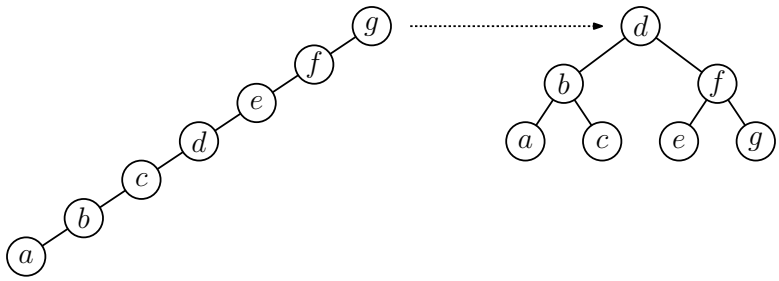


Figure 1: Rotating into balanced form.

(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.)

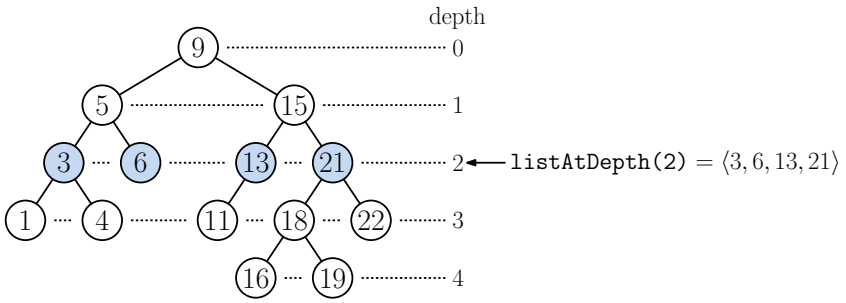


Figure 2: Nodes at some depth.

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

Problem 6. 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.

Problem 7. 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. 3, 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.

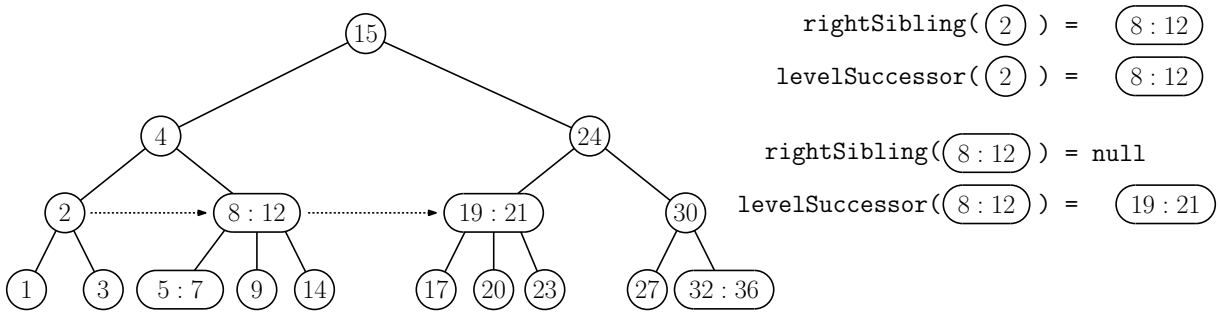


Figure 3: Sibling and level successor in a 2-3 tree.

(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. 3, 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 8. 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. 4). 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.

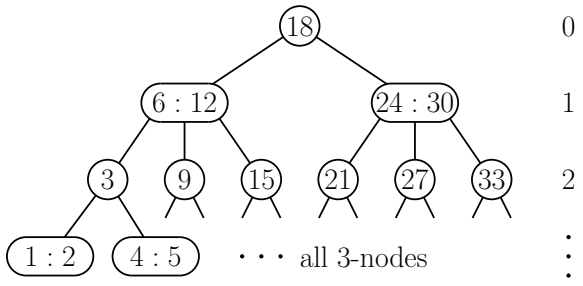


Figure 4: Alternating 2-3 tree.

(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

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

(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 9. 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.

Problem 10. 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 11. In this problem we will 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[]`, which contains as many elements as `p.next[]`. This array is defined as follows. If `p.next[i]` refers to a node `q`, then `p.span[i]` contains the distance (number of nodes) from `p` to `q` (at level 0) of the skip list.

For example, see Fig. 5. Suppose that `p` is third node in the skip list (key value "10"), and `p.next[1]` points to the fifth element of the list (key value "13"), then `p.span[1]` would be $5 - 3 = 2$, as indicated on the edge between these nodes.

Present pseudo-code for a function `int countSmaller(Key x)`, which returns a count of the number of nodes in the entire skip list whose key values are strictly smaller than `x`. For example, in Fig. 5, the call `countSmaller(22)` would return 6, since there are six items that are smaller than 22 (namely, 2, 8, 10, 11, 13, and 19).

Your procedure should run in time expected-case time $O(\log n)$ (over all random choices). Briefly explain how your function works.

Problem 12. 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:

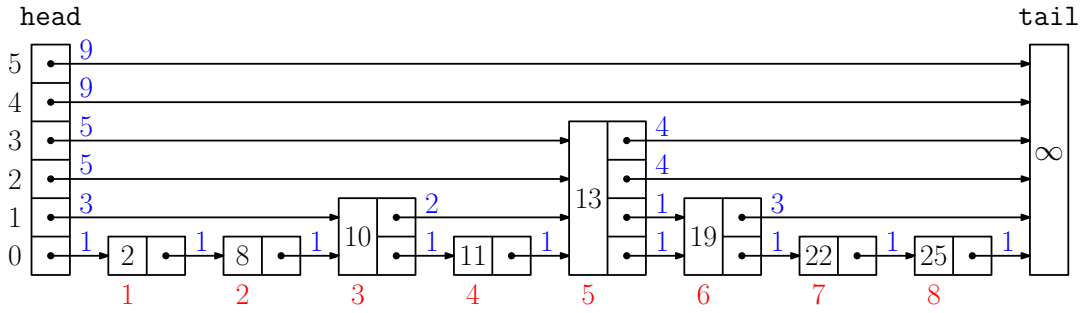


Figure 5: Skip list with span counts (labeled on each edge in blue).

- 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.

CMSC 420 (0201) - Midterm Exam 1

Problem 1. (10 points)

(a) (5 points) Show the final tree that results from performing `splay(5)` to the tree shown below. For assigning partial credit, indicate which nodes are involved in your zig-zig, zig-zag, and zig rotations.

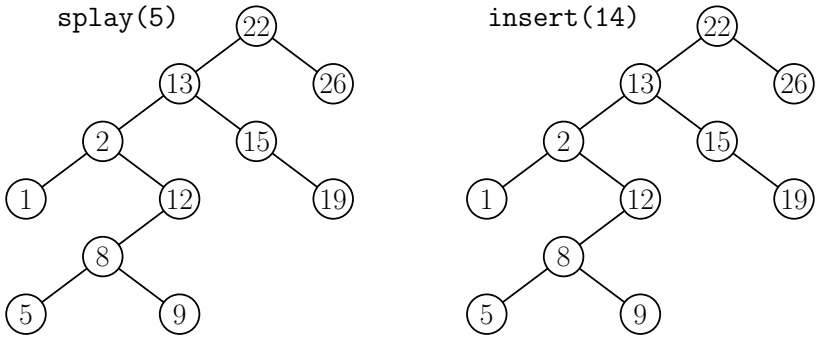


Figure 1: Splay tree operations.

(b) (5 points) Show the final tree that results from performing `insert(14)` to the tree shown below. For assigning partial credit, indicate which nodes are involved in your zig-zig, zig-zag, and zig rotations

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

(a) (4 points) 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) (5 points) You perform a preorder traversal of a *full binary tree* with n nodes. You have a counter that is incremented whenever you visit a leaf node and decremented whenever you visit an internal node. As function of n, what are the maximum and minimum values that this counter might achieve at any point in the traversal? (This is taken over all possible full binary trees with n nodes.)

(c) (5 points) 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 of trees as a function of k and n.

(d) (5 points) Your boss asks you to program a new function for your leftist heap. Given a leftist heap with n entries, 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?

(e) (4 points) You have just performed an insert into a 2-3 tree of height h. What is the maximum number of split operations that might be needed as a result?

(f) (4 points) You have just inserted n (distinct) keys into a treap. As a function of n, what is the probability that the smallest of the n keys is located at the root of the tree?

(1) 0 (That is, it cannot happen)
(2) Roughly $1/n$ (By "roughly", we mean "up to constant factors")
(3) Roughly $1/(\log n)$
(4) Roughly $1/2^n$
(5) Roughly $1/(n!)$

(g) (4 points) You have a skiplist containing n keys, where n is a very 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

(h) (4 points) You have just inserted a key into an AA tree having h levels. 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.)

Problem 3. (15 points) In this problem, we assume that we 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, `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) (5 points) 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 the figure below). 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.

(b) (10 points) 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 the figure below). It returns a pointer to the resulting tree.

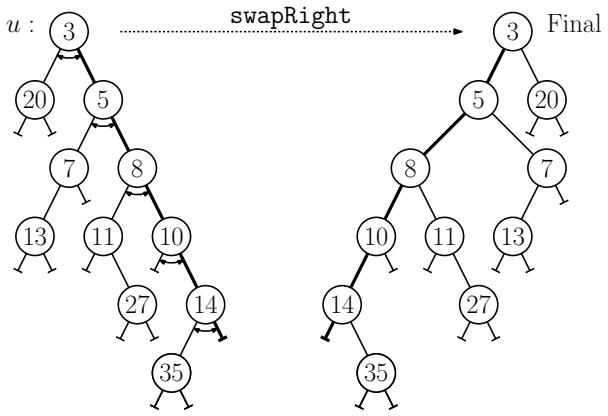


Figure 2: The function `swapRight`.

For full credit, your function should do this in one pass. That is, it is allowed to recurse down and return up, but that is all. For half credit, you can do it in two passes (one pass to merge and one pass to swap). You may use `swapRight` from (a).

