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

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

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

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

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

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

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

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

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

CMSC 420: Spring 2022

Programming Assignment 0: Dual List

Handed out: Thu, Jan 27. Due: **Tue, Feb 8 (11:59pm)**.

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, and the focus will be on reviewing Java programming and learning how to use our Gradescope testing environment.

Dual List: You will provide a Java implementation of a very simple data structure we call a `DualList`. This stores a multiset of entries (sometimes called a *bag*), each of which consists of a pair of values, which we call *keys* (e.g., (age, weight), (month, year), (country, population)). The data structure is *generic*, meaning that the types of the two keys are specified when the structure is declared. Here is the declaration:

```
class DualList<Key1 extends Comparable<Key1>, Key2 extends Comparable<Key2>>
```

The key types implement the Java interface `Comparable`, meaning that you can compare keys using `compareTo`. For example, to test $x < y$, you can do `x.compareTo(y) < 0`. This includes common object types such as `String`, `Integer`, `Float`, `Double`, and `Character`.

For example, `DualList<String, Integer>` stores string-integer pairs like `("Hello", 25)` and `DualList<Double, Character>()` stores double-character pairs like `(-23.57, 'X')`. (In the examples below, we will assume string-integer pairs.)

This *not* a map. (A *map* is a data structure storing key-value pairs, where each key is associated with a unique value.) It is just a collection of pairs. Since it is a multiset, the order of elements does not matter and duplicates are allowed. So the dual lists {`(A, 5), (Z, 3), (A, 5)`} and {`(Z, 3), (A, 5), (A, 5)`} are the same.

Efficiency is not important, and the list entries may be stored in any order you like. You may use any classes/functions from the Java libraries you like (e.g., Java `LinkedList`, `ArrayList`, `Collections.sort`).

Operations: Given the dual list `DualList<Key1, Key2>`, here are the operations that your program must support.

`DualList():` Constructor, which just creates an empty dual list.

`void insert(Key1 x1, Key2 x2):` Inserts the pair (x_1, x_2) into the dual list.

For example, given {`(A, 5), (Z, 3)`}, the operation `insert(A, 3)` would result in the list {`(A, 5), (Z, 3), (A, 3)`}. (The order of entries does not matter.)

`int size():` Returns the number of pairs in the dual list.

`ArrayList<String> listByKey1():` This returns a Java `ArrayList` of strings. The list is to be sorted in ascending order by the first key (with ties broken by the second key). Each string has the format `"(" + key1 + ",␣" + key2 + ")"`.

For example, given {`(A, 5), (Z, 3), (A, 3)`}, this returns a 3-element `ArrayList` containing three strings: `"(A, 3)"`, `"(A, 5)"`, and `"(Z, 3)"`. If the dual list is empty, the resulting `ArrayList` is also empty.

`ArrayList<String> listByKey2():` This is identical to `listByKey1()` except that the order is by the second key, with ties broken by the first key.

For example, in the above case, the result is: `"(A, 3)"`, `"(Z, 3)"`, and `"(A, 5)"`.

`Key2 extractMin1():` If the list is nonempty, this first finds the pair with the minimum first-key value x_1, it removes this pair from the list, and finally returns its associated *second-key value*. As with `listByKey1()`, ties are broken by second key value. If the list is empty, this throws an `Exception` with the error message `"Attempt to extract from an empty list"`.

For example, given the dual list {`(A, 5), (Z, 3), (A, 3)`}, both `(A, 5)` and `(A, 3)` have the minimum first-key value of `A`, and so the tie is broken in favor of the latter since $3 < 5$. We then remove `(A, 3)` from the list, and return its second-key value of 3. So, the list now contains {`(A, 5), (Z, 3)`}.

`Key1 extractMin2():` This is identical to `extractMin1` but with the roles of `Key1` and `Key2` reversed. It removes the pair with the minimum second-key value and returns the associated first-key value. It throws the same exception if the list is empty.

For example, given the dual list {`(A, 5), (Z, 3)`}, `(Z, 3)` has the minimum second-key value of 3, and so we then remove `(Z, 3)` from the list, and return its first-key value of `Z`. So, the list now contains {`(A, 5)`}.

All you need to do is to implement the above functions. We will provide a program that handles the input and output. An sample of input and output is shown below.

```
Input:             | Output:
insert:A:5         | insert(A, 5): successful
insert:Z:3         | insert(Z, 3): successful
insert:A:3         | insert(A, 3): successful
size               | size: 3
list-by-key1       | list-by-key1:
                   |   (A, 3)
                   |   (A, 5)
                   |   (Z, 3)
list-by-key2       | list-by-key2:
                   |   (A, 3)
                   |   (Z, 3)
                   |   (A, 5)
extract-min-key1   | extract-min-key1: 3
extract-min-key2   | extract-min-key2: Z
list-by-key1       | list-by-key1:
                   |   (A, 5)
```

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 template for `DualList`. 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. They can be found in the directory "`cmsc420_s22`", and all must begin with the statement "`package cmsc420_s22`".

`Tester.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 should not modify this except to change the input and/or output file names.*

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.

`CommandHandler.java`: This program provides the interface between `Tester.java` and our `DualList.java`. It invokes the functions in your `DualList` class and outputs the results. It also catches and processes any exceptions.

▷ *You should not modify this file.*

`DualList.java`: You provide the contents of this file. We will give you a template of its structure, and you fill in the details.

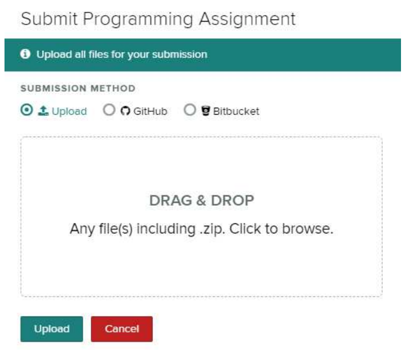
```
package cmsc420_s22; // Be sure to use this package!

import java.util.ArrayList;

public class DualList<Key1 extends Comparable<Key1>,
                      Key2 extends Comparable<Key2>> {
    public DualList() { ... }                             // constructor
    public void insert(Key1 x1, Key2 x2) { ... }          // insert new pair
    public int size() { ... }                             // number of pairs
    public Key2 extractMinKey1() throws Exception { ... } // remove Key1 min
    public Key1 extractMinKey2() throws Exception { ... } // remove Key2 min
    public ArrayList<String> listByKey1() { ... }         // Key1-sorted list
    public ArrayList<String> listByKey2() { ... }         // Key2-sorted list
}
```

▷ *Submit this file to the autograder.*

What you give us: All that you need to do is to fill in the implementation of the methods for the `DualList` class. Other than `DualList.java` and changing the file names in `Tester.java`, you should avoid modifying any of the directory structure or the files in the skeleton code.

Submission Instructions: Submissions will be made through Gradescope. There is no limit to the number of submissions you can make. The last submission will be graded. Here is what to do:

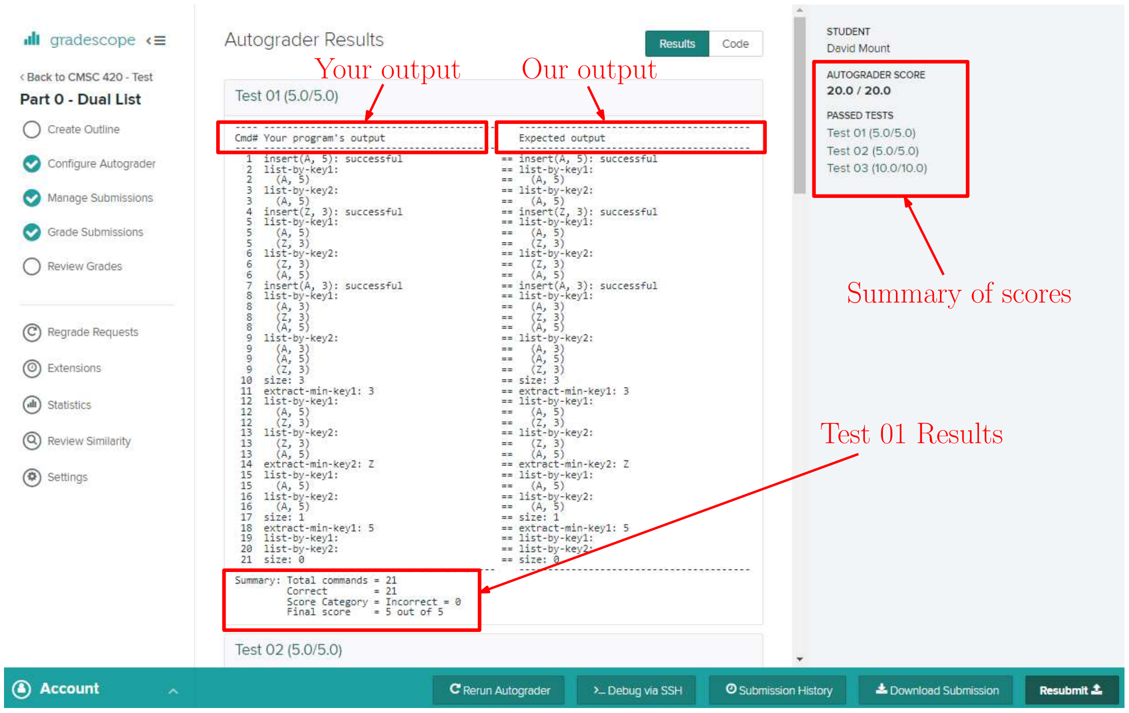
- Log into the CMSC420 page on Gradescope, select this assignment, and select "`Submit`". A window will pop up (see Fig. 1). Drag your file `DualList.java` into the window. If you generated other files, zip them up and submit them all. Select "`Upload`".