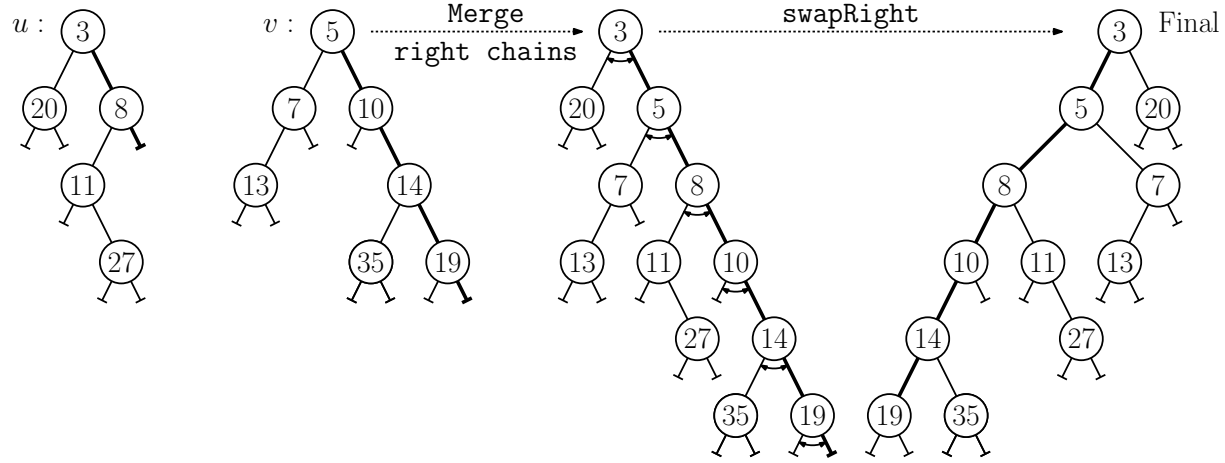
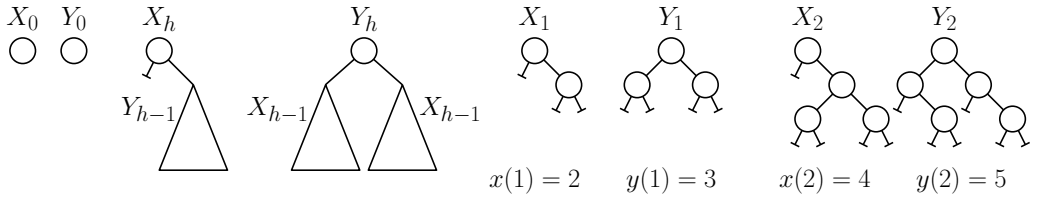


Figure 3: The function `swapMerge`.

Problem 4. (25 points) This problem involves the analysis of two new tree structures, called the *X-tree* and *Y-tree*, which are defined recursively in terms of each other. Let X_h and Y_h denote the X-tree and Y-tree of height h, respectively. X_0 and Y_0 are both defined to be a single node. For $h \geq 1$, X_h consists of a root node whose left child is `null` and whose right child is Y_{h-1}. The tree Y_h consists of a root node whose left and right children are both X_{h-1} (see the figure below).



(a) (3 points) Draw a picture of X_3 and a picture of Y_3. (You don't need to draw the null pointers.)

(b) (8 points) Let $x(h)$ and $y(h)$ denote the number of nodes in X_h and Y_h, respectively (see the figure above). Clearly $x(0) = y(0) = 1$. Assuming $h \geq 1$, give a formula that expresses $x(h)$ as a function of $y(h-1)$, and a formula that expresses $y(h)$ in terms of $x(h-1)$. **Hint:** The formulas are simple and do not involve any summations.

(c) (4 points) Assuming $h \geq 1$, give a formula that expresses $x(h)$ as a function of $x(h-2)$. **Hint:** The formula is simple and does not involve any summations.

(d) (10 points) Prove that if h is even, $x(h) = 3 \cdot 2^{h/2} - 2$.

Problem 5. (15 points) This is an extension of the Homework 1 problem on the *dual stack*, which stores two stacks in a single array. Recall that we are given an array `A` of length m, one of the stacks starts at index 0 and grows upwards and the other starts at index $m-1$ and grows downwards. Initially, the array has space for two entries ($m = 2$), and both stacks are empty.

Assuming we have space, each single operation has an actual cost of +1 unit. Whenever we run out of space, we expand the array as follows. Let n_1 and n_2 denote the numbers of elements in the two stacks. We allocate a new array of size $m' = w \cdot \max(n_1, n_2)$, for some integer constant w. (In Homework 1, $w = 3$, but here it will be a parameter that we can adjust). The actual cost of the expansion is equal to the total number of elements copied, that is, $n_1 + n_2$ (see Fig. 4).

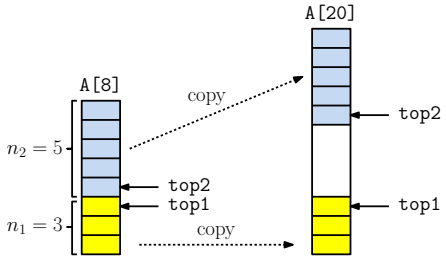


Figure 4: Expanding dual stack, where $w = 4$. When we run out of space, we allocate a new array of size $w \cdot \max(n_1, n_2) = w \cdot 5 = 20$. The actual cost is +8 for the copy and +1 for the final push.

(a) (3 points) Suppose that we have just performed a reallocation. Our current array is of size $n_1 + n_2 = m$, and our new array is of size $m' = w \cdot \max(n_1, n_2)$. What is the minimum number of stack operations until the new array needs to expand again? Express your answer as a function of some combination of m, m', and/or w. Briefly explain.

(b) (8 points) Derive the smallest constant c such that the amortized cost of our expanding dual stack is at most c and prove the correctness of your answer. (The value of c will depend on w.)

(c) (4 points) How small can w be before the amortized cost is no longer bounded by a constant? Briefly explain. (Remember that we required that w is an integer.)

CMSC 420: Fall 2022

Homework 3: Geometric Search and Hashing

Problem 1. (10 points) Consider the kd-tree shown in Fig. 1. Assume that the cutting dimensions alternate between x and y.

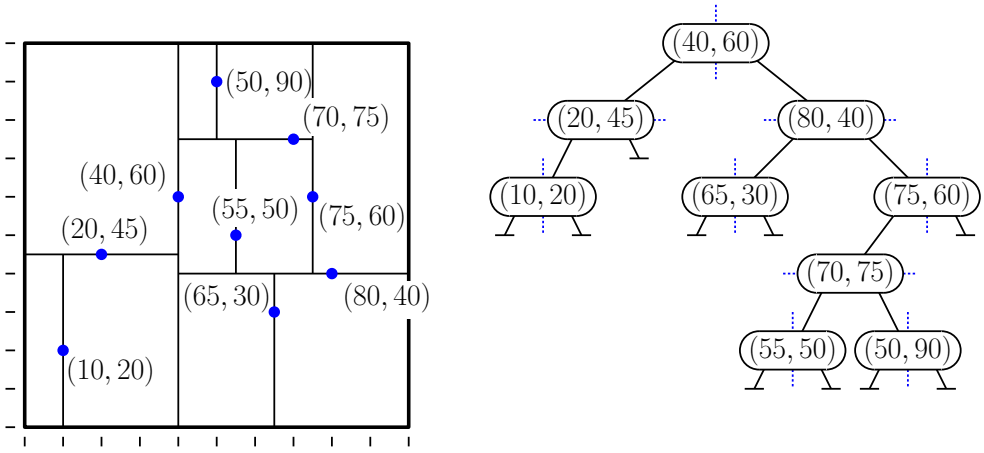


Figure 1: kd-tree operations.

(a) (5 points) Show the result of inserting $(30, 30)$ into this tree. As in Fig. 1 show both the tree structure (with cutting directions indicated) and the subdivision of space.

(b) (5 points) Show the kd-tree that results after deleting the point $(40, 60)$ from the *original* tree. Indicate which node is the replacement node, and show both the tree structure and the subdivision of space.

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

Problem 2. (5 points) Suppose that we are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ stored in a kd-tree. Recall that the *cell* of a node in the tree is the rectangular region of space associated with this node. Each `null` pointer of the tree is associated with a cell that contains no point of P, which we call a *null-pointer cell*. (In Fig. 2, the `null` pointers are shown as small squares.)

We assume that the kd-tree satisfies the *standard assumptions*: (1) The cutting dimensions alternate between x and y with each level of the tree, (2) subtrees are balanced in the sense that if the subtree rooted at a node p contains m points, then its two subtrees each contain roughly $m/2$ points. The following lemma follows from the analysis of orthogonal range searching from class (Lecture 14).

Lemma A: Given any kd-tree storing a set of n points in $\mathbb{R}^2$ that satisfies the standard assumptions and given any axis parallel line ℓ, the number of null-pointer cells that intersect ℓ is $O(\sqrt{n})$.

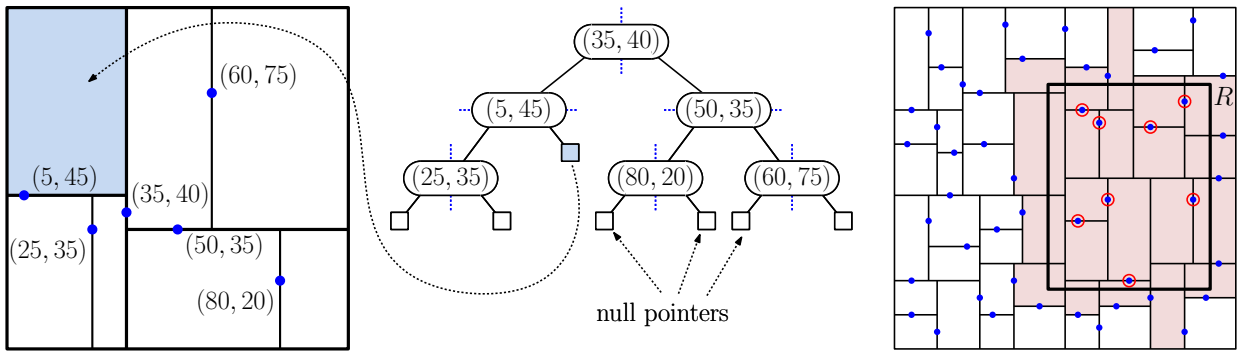


Figure 2: (a) A kd-tree and null-pointer cells and (b) illustration of Lemma B.

In this problem, we will generalize this result. Consider any axis-parallel rectangle R, and let k denote the number of points of P that lie within R. Prove the following result.