Submit Programming Assignment
Upload all files for your submission
SUBMISSION METHOD
Upload
GitHub
Bitbucket
DRAG & DROP
Any file(s) including .zip. Click to browse.
Upload
Cancel

Figure 1: Gradescope submission. Drag your file `DualList.java` into the box.

After a few minutes, Gradescope will display the results (see Fig. 2). Normally, a portion of your grade will depend on good style and efficiency, but for this initial program, only the autograder score will be used.



Test 01 (5.0/5.0)

```
---- ----------------------------------     ------------------------------------
Cmd# Your program's output                  Expected output
---- ----------------------------------     ------------------------------------
  1  insert(A, 5): successful            == insert(A, 5): successful
  2  list-by-key1:                       == list-by-key1:
  2    (A, 5)                            ==   (A, 5)
  3  list-by-key2:                       == list-by-key2:
  3    (A, 5)                            ==   (A, 5)
  4  insert(Z, 3): successful            == insert(Z, 3): successful
  5  list-by-key1:                       == list-by-key1:
  5    (A, 5)                            ==   (A, 5)
  5    (Z, 3)                            ==   (Z, 3)
  6  list-by-key2:                       == list-by-key2:
  6    (Z, 3)                            ==   (Z, 3)
  6    (A, 5)                            ==   (A, 5)
  7  insert(A, 3): successful            == insert(A, 3): successful
  8  list-by-key1:                       == list-by-key1:
  8    (A, 3)                            ==   (A, 3)
  8    (Z, 3)                            ==   (Z, 3)
  8    (A, 5)                            ==   (A, 5)
  9  list-by-key2:                       == list-by-key2:
  9    (A, 3)                            ==   (A, 3)
  9    (A, 5)                            ==   (A, 5)
  9    (Z, 3)                            ==   (Z, 3)
 10  size: 3                             == size: 3
 11  extract-min-key1: 3                 == extract-min-key1: 3
 12  list-by-key1:                       == list-by-key1:
 12    (A, 5)                            ==   (A, 5)
 12    (Z, 3)                            ==   (Z, 3)
 13  list-by-key2:                       == list-by-key2:
 13    (Z, 3)                            ==   (Z, 3)
 13    (A, 5)                            ==   (A, 5)
 14  extract-min-key2: Z                 == extract-min-key2: Z
 15  list-by-key1:                       == list-by-key1:
 15    (A, 5)                            ==   (A, 5)
 16  list-by-key2:                       == list-by-key2:
 16    (A, 5)                            ==   (A, 5)
 17  size: 1                             == size: 1
 18  extract-min-key1: 5                 == extract-min-key1: 5
 19  list-by-key1:                       == list-by-key1:
 20  list-by-key2:                       == list-by-key2:
 21  size: 0                             == size: 0
---- ----------------------------------     ------------------------------------
Summary: Total commands = 21
         Correct        = 21
         Score Category = Incorrect = 0
         Final score    = 5 out of 5
```

Test 02 (5.0/5.0)

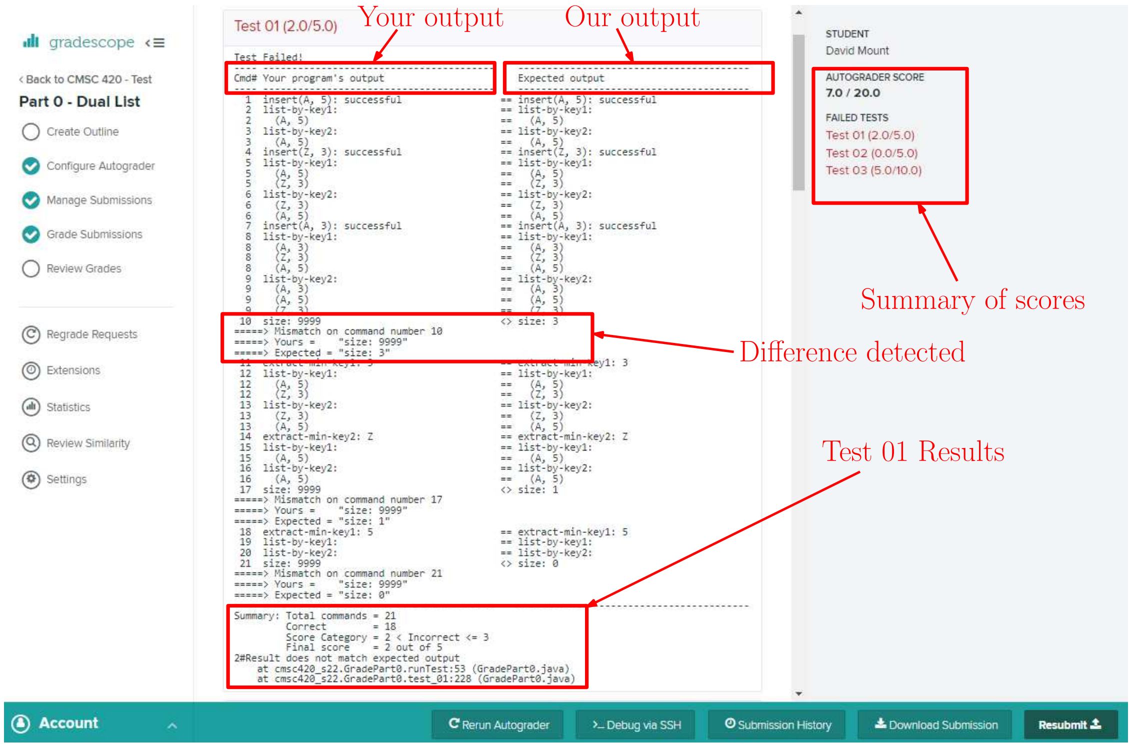
STUDENT
David Mount

AUTOGRADER SCORE
20.0 / 20.0

PASSED TESTS
Test 01 (5.0/5.0)
Test 02 (5.0/5.0)
Test 03 (10.0/10.0)

Figure 2: Gradescope autograder results (correct).

On the top-right of the page, it shows a summary of the scores of the individual tests as generated by the autograder. (If there are compilation errors, these will be displayed on this page.) The center of the window shows a line-by-line summary, with the output generated by your program on the left and the expected output on the right. If there are mismatches, these will be highlighted (see Fig. 3).



Test 01 (2.0/5.0)

```
Test Failed!
---- ----------------------------------------    ----------------------------------------
Cmd# Your program's output                        Expected output
---- ----------------------------------------    ----------------------------------------
  1  insert(A, 5): successful                  == insert(A, 5): successful
  2  list-by-key1:                             == list-by-key1:
  2    (A, 5)                                  ==   (A, 5)
  3  list-by-key2:                             == list-by-key2:
  3    (A, 5)                                  ==   (A, 5)
  4  insert(Z, 3): successful                  == insert(Z, 3): successful
  5  list-by-key1:                             == list-by-key1:
  5    (A, 5)                                  ==   (A, 5)
  5    (Z, 3)                                  ==   (Z, 3)
  6  list-by-key2:                             == list-by-key2:
  6    (Z, 3)                                  ==   (Z, 3)
  6    (A, 5)                                  ==   (A, 5)
  7  insert(A, 3): successful                  == insert(A, 3): successful
  8  list-by-key1:                             == list-by-key1:
  8    (A, 3)                                  ==   (A, 3)
  8    (Z, 3)                                  ==   (Z, 3)
  8    (A, 5)                                  ==   (A, 5)
  9  list-by-key2:                             == list-by-key2:
  9    (A, 3)                                  ==   (A, 3)
  9    (A, 5)                                  ==   (A, 5)
  9    (Z, 3)                                  ==   (Z, 3)
 10  size: 9999                                <> size: 3
=====> Mismatch on command number 10
=====> Yours =    "size: 9999"
=====> Expected = "size: 3"
 11  extract-min-key1: 3                       == extract-min-key1: 3
 12  list-by-key1:                             == list-by-key1:
 12    (A, 5)                                  ==   (A, 5)
 12    (Z, 3)                                  ==   (Z, 3)
 13  list-by-key2:                             == list-by-key2:
 13    (Z, 3)                                  ==   (Z, 3)
 13    (A, 5)                                  ==   (A, 5)
 14  extract-min-key2: Z                       == extract-min-key2: Z
 15  list-by-key1:                             == list-by-key1:
 15    (A, 5)                                  ==   (A, 5)
 16  list-by-key2:                             == list-by-key2:
 16    (A, 5)                                  ==   (A, 5)
 17  size: 9999                                <> size: 1
=====> Mismatch on command number 17
=====> Yours =    "size: 9999"
=====> Expected = "size: 1"
 18  extract-min-key1: 5                       == extract-min-key1: 5
 19  list-by-key1:                             == list-by-key1:
 20  list-by-key2:                             == list-by-key2:
 21  size: 9999                                <> size: 0
=====> Mismatch on command number 21
=====> Yours =    "size: 9999"
=====> Expected = "size: 0"
-------------------------------------------------------------------------------------------
Summary: Total commands = 21
         Correct        = 18
         Score Category = 2 < Incorrect <= 3
         Final score    = 2 out of 5
2#Result does not match expected output
    at cmsc420_s22.GradePart0.runTest:53 (GradePart0.java)
    at cmsc420_s22.GradePart0.test_01:228 (GradePart0.java)
```

STUDENT
David Mount

AUTOGRADER SCORE
7.0 / 20.0

FAILED TESTS
Test 01 (2.0/5.0)
Test 02 (0.0/5.0)
Test 03 (5.0/10.0)

Figure 3: Gradescope autograder results (incorrect).

The final score is based on the number of commands for which your program's output differs from ours. Note that the comparison program is very primitive. It compares line by line (without considering the possibility of inserted or deleted lines) and is sensitive to changes in case and the addition of white-space.

CMSC 420: Spring 2022

Programming Assignment 1A: Quake Heaps (Insertion and Merging)

Handed out: Thu, Feb 10. Due: **Thu, Feb 24, 11:59pm**.

Overview: This is the first in a two-part assignment to implement an interesting data structure called a *Quake Heap*. As with standard heaps, this data structure implements a *priority queue*. Such a data structure stores key-value pairs, where keys are from a totally ordered domain (such as integers, floats, or strings). At a minimum, a priority queue supports the operations of *insert* (add a new key-value pair) and *extract-min* (remove the entry with the smallest key, and return its associated value).

The most famous example of a heap is the *binary heap*, which is the data structure used by HeapSort. There are numerous variants, which provide improved performance for various operations, notably that of decreasing a key. The quake heap is such a variant. It was developed by Timothy Chan (described in this paper) as a simpler alternative to the Fibonacci Heap. It supports insertion and decreasing keys in $O(1)$ time, and it supports extract-min in $O(\log n)$ amortized time. (More details can be found on the CMSC420 Projects page.)

In Part-A of the assignment, we will implement only a portion of the quake heap functionality. *We will discuss the quake heap in a future lecture, but this part of the assignment is completely self-contained.* In Part-B, we will implement all the functionality.

Quake Heap: The Java class is called `QuakeHeap`. It is generic, templated by two types `Key` and `Value`. The `Key` type implements the Java `Comparable` interface, meaning that it must provide a function `compareTo()` for comparing keys. We also assume that both types support a (stable) `toString()` method.

The quake heap is represented as a collection of binary trees, where each node stores a key-value pair. The nodes of these trees are organized into *levels*. All the leaves reside on level 0, and each key in the heap is stored in exactly one leaf of some tree. (For example, in Fig. 1, the keys consist of the 13 keys in the blue leaf nodes.)

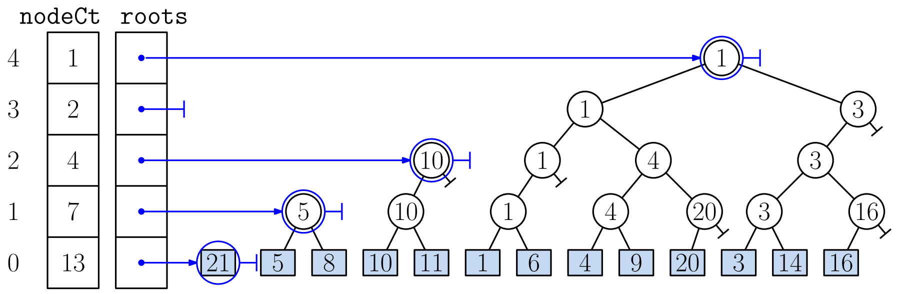


Figure 1: A quake heap storing 13 keys $\{21, 5, 8, 10, \ldots, 16\}$ (values are not shown).

Each internal node has a left child and an optional right child. Its key value is that of its left child. If the right child exists, its key value is greater than or equal to the left child. The root holds the smallest key value over all the leaves, which (by our rule that the left key is smaller) is that of the leftmost leaf. *It follows that the smallest key in the heap will be stored in one of the roots* (but we don't generally know which).

Each node stores a key-value pair, left and right child pointers, parent pointer, and its level. We maintain two additional arrays organized by level:

- `roots[lev]`: A linked list containing references to the tree roots of level `lev`. (We recommend implementing this as a Java `LinkedList` of nodes).
- `nodeCt[lev]`: Stores the total number of nodes at level `lev`.

Nodes and Locators: The principal objects being manipulated are the nodes in the quake heap. As mentioned above, each node stores a key-value pair, left and right child links, parent link, and its level in the tree. Nodes at level 0 are leaves, so both child links are `null`. A root node (at any level) has a parent link of `null`. Java provides an elegant way to define nodes by simply nesting a class, say `Node`, inside your `QuakeHeap` class (see the code block below).

One tricky element in any heap structure that supports decrease-key is that we need a mechanism for identifying the entry whose key we wish to decrease. When we insert a key-value pair, we create a new leaf node. Since `Node` is a protected object within `QuakeHeap`, we cannot return a pointer directly to it. Instead, we create a special public object, called a `Locator`, to enclose a reference to this newly inserted leaf node. The insert function returns a locator referencing the newly created node. A skeletal example is provided below.

```
package cmsc420_s22;
public class QuakeHeap<Key extends Comparable<Key>, Value> {

    class Node {                                    // a node in the heap
        Key key;
        Value value;
        // ... whatever else you need in your node
    }

    public class Locator {                          // locates a node
        private Node u;                             // the node
        private Locator(Node u) { this.u = u; }     // constructor
        private Node get() { return u; }            // getter
    }

    public Locator insert(Key x, Value v) {         // insert (x,v)
        // ... your code to create a new leaf node u
        return new Locator(u);
    }

    // ... other QuakeHeap members
}
```

Operations: For this part of the project, we will begin by implementing the basic functions needed to insert keys and to build subtrees. Here is a list of the operations you are to implement. (Further details available on the CMSC420 Projects page.)

`QuakeHeap(int nLevels)`: This constructs an empty quake heap. The parameter `nLevels` indicates the number of levels to allocate in your arrays `roots` and `nodeCt`.[1] This allocates and initializes the `roots` and `nodeCt` arrays and any other private data that your class uses.

`void clear()`: This resets the structure to its initial state. In particular, it resets all the node counts to zero and clears all the `roots` lists.[2]

`Locator insert(Key x, Value v)`: This inserts the key-value pair (x, v) in the heap. This creates a "trivial" tree consisting of a single root node at level 0, that stores this key-value pair. It inserts this node into the list `roots[0]`. It returns a `Locator` (see above) referencing the newly created leaf node.

`int getMaxLevel(Locator r)`: Given a locator `r`, this determines the maximum height of an ancestor reachable by following the reversal of left-child links up the tree. (If the keys are unique, this is the highest node in the tree that has the same key as `r`). For example, for the heap in Fig. 1, the max-level of leaf labeled 21 is 0, the max-level of 4 is 2, the max-level of 1 is 4, and the max-level of 16 is 1.

`Key getMinKey()`: This returns the smallest key in the heap and also reorganizes the heap, merging many small trees into one large tree.

- If the heap is empty, throw an `Exception` with the message `"Empty heap"`.
- Otherwise, find the minimum key in the heap by enumerating all the nodes in all the `roots` lists, we find the one with the smallest key. (Ties may be broken arbitrarily.)
- Next, consolidate trees by the following process, called `merge-trees`. Enumerate the levels bottom-up, from zero up to the second highest level (that is, `nLevels-2`). At each level `k`:
  - Sort the nodes of `roots[k]` in increasing order by their keys.[3] Ties may be broken arbitrarily.
  - Next, merge trees in pairs as follows. While `roots[k]` has at least two roots:
    - Extract the first two root nodes from the sorted list. Call them `u` and `v`. Since the list is sorted, we know that `u.key` $\leq$ `v.key`.
    - Create a new root node `w`, with `u` as its left child and `v` as its right child. (Don't forget to set `u` and `v`'s parent links to point to `w`.) By our convention, `w`'s key is set to `u.key`. (We don't care about `w`'s value field. You can just set it to `null`.)
    - Add `w` to `roots[k+1]`.