Lemma B: Given any kd-tree storing a set of n points in $\mathbb{R}^2$ that satisfies the standard assumptions and given any axis parallel rectangle R that contains k points of the set, the number of null-pointer cells that intersect R is $O(k + \sqrt{n})$.

Note that if a cell is completely contained within R, we consider to intersect R. For example, in Fig. 2(b), rectangle R contains 8 points of P (circled in red) and it intersects 21 null-pointer cells (shaded in pink).

Hint: There is no need to resort to solving recurrences. Classify the null-pointer nodes into two groups, those that are completely contained within R and those that partially overlap R.

Problem 3. (10 points) As in the previous problem, suppose that we are given a set $P = \{p_1, \ldots, p_n\}$ of n points in 2D space stored in a point kd-tree (see Fig. 3(a)), which satisfies the standard assumptions. 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. You may assume that there are no duplicate coordinate values among the points of P or the query point.

In a *segment sliding*, you are given a vertical line segment s, and the query returns the first point $p \in P$ that is hit if we were to slide the segment to its left (see Fig. 3(b)). If a point p_i lies on the segment, then the answer is p_i. If there is no point of P to the left of the segment, the query returns `null`. Since there are no duplicate x-coordinates, so the answer is always unique.

To simplify argument lists, let's assume that we have access to a class `VertSeg` that stores a vertical segment. Given an object `seg` is of this type, `seg.x` is its x-coordinate, `seg.ylo` is its lower y-coordinate, and `seg.yhi` is its upper y-coordinate.

(a) (7 points) Present pseudo-code for an efficient algorithm, `Point slideLeft(VertSeg seg)` for answering this query given the kd-tree. (**Hint:** As usual, create a recursive

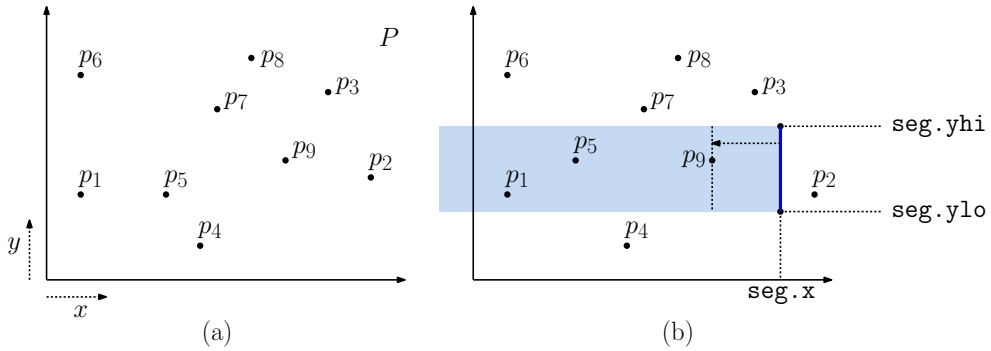


Figure 3: Vertical segment sliding queries.

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 in the search. For efficiency, you should avoid visiting nodes that cannot possibly contain the answer to the query.

(b) (3 points) Prove that your algorithm runs in $O(\sqrt{n})$ time. (**Hint:** Lemma B above is helpful here. It only bounds the number of null-pointer nodes, but you may assume that the bound applies to all the nodes that overlap the rectangle R. What is R in this case?)

Problem 4. (10 points) In this problem we will consider how to design a data structure for a particular application. You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$. Each point represents the coordinates (e.g., longitude and latitude) of a gas station (or if you prefer, a charging station for your EV). In addition to its coordinates, each point $p_i = (x_i, y_i)$, is associated with a *cost*, denoted c_i (see Fig. 4(a)). Think of this as the cost of refueling or recharging. These points and the costs are fixed, and you may build a data structure for answering the following queries.

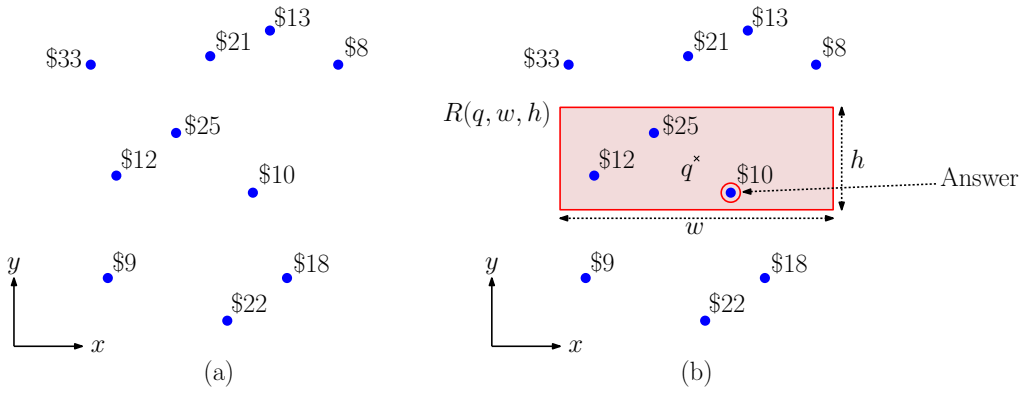


Figure 4: Cheapest fuel.

A query asks what is the lowest cost station in a given rectangular area (say, the map display on the car's navigation system). Your app knows the user's location, call it $q = (x_q, y_q)$, and there is a rectangle $R(q, w, h)$, called the *query rectangle* of width w and height h centered

about q (see Fig. 1). Your query is given q, w, and h, and the answer to the query is the identity of the station in P that lies within $R(q, w, h)$ (see Fig. 4(b)). For simplicity, you may assume that the costs are all distinct, so the answer is always unique.

Present an efficient data structure and algorithm for answering these queries. Our objective is a query time of $O(\log^a n)$ using space $O(n \log^b n)$, where $a, b > 0$ are small constants. (**Hint:** Explain how to modify range trees. Note that you cannot use a kd-tree, since that would have a query time of at least $O(\sqrt{n})$. Also note that you do not have time to list all the points inside the query rectangle, since that number could be as high as n.)

(a) (5 points) Describe your data structure, and derive its space bound. (**Hint:** Use a variant of 2-D range trees. What cost-related information is stored in each node of the tree?)

(b) (5 points) Explain how queries are answered, and derive the query time. (Note: Pseudocode is not required. A high-level English description is fine.)

Problem 5. (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. In each case, indicate the following:

- Was the insertion successful? (The insertion fails if the probe sequence loops infinitely without finding an empty slot.)
- If the insertion is successful, indicate the number of *probes*, that is, the number of array elements accessed. (The final location you place the key counts as a probe, so the number of probes is one more than the number of collisions encountered.)
- Show contents of the hash table after each insertion. (You will show three tables for each part.)

For the purposes of assigning partial credit, you can illustrate the probes made as we did in the lecture notes (with little arrows).

(a) (5 points) Show the results of inserting the keys "`X`" then "`Y`" then "`Z`" into the hash table shown in Fig. 5(a), assuming *linear probing*. (*Insert the keys in sequence, so if all are successful, the final table will contain all three keys.*)

(a) Linear probing

```
insert("X")  h("X") = 4
insert("Y")  h("Y") = 6
insert("Z")  h("Z") = 0
```

0	1	2	3	4	5	6	7	8	9
G		F		E	A	D		C	B

(b) Quadratic probing

```
insert("X")  h("X") = 6
insert("Y")  h("Y") = 3
insert("Z")  h("Z") = 8
```

0	1	2	3	4	5	6	7	8	9
			B	D		A		E	C

(c) Double hashing

```
insert("X")  h("X") = 14; g("X") = 6
insert("Y")  h("Y") = 5; g("Y") = 3
insert("Z")  h("Z") = 4; g("Z") = 4
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E		C		F	A			G		D		H		B

Figure 5: Hashing with linear probing, quadratic probing, and double hashing.

(b) (5 points) Repeat (a) using the hash table shown in Fig. 5(b) assuming *quadratic probing*. (**Hint:** You may wish to use the fact that for any integer $i \geq 0$, the value of i^2 mod 10, is one of $\{0, 1, 4, 5, 6, 9\}$.)

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

Challenge Problem: In Problem 2 above we analyze the number of null-pointer cells that intersect any axis-parallel line in a kd-tree in $\mathbb{R}^2$ with n points. In this problem, let's consider two natural generalizations to $\mathbb{R}^3$. Again, we have n points, and let us make the *standard assumptions* that the cutting dimensions cycle among x, y, and z, and subtree sizes are balanced.

(a) A plane is *axis-orthogonal* if it is orthogonal to one of the three coordinate axes. For example, the set $\{(x, y, z) \mid y = 13.4\}$ is a plane orthogonal to the y-axis that cuts the y-axis at 13.4.

Show that the number of null-pointer cells in a stanard-assumption kd-tree that intersect any axis-orthogonal plane is $O(n^c)$ for some constant $0 < c < 1$. Derive the value of c.

Hint: Derive a simple recurrence and apply the *Master Theorem* to solve it.

(b) An (infinite) line is *axis-parallel* if it is parallel to one of the three coordinate axes. For example, the set $\{(x, y, z) \mid x = 4, z = 7\}$ is a line parallel to the y-axis that pierces the (x, z)-coordinate plane at the point $(4, 0, 7)$.

Repeat part (a) for any axis-parallel line.

CMSC 420: Fall 2022

Practice Problems for Midterm 2

Problem 0. Expect at least one problem that involves working through some operations on a data structure that we have covered since the previous exam. (Good candidates are kd-trees and hashing, but I may draw something from the material shortly before the midterm, such as treaps, skiplists, or splay trees.)

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

(a) We have studied many classes of binary trees this semester. For this problem, let us ignore the keys and consider just the tree's node structure. 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 it is 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
(2) AVL trees
(3) Red-black trees
(4) AA trees
(5) Treaps
(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) In high dimensional spaces (say, dimensions greater than 10), kd-trees are preferred over quadtrees. Why is this?

(e) 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. As in class, we assume that 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.)

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

(g) You are using hashing with open addressing. Suppose that the table has just one empty slot in it. In which of the following cases are you *guaranteed* to succeed in finding the empty slot? (Select all that apply.)

(1) Linear probing (under any circumstances)
(2) Quadratic probing (under any circumstances)
(3) Quadratic probing, where the table size m is a prime number
(4) Double hashing (under any circumstances)
(5) Double hashing, where the table size m and hash function $h(x)$ are relatively prime
(6) Double hashing, where the table size m and secondary hash function $g(x)$ are relatively prime (that is, they share no common factors)

Problem 2. Suppose that you are given a treap data structure storing n keys. The node structure is shown in Fig. 1. 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
}
```

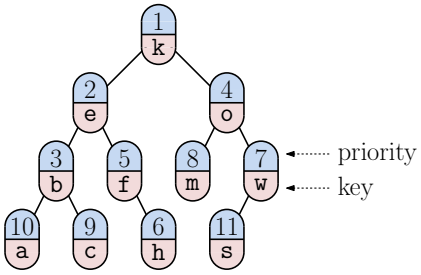


Figure 1: Treap node structure and an example.