Observe that when the merge-tree process is finished, every level, except possibly the top one, has at most one root. This is illustrated in Fig. 2.

[1] From a software design perspective, it would be better if the constructor did not have this parameter, and the array just grows dynamically as needed. Limiting the array size will be useful for testing purposes.

[2] You might realize that there is a potential memory leak here since locators may have been generated that refer to entries that have been removed. Fixing this is not trivial, but we won't worry about it.

[3] If you store your `roots[k]` as a Java `LinkedList`, you can invoke `Collections.sort()` to sort them.

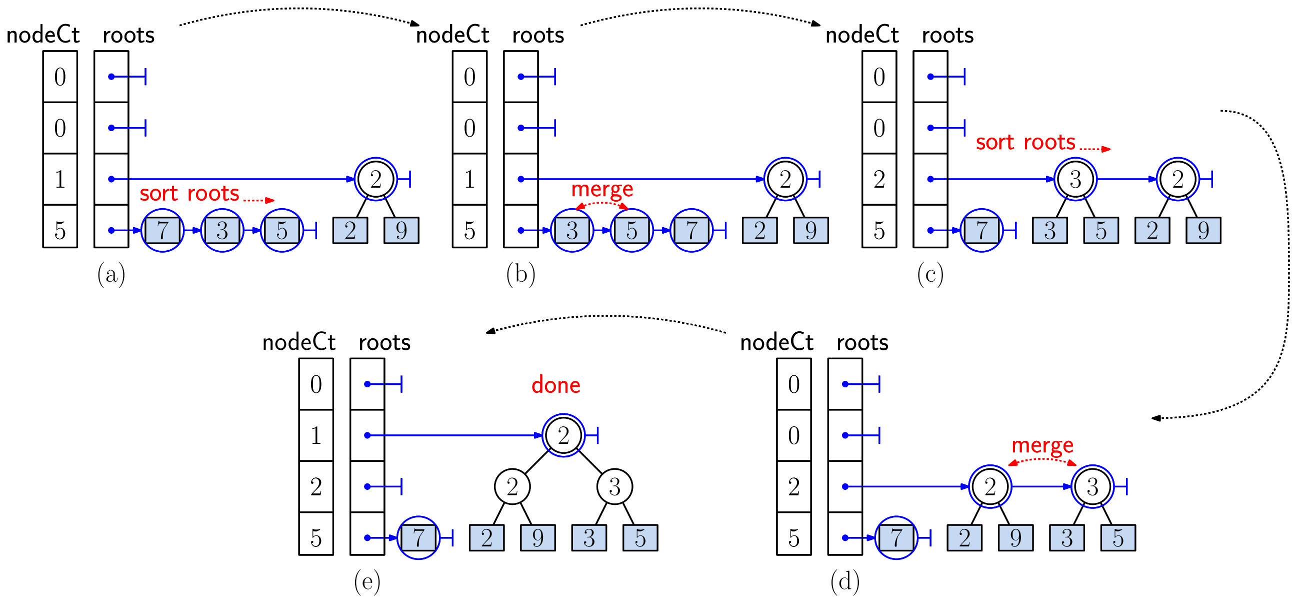


Figure 2: Merging trees. Working bottom-up, we sort the roots at each level, and then merge consecutive pairs until either zero or one root remains.

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

Enumerate the levels of the tree from bottom up. For each level, do the following:

- If the the node count for this level is zero, skip this level and go on to the next. Otherwise, sort the root nodes of this level by their key values, just as in `getMinKey()`.
- Generate a *level header* in the form of a string "`{lev:` *xxx* `nodeCt:` *yyy*`}`" and add it to the array-list. Here, "*xxx*" is the level index and "*yyy*" is the node count for this level. For example, if there are four nodes on level two, this generates the string, "`{lev: 2 nodeCt: 4}`".
- For each root node r in the `roots` list for this level, enumerate the the nodes of this tree based on a preorder traversal. For each node `u` visited in this traversal, we do the following:

  Internal: (`u.level` ≥ 1) Generate the string `"(" + u.key + ")"`. Recursively visit `u.left` and `u.right`.

  Leaf: (`u.level` $= 0$) Generate the string `"[" + u.key + "␣" + u.value + "]"` (where "␣" denotes a single space) and return.

  Null: (`u` $=$ `null`) This cannot happen in Part-A, but it can in Part-B. If so, generate the string `"[null]"` and return.

As an example, invoking `listHeap` on the structure appearing in to Fig. 2(e) would result in the 11-element array-list shown below. (For simplicity, we set the value of key x to be the string "`X0`x".)

Index	Array-List Contents
0:	{lev: 0 nodeCt: 5}
1:	[7 X07]
2:	{lev: 1 nodeCt: 2}
3:	{lev: 2 nodeCt: 1}
4:	(2)
5:	(2)
6:	[2 X02]
7:	[9 X09]
8:	(3)
9:	[3 X03]
10:	[5 X05]

Unfortunately, it is not easy to interpret the tree structure from this preorder listing, but we have provided a function in `CommandHandler.java` that reformats the tree so it is easier to read. For example, given the above array-list, our function would generate the following output for you. (Contrast this with the tree of Fig. 2(e).)

```
Structured list:
  {lev: 0 nodeCt: 5}
    Tree: 0
      [7 X07]
  {lev: 1 nodeCt: 2}
  {lev: 2 nodeCt: 1}
    Tree: 0
      | | [2 X02]
      | (2)
      | | [9 X09]
      (2)
      | | [3 X03]
      | (3)
      | | [5 X05]
```

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 `QuakeHeap.java`. Remember that you must use the package "`cmsc420_s22`" in all your source files in order for the autgrader to work. As before, we will provide the programs `Tester.java` and `CommandHandler.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 `QuakeHeap.java`.

```
package cmsc420_s22; // don't change this!
import java.util.ArrayList;

public class QuakeHeap<Key extends Comparable<Key>, Value> {

    class Node { ... }
    public class Locator { ... }
```

```
        public QuakeHeap(int nLevels) { ... }
        public void clear() { ... }
        public Locator insert(Key x, Value v) { ... }
        public Key getMinKey() throws Exception { ... }
        public int getMaxLevel(Locator r) { ... }
        public ArrayList<String> listHeap() { ... }
    }
```

Efficiency requirements: The function `insert()` should run in $O(1)$ time, the function `getMinKey()` should run in time proportional to the number of roots (plus the time needed for sorting each level), and the function `getMaxLevel()` should run in time proportional to the maximum number of levels. A portion of your grade will depend on the efficiency of your program.

Testing/Grading: Submissions will be made through Gradescope (you need only upload your modified `QuakeHeap.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 30 points. Of these, 25 points will be purely for input/output correctness as tested by the autograder, and the remaining 5 points will be for clean programming style and the above efficiency requirements.

CMSC 420: Spring 2022

Programming Assignment 1B: Quake Heaps (Decreasing and Extracting)

Handed out: Tue, Mar 8. Due: **Thu, Mar 31, 11:59pm.**

Overview: This assignment is the continuation of Programming Assignment 1A on the Quake Heap. In Part-A, you implemented functions for insertion (and the use of locators), getting the minimum key (and merging trees), and producing a symbolic listing of the structure. In this part, we will complete the remainder of the structure by adding the operations of decrease-key, extract-min, and a few others.

Quake Heap: Please refer to Part-A of the assignment for a general description of the Quake Heap data structure (see Fig. 1. We will employ the same basic elements here. Further details available on quake heap can be found in the CMSC420 Projects page and the Quake Heap lecture notes. The operations from Part-A will be the same as before. This includes the constructor, `clear`, `insert`, `getMaxLevel`, `getMinKey`, and `listHeap`. In this part, you will implement the following additional operations:

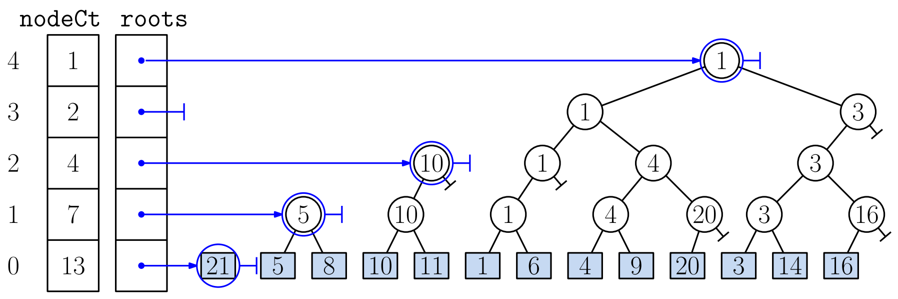


Figure 1: A quake heap storing 13 keys $\{21, 5, 8, 10, \ldots, 16\}$ (values are not shown).

`void decreaseKey(Locator r, Key newKey)`: This decreases the key of the item referenced by locator `r` to have the key value of `newKey`. If `newKey` is strictly larger than the current key for this item, an `Exception` is thrown with the message `"Invalid key for decrease-key"`. Otherwise, the decrease-key operation described in class is performed. (For testing purposes, we would like you to perform the operation even if the new key value is the same as the original key value.)