(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 $x_0 \le x \le x_1$. 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. 1 the query `minPriority("c", "g")` would return 2 from node `"e"`, since it is the lowest priority among all keys x where `"c"` $\le x \le$ `"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 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 `size(p.left)/size(p)` $\ge 2/3$ (see the figure below). Let T be a left-heavy tree that contains n nodes.

(a) Consider any left-heavy tree T with n nodes, and let `s` be the leftmost node in the tree. Prove that `depth(s)` $\ge \log_{3/2} n$. (If you are super careful in your proof, you may discover this is not quite true. The actual bound is $(\log_{3/2} n) - c$, for a constant c. Don't worry about this small correction term.)

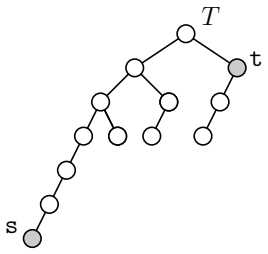


Figure 2: A left-heavy tree.

(b) Consider any left-heavy tree T with n nodes, and let `t` be the rightmost node in the tree. 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. This is useful when you believe that the next key you are searching for is close to the last one you visited.

Let us suppose we have a skiplist, with the node structure as shown in Fig. 3. Suppose that we have two keys, `x` $<$ `y`, and we have already found the node `p` that contains the key `x`. In order to find `y`, it would be wasteful to start the search at the head of the skip list. Instead, we start at `p`. The operation `forwardSearch(p, y)` searches for key `y` starting at node `p` (whose key is smaller than `y`). If `y` is found it returns the associated value, and otherwise it returns `null`.

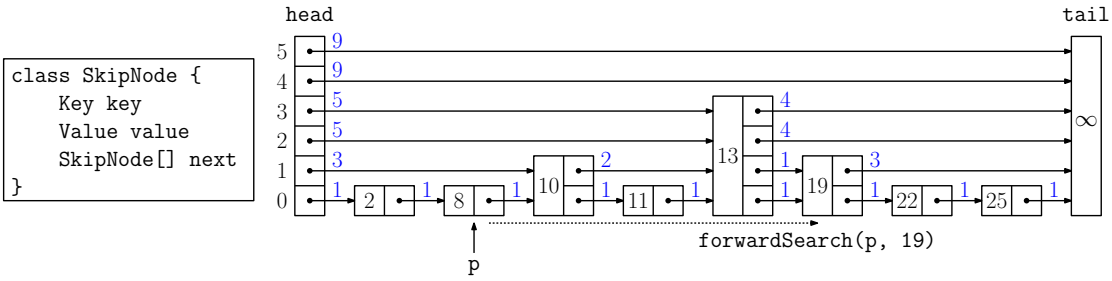


Figure 3: The operation `forwardSearch`.

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

Present pseudo-code for an algorithm for an efficient function. You do not need to analyze the running time. (**Hint:** The height of any node `p` can be determined as the length of its array of next pointers, that is, `p.next.length`.)

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).

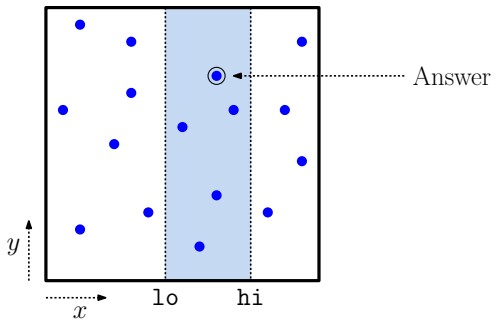


Figure 4: Partial-range max query.

(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. To simplify notation, let's assume we have an vertical strip object, `Strip`, where `strip.lo` and `strip.hi` are the strip bounds. You may make use of any primitive operations on points and rectangles (but please explain them). **Hint:** A possible signature for your helper function might be:

```
Point partialMax(Strip s, 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.

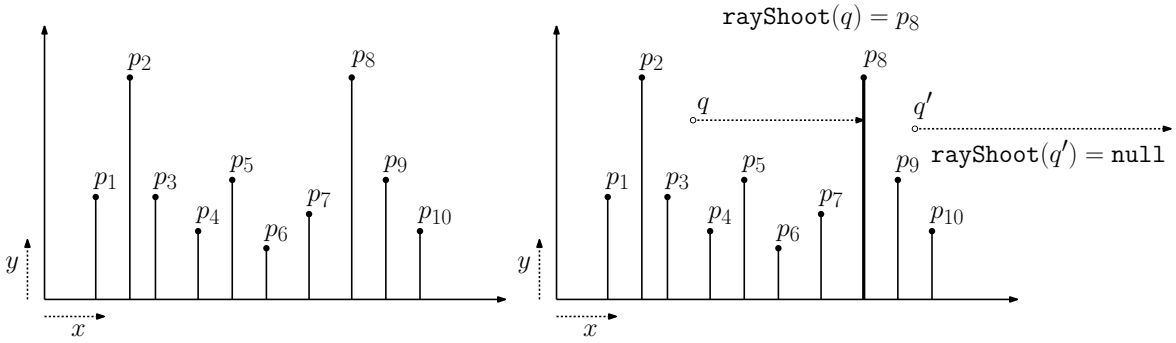


Figure 5: Ray shooting in a kd-tree.

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 might wonder how to store segments in a kd-tree. It turns out that to answer this query you do not need to store segments, just points. The function `rayShoot(q)` will invoke a recursive helper function. Here is a suggested form, which you are *not* required to use:

```
Point rayShoot(Point2D q, KDNode p, Rectangle cell, Point best),
```

Be sure to indicate how `rayShoot(q)` makes its initial call to the helper function.

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. In this problem, we will consider how to use/modify range trees to answer two queries efficiently. Throughout, $P = \{p_1, \ldots, p_n\}$ is a set of n points in $\mathbb{R}^2$ (Fig. 6(a)). Your answer should be based on range trees, you may make modifications to P including possibly transforming the points and adding additional coordinates.

In each case, the various layers of your search structure (what points are stored there and how they are sorted) and explain how your search algorithm operates. An English explanation (as opposed to pseudocode) is sufficient. Justify your algorithm's correctness and derive its running time.

(a) Assume that all the points of P have positive x- and y-coordinates. In a *platform-dropping query*, we are given a point $q = (q_x, q_y)$ with positive coordinates. This defines a horizontal segment running from the y-axis to q. The objective is to report the first point $p \in P$ that would be hit if we drop the platform (see Fig. 6(b)). Formally, this point has the maximum y-coordinate such that $p_x \leq q_x$ and $p_y \leq q_y$. If no point of P is hit by the platform, the query returns `null`.

Hint: Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

(b) In a *max empty-triangle query* you are given a point $q = (q_x, q_y)$. The objective is to compute the largest axis-parallel 45-45 right triangle that extends to the upper-right of q and contains no point of P in its interior. The answer to the query is the point of P that lies on the triangle's hypotenuse (see Fig. 6(c)). (Alternatively, you can think of this as sliding the 45° hypotenuse until it first hits a point of P). If the triangle can be grown to infinite size, return `null`.

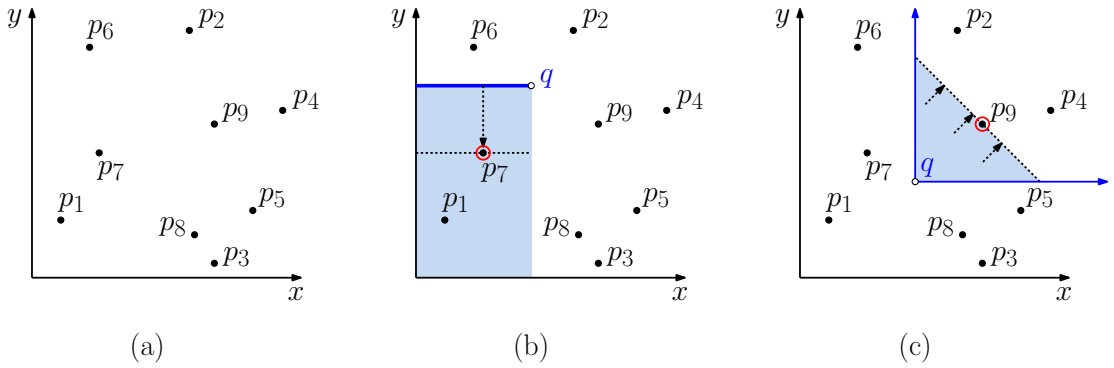


Figure 6: Platform-dropping and max empty-triangle queries.

Hint: Your data structure should use $O(n \log^2 n)$ storage and answer queries in $O(\log^3 n)$ time.

Problem 9. 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. (Don't give a formal analysis)**

CMSC 420 (0201) - Midterm Exam 2

Problem 1. (18 points) Hashing:

(a) (9 points) Show the results of inserting the sequence "`X`" then "`Y`" then "`Z`" into the hash table shown in Fig. 1(a), assuming *quadratic probing*. The operations are performed *as a sequence* (that is, prior insertions affect later ones). Indicate the number of *probes*, that is, array accesses. (The final insertion counts as a probe.) If the operation fails, give the number of probes as "∞".

(b) (9 points) Repeat (a) with the hash table shown in Fig. 1(b) assuming *double hashing*, where `g()` is the jump size.

(a) Quadratic Probing

```
insert("X")  h("X") = 3
insert("Y")  h("Y") = 9
insert("Z")  h("Z") = 4
```

0	1	2	3	4	5	6	7	8	9
C			A	E	D				B

(b) Double Hashing