In class, we presented two methods. One was a simple method running in $O(\log n)$ time and the other was more sophisticated and ran in $O(1)$ time. You may implement the simple method for full credit, and we will leave the sophisticated method as a challenge problem for extra credit points.

The simple method starts with the leaf node `u` specified by the locator `r`. It walks up the tree, always following left-child links. For each node visited (including the initial leaf) the key value is changed to `newKey`. The process stops either when we pop off the top of the tree or when we first visit a parent along a right-child link.

Note that there may be duplicate keys in the heap. For this reason, it is not a good idea to compare keys to determine whether a node is the left child of its parent.

`Value extractMin():` If the heap is nonempty, this function finds the entry with the smallest key value, removes this entry from the heap, and returns the associated value. This is done by the process described in the class notes, which involves first searching to find the root node with the smallest key, performing cuts along its left-most path, merging trees, and quaking. If there are ties for the minimum key, you may extract any one. If the heap is empty, an `Exception` is thrown with the message `"Empty heap"`.

The processes of searching for the smallest key and merging trees are exactly the same as in Part-A. (Remember that tree roots are to be sorted on each level before merging. This is not required by the quake heap, but it useful for testing purposes.) The processes of cutting and quaking are described in the lecture notes.

Note that unlike `getMinKey()`, this function returns the *value* associated with the minimum key, not the key itself.

`int size():` This returns the number of entries in the heap, which is equivalent to the number of leaf nodes in all the trees. This should be answered in $O(1)$ time. The easiest way to implement this is just to maintain a counter, which is incremented whenever entries are inserted and decremented when entries are removed.

`void setQuakeRatio(double newRatio):` This sets the quake ratio from its current value (which is initially 3/4) to the value `newRatio`. If `newRatio` is strictly smaller than 1/2 or strictly greater than 1, an `Exception` is thrown with the message `"Quake ratio is outside valid bounds"`. The current structure is not modified, but in all future instances where the quake operation is performed, this value will be used.

`void setNLevels(int nl):` This sets the number of levels in the quake heap to `nl`. If `nl` is smaller than 1, an `Exception` is thrown with the message `"Attempt to set an invalid number of levels"`. Otherwise, the function sets the number of levels to this value, and adjusts the `roots` and `nodeCt` arrays accordingly.

If `nl` is greater than or equal to the current number of levels, these new levels are added to the structure. The structure is otherwise unchanged. (In particular, we do *not* invoke merge-trees at this time to take advantage of the fact that there are additional levels. The next time that merge-trees would have been invoked, we will take advantage of the new levels.)

On the other hand, if `nl` is smaller than the current number of levels, we remove all nodes in the tree at levels greater than or equal to `nl` (in the same manner as if we were to force a quake were triggered), and convert all the newly exposed nodes at level `nl` $- 1$ to be new root nodes. Then we reduce the number of levels to the new value.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. The only file that you should expect to modify is `QuakeHeap.java`. Use must use the package "`cmsc420_s22`" for all your source files. (This is required for the autgrader to work.) We will provide a driver program that will input a set of commands. You need only implement the data structure and the functions listed above. Here is a portion of the class's public interface (and of course, you will add all the private data and helper functions).

```
package cmsc420_s22;
```

```
import java.util.ArrayList;

public class QuakeHeap<Key extends Comparable<Key>, Value> {

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

    public QuakeHeap(int nLevels) { ... }
    public void clear() { ... }
    public Locator insert(Key x, Value v) { ... }
    public Key getMinKey() throws Exception { ... }
    public int getMaxLevel(Locator r) { ... }
    public ArrayList<String> listHeap() { ... }

    // New functions

    public int size() { ... }
    public void setQuakeRatio(double newRatio) throws Exception { ... }
    public void setNLevels(int nl) throws Exception { ... }
    public Value extractMin() throws Exception { ... }
}
```

Efficiency requirements: As in Part-A, the function `insert()` should run in $O(1)$ time, the function `getMinKey()` should run in time proportional to the number of roots (plus the time needed for sorting each level), and the function `getMaxLevel()` should run in time proportional to the maximum number of levels. For Part-B, the function `size` should run in $O(1)$ time, and `decreaseKey` should run in time proportional to the number of levels.

Testing/Grading: Submissions will be made through Gradescope (you need only upload your modified `QuakeHeap.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 50 points. Of these, 45 points will be purely for input/output correctness as tested by the autograder, and the remaining 5 points will be for clean programming style and the above efficiency requirements.

Challenge Problem: The simple decrease-key function described above takes time proportional to the tree height, which is $O(\log n)$ time. Using the methods described in the lecture notes, implement decrease-key to run in $O(1)$ time.

If you attempt this, add a comment in the first line of your program (or somewhere near the top) so we can check it. Please tell us where your `decreaseKey` function can be found in your source code. For example:

```
// I HAVE ATTEMPTED THE CHALLENGE PROBLEM. SEE DECREASEKEY ON LINE 327
package cmsc420_s22;
...
```

CMSC 420: Spring 2022

Programming Assignment 2: Height-Balanced kd-Trees

Handed out: Tue, Apr 19. Due: **Tue, Apr 26, 11:59pm**. (Submission via Gradescope.)

Overview: In this assignment you will implement a variant of the kd-tree data structure, called a *height-balanced kd-tree* (or `HBkdTree`) to store a set of points in 2-dimensional space. It will support insertion, deletion, and a few other queries.

This data structure borrows ideas from AVL trees, scapegoat trees, and classical point kd-trees. When constructed, a positive integer parameter `maxHeightDifference` is given. Whenever an internal node's two subtrees have heights that differ by more than this value, the subtree rooted at this node is rebuilt[1] into a perfectly balanced tree (in the manner of scapegoat trees).

The data structure is generic and is templated with the point type, which call a *labeled point*. This encapsulates the concept of a 2-dimensional point that is associated with a string, called its *label*. This may be any class that implements the Java interface (which we will provide) called `LabeledPoint2D`. Such an object is a 2-dimensional point, represented by its (x, y)-coordinates and an associated string *label*.

```
public interface LabeledPoint2D {
    public float getX(); // get point's x-coordinate
    public float getY(); // get point's y-coordinate
    public float get(int i); // get point's i-th coordinate (0=x, 1=y)
    public Point2D getPoint2D(); // get the point itself
    public String getLabel(); // get the label
}
```

The `Point2D` object is an enhanced version of the Java built-in `Point2D` object, which we will provide to you.

In our case, the labeled points represent airports, where the (x, y) coordinates are the airports location (think latitude and longitude) and the labels are the 3-letter airport codes (e.g., "`BWI`" for Baltimore-Washington Airport). The individual coordinates (which are `doubles`) can be extracted directly using the functions `getX()` and `getY()`, or `get(i)`, where $i = 0$ for x and $i = 1$ for y.

Your wrapped kd-tree will be templated with one type, which we will call `LPoint` (for "labeled point"). For example, your file `HBkdTree` will contain the following public class:

```
public class HBkdTree<LPoint extends LabeledPoint2D> { ... }
```

Height-Balanced kd-Tree: Recall that a point kd-tree is a data structure based on a hierarchical decomposition of space through the use of axis-orthogonal splits. A *height-balanced kd-tree* imposes the additional requirement that for every internal node, the heights of its two subtrees

[1] You might wonder why we don't just apply rotations as we did with AVL trees. The issue is that rotations are a one-dimensional operation and they do not make sense in the context of multidimensional structures like kd-trees.

can differ by at most a user-specified integer parameter `maxHeightDifference`, which is at least 1. The insertion and deletion processes are exactly the same as given in the lecture on kd-trees (see the latex lecture notes for Lecture 13, which has all the details spelled out), but when inserting a new point, the cutting dimension is selected based on the shape of the current cell. We select the cutting dimension so that we split the longer side of the current cell. More formally, if its width (along x) is greater than or equal to its height (along y) the cutting dimension is 0 (x or vertical) and otherwise it is 1 (y or horizontal). Note that ties are broken in favor of vertical cuts.

For example, consider the insertion of `ATL` in Fig. 1. When we fall out of the tree (along the left child link from `ORD`), the cell associated with this null pointer is the rectangle whose lower-left corner is $(0, 4)$ and whose upper-right corner is $(2, 8)$.) Since this rectangle is taller than wide, we cut horizontally, thus setting the cutting dimension of the new node to 1.

After a point has been inserted into or deleted from the tree, we walk backwards upward along the search path (exactly has we would do if this were an AVL), updating the heights as we go. Whenever we reach a node `p` where the heights of its two subtrees differ by more than `maxHeightDifference`, we completely rebuild the subtree rooted at `p`. We first traverse the subtree rooted at `p` and store all the labeled points of this subtree in a list (e.g., a Java `ArrayList`). Given this list, the subtree is rebuilt by the following recursive process (see Fig. 1):

Basis: If the list is empty, return `null`. Otherwise, continue with the following steps.