```
insert("X")  h("X") = 4; g("X") = 6
insert("Y")  h("Y") = 5; g("Y") = 4
insert("Z")  h("Z") = 5; g("Z") = 2
```

0	1	2	3	4	5	6	7	8	9
F	C		A	E	D				B

Figure 1: Hashing.

Problem 2. (32 points) Short answer questions. No explanations required.

(a) (8 points) What are the min and max number of nodes in an AVL tree of height 2?

(b) (4 points) Which data structure did we see this semester that used the operation of *key rotation* (also called *adoption*) to maintain its structure?

(c) (4 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 1/3 and tails with probability 2/3. 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?

(d) (8 points) Suppose you store 20 points in $\mathbb{R}^2$ in an extended kd-tree with a bucket size of two (as in Programming Assignment 2). What are the minimum and maximum number of internal nodes this tree might have?

(e) (3 points) Among the open-addressing hashing methods we have covered (linear probing, quadratic probing, double hashing), which does the best job of *avoiding* clustering?

(f) (5 points) In a hash table, what is the definition of the *load factor*? (Let m denote the table size, and let n be the current number of entries.)

Problem 3. (10 points) You are given a standard (unbalanced) binary search tree, where each node `p` has a key (`p.key`) and left and right child pointers (`p.left` and `p.right`).

The operation `Key findUp(Key x)` returns the smallest key in the tree whose value is greater than or equal to `x`. If `x` appears in the tree, it returns `x`, and if all the keys in the tree are strictly smaller than `x`, it returns `null` (see Fig. 2).

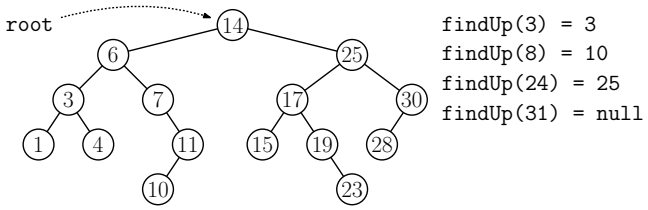


Figure 2: Find-up queries.

Present pseudocode for this function. Briefly explain how your function works. For full credit, your function should run in time $O(h)$, where h is the height of the tree. As always, you may create whatever helper functions you like, but you cannot alter the tree structure (e.g., you may not assume there are parent links or threads).

Problem 4. (15 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ stored in a kd-tree. Assume that every node `p` of the tree stores a point (`p.point`), its cutting dimension (`p.cutDim`), its cutting value (`p.cutVal`), and its size (`p.size`), which is defined to be the number of points in the subtree rooted at `p`. Assume that all the points are contained in a bounding box `bbox` (see Fig. 3(a)).

In a *circular range counting query* (CRC), you are given a center point $c = (c_x, c_y)$ and a radius r, and the problem is count the number of points of P that lie within the circular disk of radius r centered at c (see Fig. 3(b)).

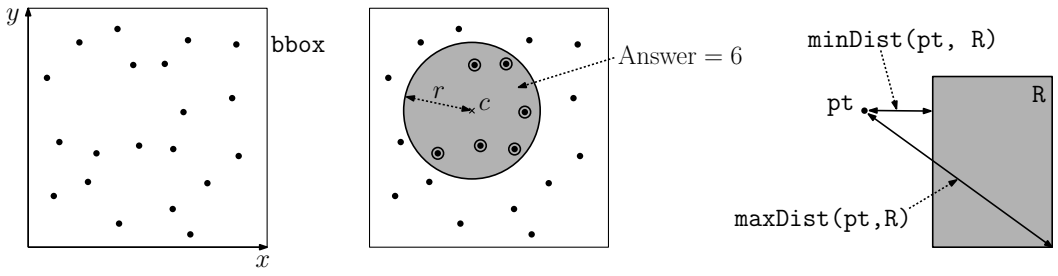


Figure 3: Circular range counting (CRC) queries.

(a) (10 points) Give pseudocode for an efficient algorithm, `int crc(Point c, double r)` for answering this CRC queries in the kd-tree.

Hint: Create a recursive helper function and explain how it is initially called. You may assume you have access to any geometric primitives you like (for example):

- `double dist(Point pt, Point q)`: Distance between `pt` and `q`
- `double minDist(Point pt, Rectangle R)`: Minimum distance between `pt` and `R`
- `double maxDist(Point pt, Rectangle R)`: Maximum distance between `pt` and `R`

(b) (3 points) No doubt, your algorithm is as efficient as possible for a kd-tree. But, what is the *worst-case* query time of your algorithm?

You may make the "standard assumptions" that the cutting dimensions alternate and the tree is well-balanced. Select the best option below. (No justification needed.)

(1) $O(\log n)$
(2) $O(\sqrt{n})$, irrespective of the number of points in the disk
(3) $O(\sqrt{n} + k)$, where k is the number of points in the circular disk
(4) $O(n)$
(5) $O(n \log n)$
(6) None of the above. (Indicate what you think it should be)

(c) (2 points) Suppose that your query algorithm reports that there are zero points of P in the circular disk. What is the *worst case* query time subject to this condition? (Same choices as in (b).)

Problem 5. (10 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^2$ (see Fig. 4(a)). In a *segment sliding query*, you are given a vertical line segment s, and the query returns the first point $p_i \in P$ that is hit if we were to slide the segment to its right (see Fig. 4(b)). If a point of P lies on the segment, then the answer is this point. If there is no point of P hit by the segment, the query returns `null`. You may assume there are no duplicate x-coordinates. Given a `Segment` object `s`, let `s.x` denote its x-coordinate, and let `s.yhi` and `s.ylo` denote its upper and lower y-coordinates, respectively.

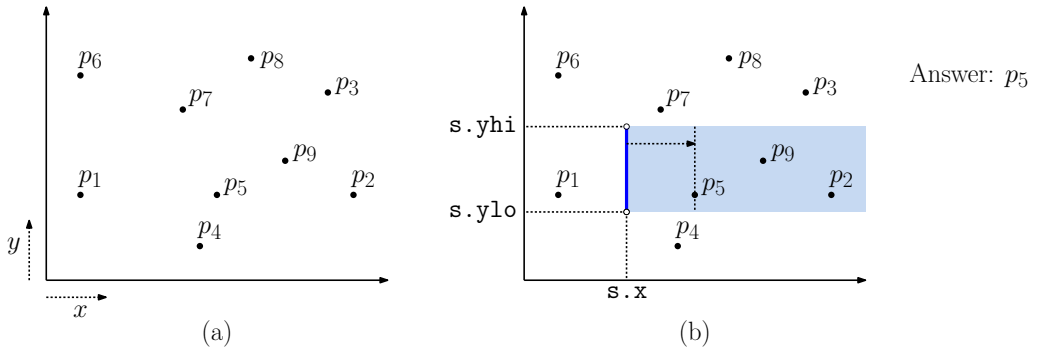


Figure 4: Vertical segment-sliding.

Present an efficient data structure and algorithm for answering these queries. Our objective is a query time of $O(\log^a n)$ using space $O(n \log^b n)$, where $a, b > 0$ are small constants. Partial credit will be given if your answer is correct, but not as efficient as it might be.

(a) (5 points) Describe your data structure and derive its space bound. (**Hint:** Use a variant of range trees. You will *not* get any credit for a kd-tree solution. Explain what layers

you use, how each layer is sorted, and what auxiliary information (if any) you store in each node.)

(b) (5 points) Explain how queries are answered and derive the query time. (**Note:** Pseudocode is not required. A high-level English description is fine.)

Problem 6. (15 points) Throughout this problem, we start with a large, perfectly balanced binary search tree (see Fig. 5(a)). The only operations we perform are `delete` and `find`.

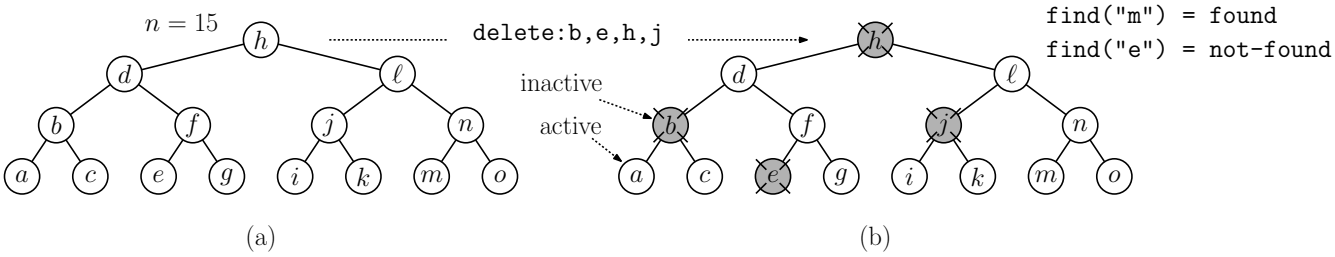


Figure 5: (a) A balanced binary search tree and (b) lazy deletion.

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. 5(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. 6). Let n denote the total number of nodes in the tree (both active and inactive).

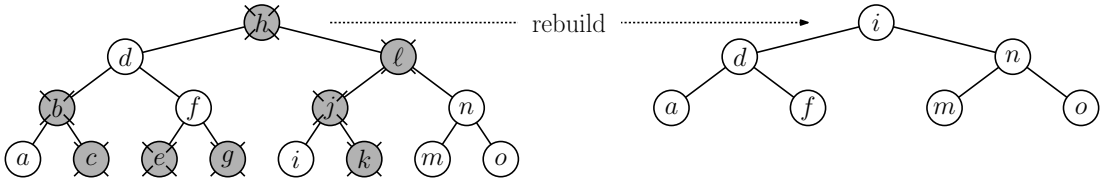


Figure 6: Rebuilding (with 8 inactive nodes and 7 active nodes).

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) (5 points) 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) (10 points) Derive the *amortized time* of lazy deletion. Express your answer asymptotically as a function of n (e.g., $O(1)$, $O(\log n)$ or $O(n)$.)

Hint: Analyze just a single run from an initial tree of size n until the next rebuilding. Since this is asymptotic, think of n has being very large and constant factors do not matter.

CMSC 420: Fall 2022

Homework 4: B-Trees, Tries, and Memory Management

Problem 1. (18 points, 6 points each) Consider the B-trees of order 4 shown in Fig. 1 below. Recall that each non-leaf node has between 2 and 4 children, and every node has between 1 and 3 keys. Let us assume two conventions. First, key rotation (when possible) has precedence over splitting/merging. Second, when splitting a node, if the number of keys shared by the two new nodes is an odd number, the rightmost node receives the larger number of keys. **Note: An earlier version said "leftmost" instead of "rightmost", but this is not consistent with the lecture notes, so I changed it. We will grant full credit either way you do it, as long as you apply your rule uniformly.**

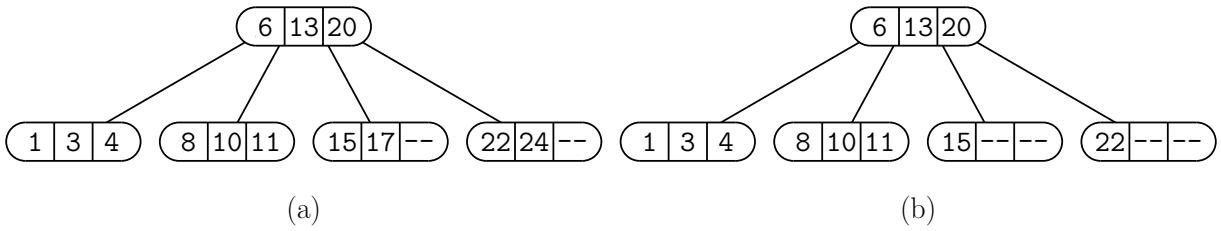


Figure 1: B-tree operations.

(a) Show the B-tree that results after inserting the key 9 into the tree of Fig. 1(a).

(b) Show the B-tree that results after inserting the key 2 into the (original) tree of Fig. 1(a).

(c) Show the B-tree that results after deleting the key 22 from the tree of Fig. 1(b).

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

Problem 2. (16 points, 8 points each) In this problem we will build a suffix tree for $S =$ `"aababaabaab$"`.

(a) Recall that the 12 suffixes of S are (in reverse order):

$$S_{11} = \texttt{"\$"}, \quad S_{10} = \texttt{"b\$"}, \quad S_9 = \texttt{"ab\$"}, \quad \ldots, \quad S_0 = \texttt{"aababaabaab\$"}.$$

Let id_j denote the *substring identifier* for S_j. (Recall from Lecture 18 that this is defined to be the shortest prefix of S_j that uniquely identifies it.) List all 12 substring identifiers for these suffixes in index order (from first to last $\mathrm{id}_0 \ldots \mathrm{id}_{11}$).

(b) Draw the suffix tree for S. Draw your tree in the same edge labeling style we used in Fig. 7 in Lecture 18 LaTeX lecture notes. Order the children of each node in alphabetical order from left to right. (The form of your drawing is important. There are many online suffix-tree generators, and if it appears that you copied your answer from one of these, you will receive no credit.)

Hint: Begin by writing out all the substring identifiers in alphabetical order, one above the other. This makes it easy to determine common substrings.

Problem 3. (16 points, 8 points each) This problem involves performing operations using the buddy system for memory allocation.

(a) Consider the buddy allocation shown in Fig. 2. Explain which blocks are split in order to perform the operation `alloc(2)`. Show the final blocks and indicate what level of the structure they reside. Assume that we always split the leftmost block of sufficient size. You may assume that the size of the final block is exactly 2, there is no need to round the size up for the sake of adding header information. (You don't need to redraw everything, just the portion that changes.)

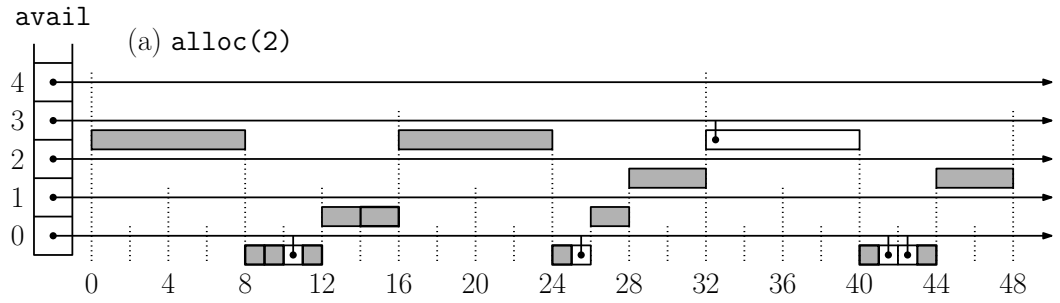


Figure 2: Problem 4(a): Buddy system allocation.

(b) Consider the buddy allocation shown in Fig. 3. Explain which blocks are merged in order to perform the operation `dealloc(24)`, which deallocates the shaded block of size 1 at address 24 as shown in the figure. Show the final merged block and indicate which level it resides at. (You don't need to redraw everything, just the portion that changes.)

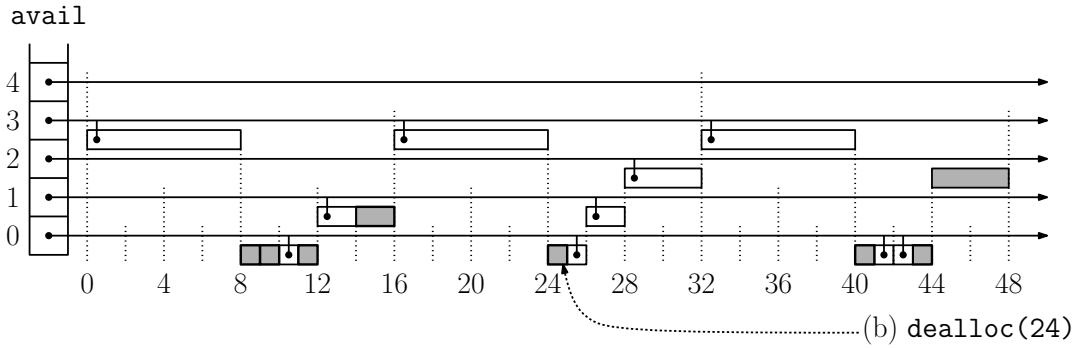


Figure 3: Problem 4(b): Buddy system deallocation.

CMSC 420: Fall 2022

Practice Problems for the Final Exam

Problem 0. Since the exam is comprehensive, please look back over the previous homework assignments, the two midterm exams, and the practice problems for both midterms. You should expect at least one problem that involves tracing through an algorithm or construction given in class.

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

(a) Let T be 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.)

(i) In a *postorder traversal*, all the external nodes appear in the order *before* any of the internal nodes

(ii) In a *preorder traversal*, all the internal nodes appear in the order *before* any of the external nodes

(iii) In an *inorder traversal*, internal and external node *alternate* with each other

(iv) None of the above is true

(b) 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.)

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

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

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

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

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

(c) 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.

(d) Repeat (c) in the case of deletion. (Give your answer as an asymptotic function of n.)

(e) 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?

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

(g) 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?

(h) 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.

(i) 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?

(j) Between the classical dynamic storage allocation algorithm (with arbitrary-sized blocks) or the buddy system (with blocks of size power of 2) which is more susceptible to *internal fragmentation*? Explain briefly.

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).

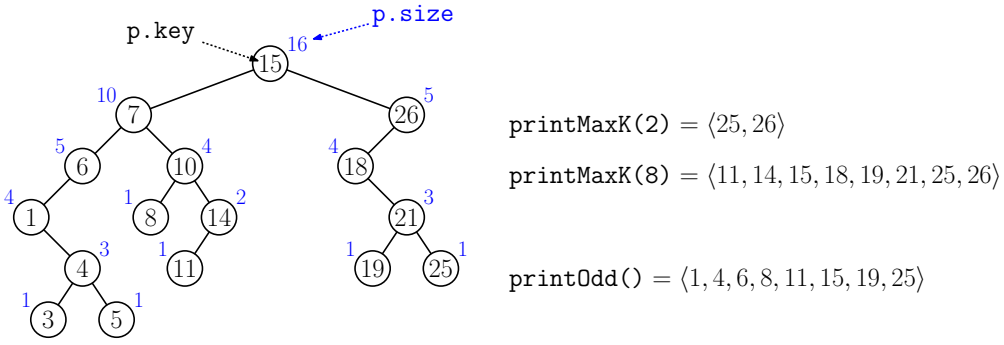


Figure 1: The functions printMaxK and printOdd.

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.

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. Throughout this problem, assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$. Assume the tree 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.) You may also assume that that any geometric computations on primitive objects (distances, disjointness, containment, etc.) can be computed in constant time, without explanation.

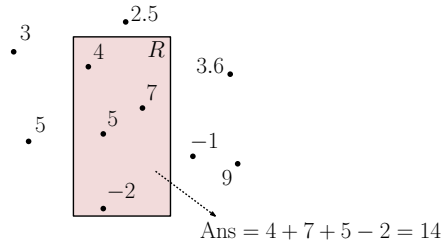


Figure 2: Weighted range query.

(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. 2(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 4. As in the previous problem, assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$ that satisfies the *standard assumptions*. 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. 3). Present (in pseudo-code) an efficient kd-tree algorithm that answers such a query.

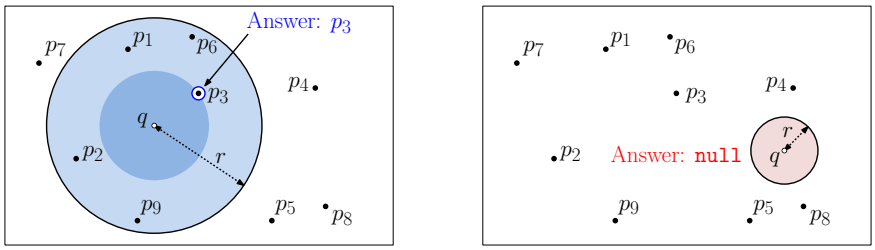


Figure 3: Faxed-radius nearest-neighbor query.

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 5. In this problem we will build a suffix tree for the string $S =$ `baabaababababaa$`.

(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 6. In this problem, we will consider how to use/modify range trees to answer two related queries. While the answer should be based on range trees, you may need to make modifications including possibly transforming the points and even adding additional coordinates. In each case, describe the points that are stored in the range tree and how the search process works. An English explanation (as opposed to pseudocode) is sufficient. Justify your algorithm's correctness and derive its running time.

(a) Assume you are given an n-element point set P in $\mathbb{R}^2$ (see Fig. 4(a)). In addition to its coordinates (p_x, p_y), each point $p \in P$ is associated with a numeric *rating*, p_z. In an *orthogonal top-k query*, you are given an axis-aligned query rectangle R (given, say, by its lower-left and upper-right corners) and a positive integer k. The query returns a list

of the (up to) k points of P that lie within R having the highest ratings (see Fig. 4(b)). (As an application, imagine you are searching for the k highest rated restaurants in a rectangular region of some city.)

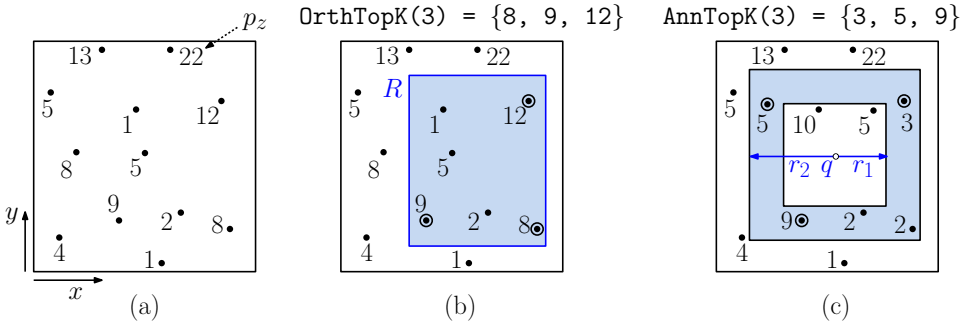


Figure 4: Orthogonal top-k queries and annulus top-k queries.

Describe how to preprocess the point set P into a data structure that can efficiently answer any orthogonal top-k query (R, k). Your data structure should use $O(n \log^2 n)$ storage and answer queries in at most $O(k \log^2 n)$ time. (I don't care how you handle edge cases, such as points lying on the boundary of the rectangle or points having the same rating.) If there are k points or fewer in the query region, the list will contain them all.

(b) In an *annulus top-k query* a query is given by a query point $q \in \mathbb{R}^2$ and two positive radii $r_1 < r_2$. Let $S_1 = S(q, r_1)$ be the square centered at q whose half side length is r_1 and define S_2 similarly for q and r_2. The square annulus $A(q, r_1, r_2)$ is defined to be the region between these two squares. The query returns a list of the (up to) k points of P that lie within the annulus $A(q, r_1, r_2)$ that have the highest ratings (see Fig. 4(c)).

Problem 7. In this problem, we are given a set L of n horizontal line segments $\overline{s_i t_i}$ in the plane, where $s_i = (x_i^-, y_i)$ and $t_i = (x_i^+, y_i)$ (Fig. 5(a–b)). We want to preprocess them to answer the following queries efficiently:

Segment stabbing queries: Consider a vertical query line segment with x-coordinate q_x, whose lower endpoint has the y-coordinate q_y^-, and whose upper endpoint has y-coordinate q_y^+. *How many of the segments of L does this segment intersect?* (For example, the vertical segment in Fig. 5(c) intersects 5 segments of L.)

Answer the following for the query vertical query (q_x, q_y^-, q_y^+) and horizontal segment $\overline{s_i t_i}$. (**Hint:** To simplify your answer, you may assume that all the coordinates are distinct, so the endpoint of one segment will never lie in the interior of another.)

(a) Describe a range tree-based data structure for this problem and derive its space bound. (Query processing comes later.)

(b) Explain how segment-stabbing queries are answered and derive the query time.

Problem 8. 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

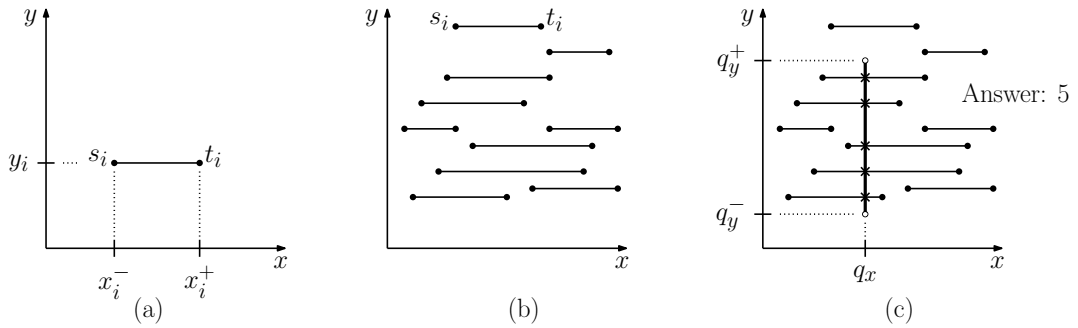


Figure 5: Segment stabbing queries.

`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`.)

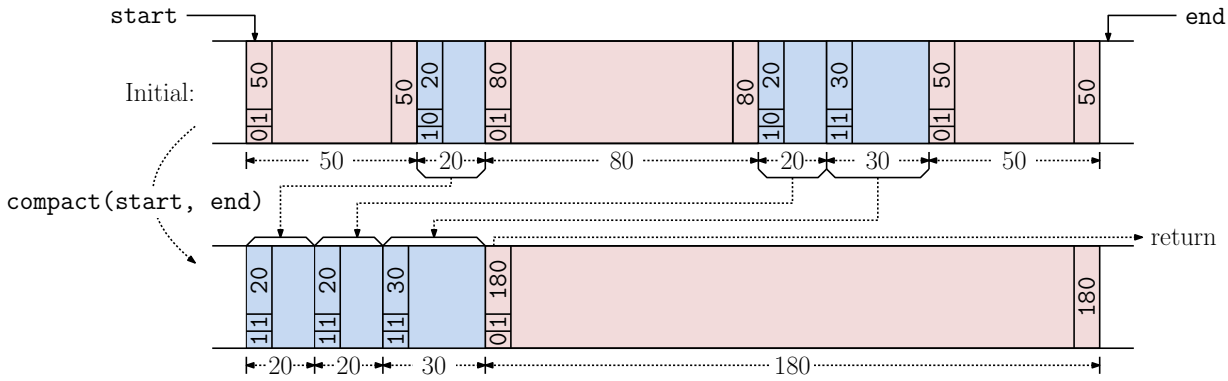


Figure 6: Memory compactor.

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 9. Recall the buddy system of allocating blocks of memory (see Fig. 7). Throughout this problem you may use the following standard bit-wise operators:

`&`	bit-wise "and"	`|`	bit-wise "or"
`^`	bit-wise "exclusive-or"	`~`	bit-wise "complement"
`<<`	left shift (filling with zeros)	`>>`	right shift (filling with zeros)

You may also assume that you have access to a function `bitMask(k)`, which returns a binary number whose k lowest-order bits are all 1's. For example `bitMask(3)` $= 111_2 = 7$.

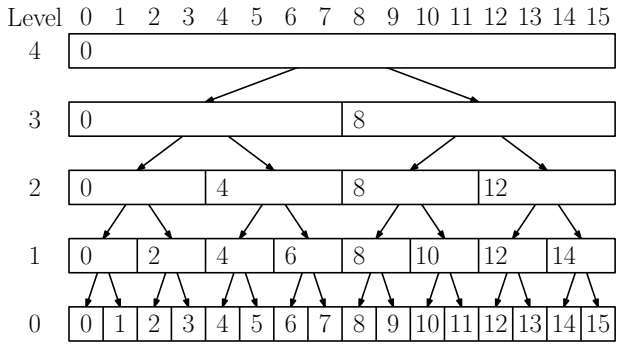


Figure 7: Buddy relatives.

Present a short (one-line) expression for each of the following functions in terms of the above bit-wise functions:

(a) `boolean isValid(int k, int x)`: True if and only if $x \geq 0$ a valid starting address for a buddy block at level $k \geq 0$.

(b) `int sibling(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address x, returns the starting address of its *sibling* (that is, its "buddy").

(c) `int parent(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address x, returns the starting address of its *parent* at level $k + 1$.

(d) `int left(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address x, returns the starting address of its *left child* at level $k - 1$.

(e) `int right(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address x, returns the starting address of its *right child* at level $k - 1$.

For example, given the tree shown in the figure, we have