Cutting Dimension: Let `cell` denote the cell associated with the current node. As with insertion, we select the cutting dimension so that it splits the longer side of `cell`.

Sort: Sort the points according to the cutting dimension. If the cutting dimension is x, sort the points in increasing order first by x and break ties by sorting in increasing order y. If the cutting dimension is y, then sort first by y with ties broken by x.

Split and Recurse: Letting k denote the size of the list (and assuming as usual that entries are indexed from 0 to $k - 1$), define the median element to be point at index $m \leftarrow \lfloor k/2 \rfloor$. Recursively build a balanced tree on the left-side sublist of entries with indices 0 through $m-1$, and recursively build a balanced tree on the right-side sublist of entries with indices $m + 1$ through $k - 1$. Join these two subtrees under a node whose point is the median point, and whose cutting dimension is as chosen above. Return this tree.

Unlike scapegoat trees (where each operation can trigger at most one rebuild), we continue all the way up to the root, updating the heights as we go and checking the height difference condition. This may trigger further rebuilds. (See Fig. 1 for an example.)

Requirements: Your program will implement the following functions for the `HBkdTree`. 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. As part of the skeleton code, we will provide you with the `LabeledPoint2D` interface, and two useful classes, `Point2D` and `Rectangle2D`. (If you wish to modify these objects, do not alter them. Instead, create your own copy, say `MyPoint2D`, and make modifications there.)

`HBkdTree(int maxHeightDifference, Rectangle2D bbox)`: This constructs a new `HBkdTree` with the given max height difference and the given axis aligned bounding box.

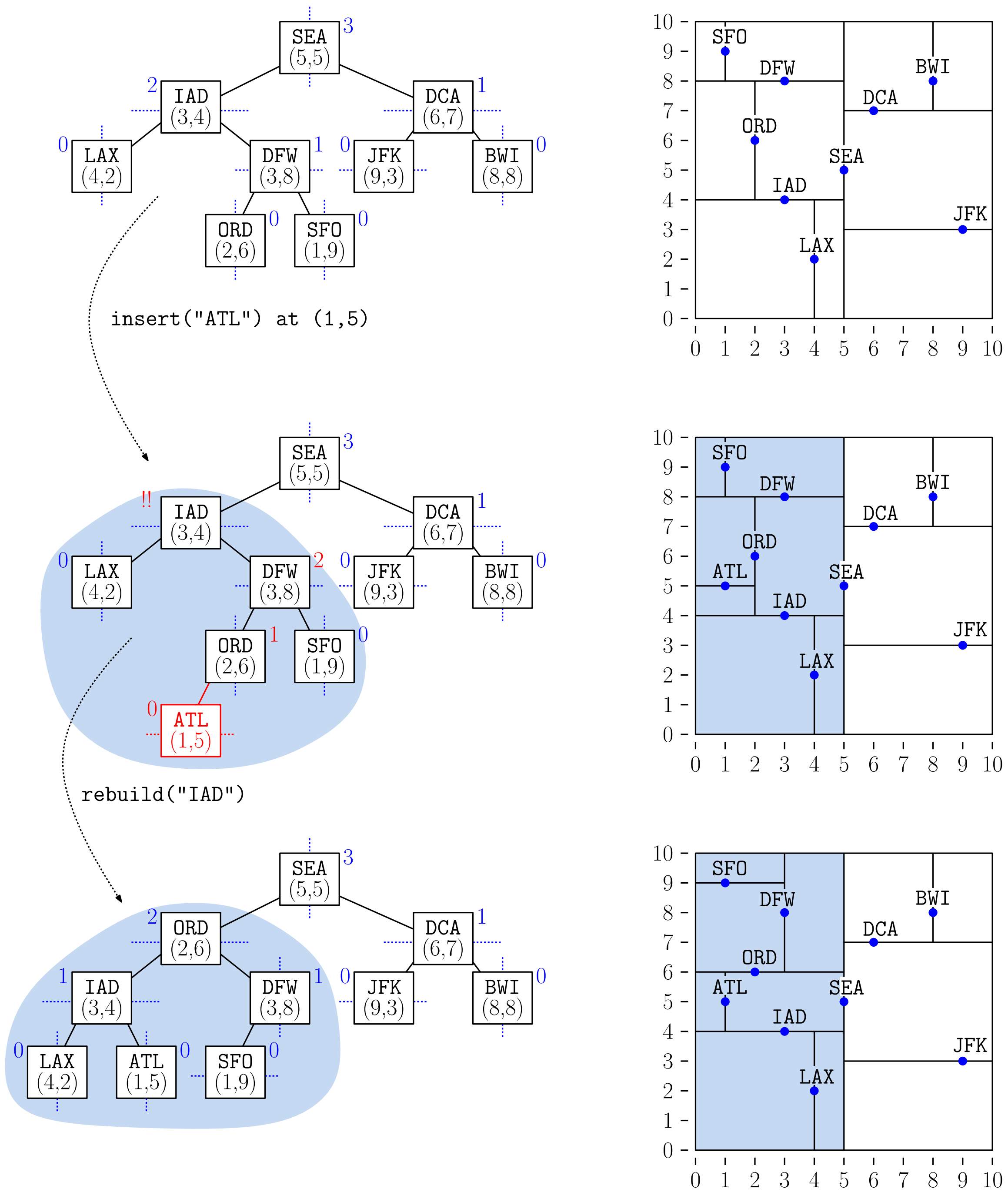


Figure 1: Let `maxHeightDifference = 1`. Suppose we insert `ATL` at $(1,5)$. This is inserted as the left child of `ORD`. On returning from the recursive calls, we update the node heights at `ORD`, `DFW`, and `IAD`. At `IAD`,the absolute height difference in our left and right subtrees is $2 - 0 = 2$, which exceeds `maxHeightDifference`. We rebuild this entire subtree highlighted in blue. Since the cell (shaded in blue) is taller than wide, we cut horizontally. We sort along y (yielding $\langle$`LAX`, `IAD`, `ATL`, `ORD`, `DFW`, `SFO`$\rangle$) and split about the median `ORD`. We recursively build the other subtrees similarly. We continue back up the root, updating heights, but no further rebuilds are needed.

`LPoint find(Point2D pt)`: Given an (unlabeled) point, determine whether it exists within the tree, and if so return the associated labeled point. Otherwise, return `null`.

`void insert(LPoint pt)`: Inserts point given labeled point in the tree (and performs rebuilding if necessary, as described above). If the point lies outside the bounding box, throw an `Exception` with the error message `"Attempt to insert a point outside bounding box"`. If a point with the same coordinates (and possibly different label) exists in the tree, throw an `Exception` with the message `"Attempt to insert a duplicate point"`. Otherwise, apply the insertion and rebuilding process described above.

`void delete(Point2D pt) throws Exception`: Given an (unlabeled) point, this deletes the point of the tree having the same coordinates (and performs rebuilding if necessary, as described above). If there is no such point, it throws an `Exception` with the error message `"Attempt to delete a nonexistent point"`. The deletion process is the same as described in the Lecture 13 notes. (In particular, the process by which the replacement nodes are selected is the same as given in the lecture notes.)

Update (4/20): In the utility function `findMin`, which is used to find the replacement node, if there are ties for the point with the smallest ith coordinate, break the ties by taking the point with the smallest other coordinate (that is, coordinate $1 - i$).

Update (4/20): The one change is that on returning up along the search path node heights are to be updated, and whenever a node is found to fail the height difference criteria, its subtree is rebuilt. Generally, a deletion may result in multiple replacements. The balance condition testing and rebuilding is applied only *after* the standard deletion process is completely finished. Balance checking commences at the final leaf node whose deletion terminates the standard kd-tree deletion process. This is very easy to code. Simply code the deletion as given in the lecture notes, but just prior to returning from the deletion helper (that is, just prior to the line "`return p`" from the lecture notes), update the current node's height, check the height difference, and trigger rebuilding if needed. If the tree is rebuilt, return a pointer to the newly rebuilt tree.

Update (4/20): Note, by the way that due to replacement, many cells in the tree can change shape. (In particular, these are the cells that are incident on the splitting line through the deleting point.) As a result, some cells that were taller have switched to being wider and vice versa. However, you should not alter the cutting dimension for any of the nodes. Once the cutting dimension of a node has been set, it should remain unchanged until the node is deleted or it has been part of a rebuilding. (As a consequence of this, you should *not* store each node's cell as a member of the kd-tree node, since otherwise updating these cells will take too much time. Instead, you should compute cells on the fly as you traverse the tree.)