```
isValid(2, 12) = isValid(2, 01100) = True
isValid(2, 10) = isValid(2, 01010) = False
sibling(2, 12) = sibling(2, 01100) =  8 = 01000
 parent(2, 12) =  parent(2, 01100) =  8 = 01000
   left(2, 12) =    left(2, 01100) = 12 = 01100
  right(2, 12) =   right(2, 01100) = 14 = 01110
```

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 $n = \texttt{top} + 1$ 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 (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. (50 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 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) (4 points) 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) (4 points) 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) At most one replacement is needed for each deletion operation

(d) (4 points) 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 of trees as a function of k and n.

(e) (4 points) 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?

(f) (8 points) 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.)

(g) (4 points) 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?

(h) (4 points) 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.)

(i) (4 points) You have just inserted n (distinct) keys into a treap. As a function of n, what is the probability that the smallest of the n keys is located at the root of the tree?

(1) 0 (That is, it cannot happen)
(2) Roughly $1/n$ (By "roughly", we mean "up to constant factors")
(3) Roughly $1/(\log n)$
(4) Roughly $1/2^n$
(5) Roughly $1/(n!)$

(j) (2 points) The AA-tree data structure has the following constraint: "*The left child of any node must be black.*" Which of the following operations is invoked when this condition is violated?

(k) (4 points) You have a skip list containing n keys, where n is a very 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? (Select one.)

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

(l) (4 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 3/4 and tails with probability 1/4. 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?

(m) (2 points) A node in a B-tree has too many children. Suppose that it is possible to resolve this either by *splitting* or *key-rotation* (adoption). Which is preferred? (No explanation needed.)

Problem 2. (15 points) In this problem, we will assume that we have a standard (unbalanced) binary search tree. Each node stores a `key`, `left` and `right` child links, and one additional field, `size`, which indicates the number of nodes in the subtree rooted at the current node.

(a) (5 points) Present pseudo-code for an operation `Node rotateLeft(Node p)`, which performs a left rotation at the given node `p`, and also updates the `size` values for all nodes whose sizes are affected by the rotation. It returns the node that takes the place of `p` after the rotation. You may assume that `p.right` is not `null` and all the size values are valid prior to the rotation. For full credit this should run in $O(1)$ time.

(b) (10 points) Present pseudo-code for an operation `Key getKth(int k)`. Given an integer $k \geq 1$, it returns the kth smallest key in the tree. If the tree has fewer than k keys,

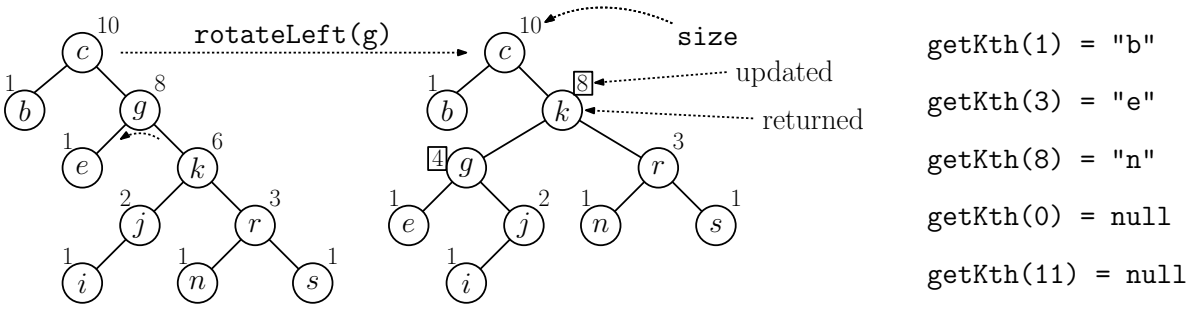


Figure 1: `rotateLeft` and `getKth`.

this should return `null`. (Hint: Write a helper function `Key getKth(int k, Node p)`, which returns the value of the kth smallest key in the subtree rooted at `p`.) Briefly explain how your function works. For full credit, this should run in time proportional to the height of the tree, independent of the value of k (which may be very large).

Problem 3. (10 points) Consider the following set of strings over the alphabet $\{\texttt{a}, \texttt{b}\}$.

$$\begin{aligned} S_0 &= \texttt{aaa} \\ S_1 &= \texttt{aabaabaab} \\ S_2 &= \texttt{aabaabb} \\ S_3 &= \texttt{aabab} \\ S_4 &= \texttt{bab} \\ S_5 &= \texttt{bba} \end{aligned}$$

Draw the *Patricia trie* for $\{S_0, \ldots, S_5\}$. (Remember that such a trie uses path compression whenever possible.) Draw your tree using the convention given in class, where each edge is labeled with the substring it matches. For consistency, order the children of each node in alphabetical order from left to right. Label the external nodes with integers 0 to 5 according to which string this is.

Problem 4. (10 points, 5 points each) This problem involves performing operations using the buddy system for memory allocation.

(a) Consider the buddy allocation shown in Fig. 2. Explain which blocks are split in order to perform the operation `alloc(2)`. Show the final blocks and indicate what level of the structure they reside. Assume that we always split the leftmost block of sufficient size. You may assume that the size of the final block is exactly 2, there is no need to round the size up for the sake of adding header information. (You don't need to redraw everything, just the portion that changes.)

(b) Consider the buddy allocation shown in Fig. 3. Explain which blocks are merged in order to perform the operation `dealloc(26)`, which deallocates the shaded block of size 1 at address 26 as shown in the figure. Show the final merged block and indicate which level it resides at. (You don't need to redraw everything, just the portion that changes.)

Problem 5. (15 points) The local weather service measures and stores temperatures throughout the day over many decades. (This is a lot of data!) Many people are interested in climate

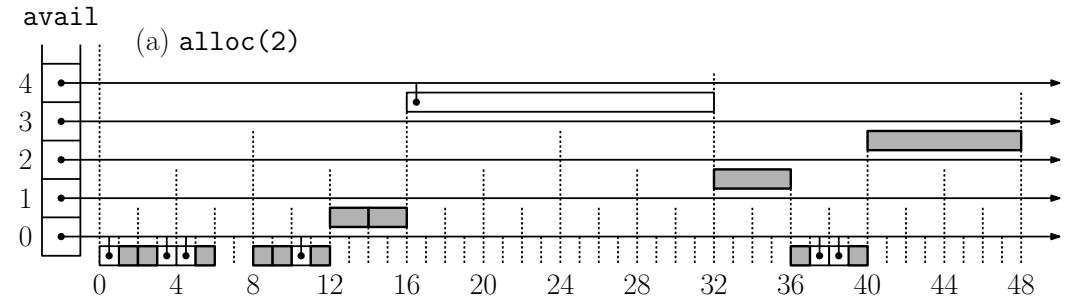


Figure 2: Buddy system allocation.

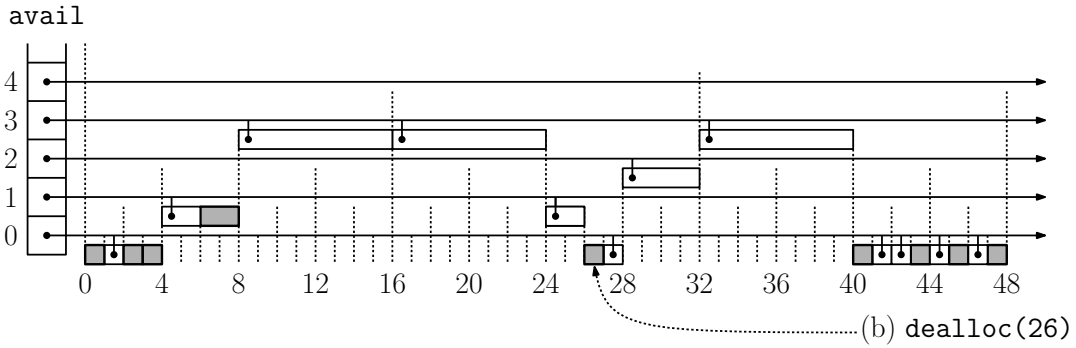


Figure 3: Buddy system deallocation.

change, and they are often asked *temperature queries* of the form: "When was the last time prior to 11:00am, June 16, 2022 when the temperature exceeded 105°?"

This can be modeled as a 2-dimensional retrieval problem. You are given a set of n points (x, y), where x represents the date and time of the measurement (encoded as a real number) and y denotes the temperature at that date and time (see Fig. 4, left).

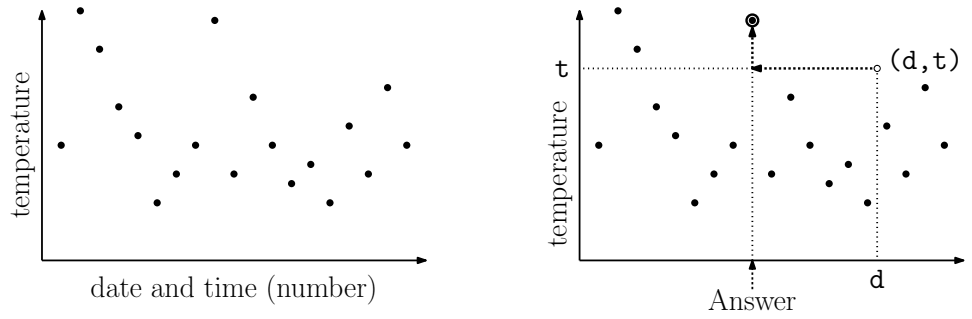


Figure 4: Temperature data and temperature queries.

Let us assume that these points are stored in a standard 2-dimensional kd-tree.

(a) (10 points) Present pseudocode for an efficient function that answers *temperature queries*. Given a pair (`d`, `t`), where `d` is a date/time (encoded as a real) and `t` is a temperature,

the query returns the date/time on or before d such that the temperature at that time was greater than or equal to t (see Fig. 4, right).

Assume a standard kd-tree (cutting dimensions alternate and the tree is well-balanced). For full credit, your algorithm should not visit any nodes that obviously cannot contribute to the final answer. You may assume any geometric primitive operations you like.

Hint: You can use any helper, but here is a suggestion:

```
double tempQuery(double d, double t, KDNode p, Rect cell, double best)
```

where `d` is the query date/time, `t` is the query temperature, `p` is the current node of the kd-tree, `cell` is the rectangular cell associated with `p`, and `best` is the best answer so far.

Checklist:

- Handle the case when `p == null`?
- Check whether the cell is relevant?
- Process `p.point`?
- Order of recursive calls?
- Initial call to your helper?

(b) (5 points) What is the running time of your algorithm? (No explanation needed.)

Problem 6. (10 points, 5 points each) This is a repeat of Problem 5 (Temperature Queries), but this time, explain how to solve it using *range trees*.

(a) Briefly describe your data structure and derive its space bound. (What layers are used? How is each sorted? What auxiliary data, if any, is stored in each node?)

(b) Briefly explain how queries are answered and derive the query time.

Problem 7. (10 points) Suppose that we are given a set of n items (initially each item in its own set), and we perform a sequence of m `unions` and `finds` (using height-balanced `unions` and path-compression `finds` as given in class). *Further, suppose that all the* ***`unions`*** *occur before any of the* ***`finds`****.* Prove that after initialization, the resulting sequence takes $O(m)$ time to execute (rather than the $O(m \cdot \alpha(m, n))$ time given in class).

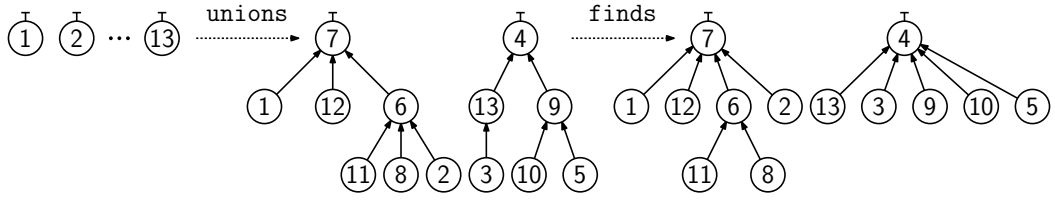


Figure 5: Unions before finds.

Hint: Classify the links of the Union-Find tree as being of two types: (1) those that point directly to a root node and (2) those that point to a non-root node. Start by proving that for any $k \geq 1$, if a `find` traverses k links, then $k - 1$ links in the tree switch from type-(2) to type-(1).