`ArrayList<String> getPreorderList()`: This operation generates a preorder enumeration of the nodes in the tree. This is represented as a Java `ArrayList` of type `String`, with one entry per node. You will probably implement this by writing a recursive helper function that starts at the root. When it visits a node `p`, it does the following. If `p == null`, then generate the string `"[]"` and return. Otherwise, generate the following string and recursively invoke the procedure on the left and right children. Depending on whether the cutting dimension is x or y, this generates either:

```
"(x=" + cutVal + " ht=" + height + ")␣" + point.toString()
```

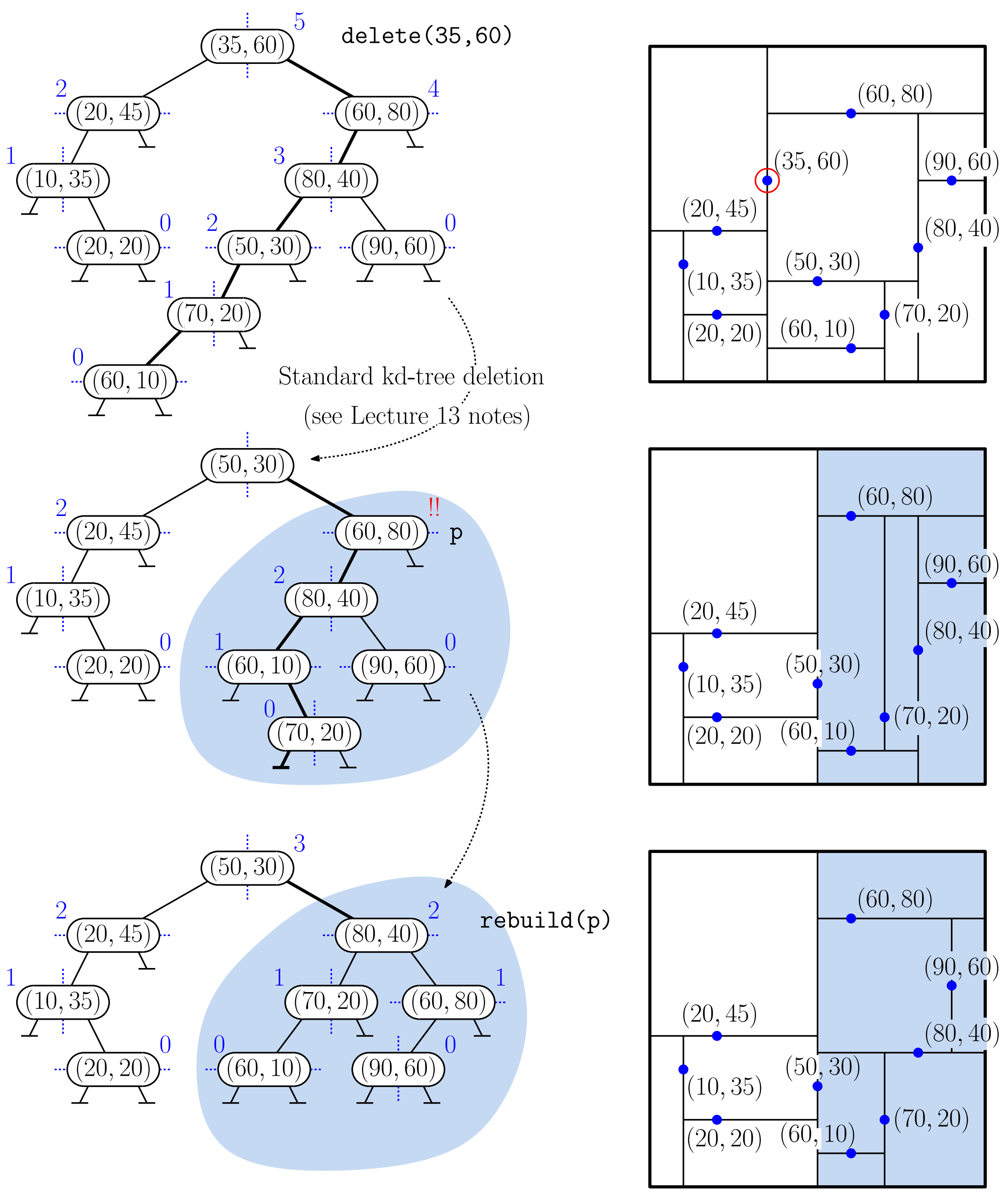


Figure 2: Let `maxHeightDifference = 1`. (Note that this tree is not valid, since it fails the height-difference condition at many nodes. We have chosen it to match the example from Lecture 13.) Suppose we delete the point `(35,60)`. We first perform the standard kd-tree deletion as described in Lecture 13. Note that when the replacement point `(50,30)` is copied to the root, the root's cutting dimension does not change. At the end of the process, on returning from the recursive calls, we update the node heights at `(70,20)` (now 0), `(60,10)` (now 1), and `(80,40)` (now 2). At `(60,80)`, the absolute height difference in the left and right subtrees is $2 - (-1) = 3$, which exceeds `maxHeightDifference`. We rebuild this entire subtree highlighted in blue. Since the cell (shaded in blue) is taller than wide, we cut horizontally. We sort along y (yielding ⟨`(60,10)`, `(70,20)`, `(80,40)`, `(90,60)`, `(60,80)`⟩) and split about the median `(80,40)`. We recursively build the other subtrees similarly. We continue back up the root, updating heights, but no further rebuilds are needed.

```
"(y=" + cutVal + " ht=" + height + ")␣" + point.toString()
```

where `cutVal` is the cutting value for this node (that is, the coordinate of the node's point associated with the cutting dimension), `height` is the height of the subtree rooted at this node, and `point.toString()` invokes the `toString()` method for the point stored in this node. (This function will be provided to you as part of our skeleton code.)

Here is example of what this would look like for the tree at the top of Fig. 1.

```
(x=5.0 ht=3) SEA: (5.0,5.0)
(y=4.0 ht=2) IAD: (3.0,4.0)
(x=4.0 ht=0) LAX: (4.0,2.0)
[]
[]
(y=8.0 ht=1) DFW: (3.0,8.0)
(x=2.0 ht=0) ORD: (2.0,6.0)
[]
[]
(x=1.0 ht=0) SFO: (1.0,9.0)
[]
[]
(y=7.0 ht=1) DCA: (6.0,7.0)
(y=3.0 ht=0) JFK: (9.0,3.0)
[]
[]
(x=8.0 ht=0) BWI: (8.0,8.0)
[]
[]
```

Note that our autograder is sensitive to both case and whitespace.

`ArrayList<LPoint> orthogRangeReport(Rectangle2D query)`: This function performs an orthogonal range reporting query. It is given an axis-aligned rectangle `query` and it returns a Java `ArrayList` containing the points lying within this rectangle. (You may find it useful to use the function from class `Rectangle2D`, such as `contains`, `leftPart`, and `rightPart`.) The order in which elements appear in the final list does not matter. We will sort the list before outputting it.

`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 at the top of Fig. 1, this would return 9.

`void setHeightDifference(int newDiff)`: `Update (4/19)`: You no longer need to implement this operation.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You should replace the `HBkdTree.java` file with your own, and you should add the implementation of the above functions to `HBkdTree.java`. You should not modify any of the other files, but you can add new files of your own.

As mentioned above, you should not modify `Point2D` or `Rectangle2D` (since our testing functions use these), but you can create copies and make modifications to these copies if you like.

You must use the package "`cmsc420_s22`" for all your source files. (This is required for the autgrader to work.) As usual, we will provide a driver program (`Tester.java` and `CommandHandler.java`) that will input a set of commands. Here is a portion of the class's public interface (and of course, you will add all the private data and helper functions). You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

```
package cmsc420_s22;

import java.util.ArrayList;

public class HBkdTree<LPoint extends LabeledPoint2D> {

    public HBkdTree(int maxHeightDifference, Rectangle2D bbox) { /* ... */ }
    public LPoint find(Point2D pt) { /* ... */ return null; }
    public void insert(LPoint pt) throws Exception { /* ... */ }
    public void delete(Point2D pt) throws Exception { /* ... */ }
    // ... and so on
}
```

Efficiency requirements: `Update (4/19):` Excluding the time for rebuilding, the operations `find`, `insert`, and `delete` must run in time proportional to the tree height. (Because of rebalancing, the tree height will be $O(\log n)$.) The operation `orthogRangeReport` should be efficient in the sense that it does not waste time making recursive calls into subtrees whose cell does not overlap the query range. For this reason, the helper function for this operation will need to test the node's cell against the query range.

The operation `size` should run in constant time. (This is best handled by maintaining a separate counter that keeps track of the number of points currrently in the structure.)

Testing/Grading: `Update (4/19):` We will use the standard Gradescope-based grading process that we have used in previous assignments.

CMSC 420: Spring 2022

Programming Assignment 3: Euclidean Minimum Spanning Trees

Handed out: Thu, Apr 28. Due: TBD.

Overview: The principal purpose of this assignment is to combine some of the data structures we have seen to compute a fundamental geometric structure, called the *Euclidean minimum spanning tree* (EMST).

Let us first recall the graph-based minimum spanning tree. Given a connected, undirected graph $G = (V, E)$ with vertex set V and edge set E, assume that each edge $(u, v) \in E$ has an associated weight $w(u, v)$. A *spanning tree* is a subset of edges of E so that the subgraph of G induced by these edges is connected, acyclic, and includes all the vertices of G. The *weight* of the spanning tree is the sum of its edge weights. The *minimum spanning tree* (MST) is the spanning tree of minimum weight. (If there are multiple spanning trees of the same minimum weight, any of them is a valid answer to the MST problem.)

We can define the MST problem in a geometric context. We are given a set $P = \{p_1, \ldots, p_n\}$ in $\mathbb{R}^2$. This set implicitly defines a graph, called the *Euclidean graph*, whose vertex set is P, whose edges consist of all (unordered) pairs (p_i, p_j), and where the weight of an edge is the Euclidean distance between these points. Note that $|E| = \binom{n}{2}$, so G has $O(n^2)$ size. The *Euclidean minimum spanning tree* (EMST) is the MST of P's Euclidean graph (see Fig. 1).

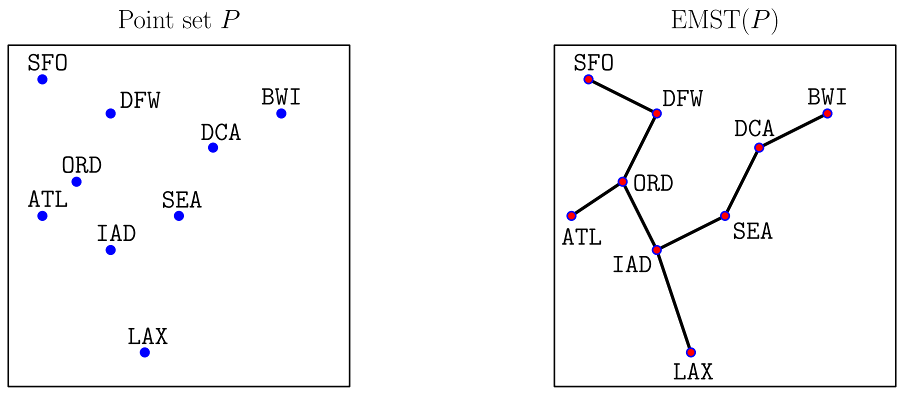


Figure 1: Euclidean minimum spanning tree.

In this assignment, you will implement a class `EMSTree`, whose principal purpose is to compute the EMST of a set of points in the plane efficiently. This class with provide functions for adding points to the set, clearing the set, and computing the EMST of the set. As in our previous assignment, it is templated with the point type, which as in the previous assignment will be labeled point:

```
public class EMSTree<LPoint extends LabeledPoint2D>
```

The EMST will be computed by a geometric variant of Prim's MST algorithm. In order to achieve efficiency, your program will need to implement a *kd-tree* (e.g., your `HBkdTree` from Programming Assignment 2) augmented with a function for answering *nearest-neighbor queries* and a *priority queue* (e.g., your `QuakeHeap` from Programming Assignment 1).

Points and Distances: As in the previous assignment, the points in our assignment will be any class that implements the `LabeledPoint2D` interface, such as the `Airport` class.

The EMST has the same structure whether defined in terms of Euclidean distances or squared Euclidean distances.[1] By avoiding square roots, it will be both more efficient and more accurate. Letting $p_i = (x_i, y_i)$, the weight of the edge between them will be the *squared Euclidean distance* defined as

$$d(p_i, p_j) \;=\; (x_i - x_j)^2 + (y_i - y_j)^2.$$

To assist you, we have added a member function `double distanceSq(Point2D q)` to the class `Point2D`, which computes the squared distance between the current point and another point `q`. For example, the squared distance between points `p` and `q` can be computed as `p.distanceSq(q)`. This function has the feature that if the argument is `null`, it returns `Double.POSITIVE_INFINITY`.)

Prim's Algorithm: Prim's MST algorithm is given a starting vertex s_0, and builds the spanning tree by repeatedly adding the point that lies outside the tree, but is closest to some point of the tree. A new point is added with each iteration. Let S denote the set of points that are currently in the spanning tree (see the shaded region in Fig. 2). Initially $S = \{s_0\}$ and the algorithm terminates when all the points of P are in S.

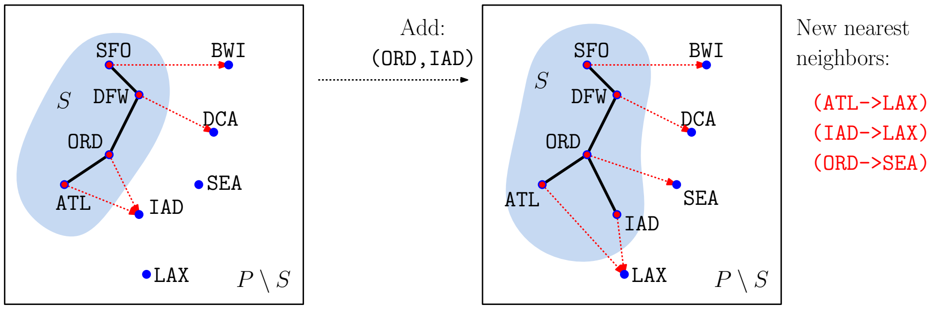


Figure 2: Originally, $S = \{$`SFO`, `DFW`, `ORD`, `ATL`$\}$ with the nearest-neighbor pairs `(SFO,BWI)`, `(DFW,DCA)`, `(ORD,IAD)`, and `(ATL,IAD)`. The closest of these, `(ORD,IAD)` is added to the EMST, `IAD` is added to S, and new nearest neighbors `(ATL,IAD)`, `(ORD,IAD)`, and `(IAD,SEA)` are computed.

Each point $p_i \in S$ computes its nearest point in the complement set $P \setminus S$ (indicated by red broken lines in the figure). Let's call these the *nearest neighbor pairs*. Let (p_i, p_j) be the closest of all the nearest neighbors. In the next iteration, this edge is added to the spanning tree, p_j is added to S, and we need to update the nearest neighbors. This will certainly include p_i, p_j, and any it will also include any other points of S whose nearest neighbor was p_j (see Fig. 2). The process is repeated $n - 1$ times, after which all the points have been added to the spanning tree.

Implementing this algorithm efficiently will involve a number of data structures.

[1]This fact is by no means obvious. In fact it holds for any strictly monotone function of distance.

List: to store the edges of the spanning tree. This can be implemented using a Java `ArrayList` or `LinkedList`. Since Java has no primitive type for storing a pair, we will provide you with a simple generic class `Pair` in the skeleton code. You can create a new pair (`new Pair(a,b)`), access its components (`getFirst()`, `getSecond()`), and test equality (`pair1.equals(pair2)`).

Set: to maintain the points of S. This must support the operations `insert` and `contains` (to test membership). This can be done using a Java `HashSet`.

Spatial index: to store the points of $P \setminus S$ waiting to be inserted into the EMST. This must support the operations of `insert`, `delete`, and `nearestNeighbor`. In a nearest-neighbor query, we are given a query point q, and the answer is the closest point in the tree to q. This can be done using your `HBkdTree` from Programming Assignment 2, and adding a nearest-neighbor function. (See Lecture 14 latex notes for details on how to do this.)

Priority Queue: to store the nearest-neighbor pairs ordered by their distance. Each entry stores the associated pair of points (e.g., from Fig. 2, `(SFO, BWI)` is one of these pairs, and the associated key is the squared distance $d($`SFO`$,$`BWI`$)$.) This can be implemented using your `QuakeHeap` data structure from Programming Assignment 1.

Initially, all the points except the start point s_0 are inserted into a kd-tree, we compute s_0's nearest neighbor, and add this pair to the initially empty priority queue. Then we each iteration, we extract the closest pair (p_i, p_j) from the priority queue, add this edge to the spanning tree edge list, add p_j to the set S, remove p_j from the kd-tree, and finally update the nearest neighbor pairs and insert them in the priority queue based on squared distances.

Dependents Lists: The final question that we need to answer is how to determine which points of S need their nearest neighbors updated at the end of each iteration. Certainly, we need to do this for the new point p_j. In addition, every point $p_k \in S$ that depends on p_j as its nearest neighbor must also be updated.

We say that p_k *depends* on $p_j \in P \setminus S$ if p_j is the nearest neighbor of p_k. The set of all points in S that depend on p_j constitute its *dependents list*, denoted $\text{dep}(p_j)$. Whenever a point $p_j \in P \setminus S$ is added to the spanning tree, we need to update the nearest neighbor of p_j and all the members of $\text{dep}(p_j)$. For example, for the situation shown on the right side of Fig. 2, we have the following. (Note that the points of S do not need dependents lists.)

Point (p)	**Dependency list (dep(p))**
`BWI`	{`SFO`}
`DCA`	{`DFW`}
`SEA`	{}
`IAD`	{`ORD, ATL`}
`LAX`	{}

Each such list can be stored, for example, as a Java `ArrayList`. There is one for each point of $P \setminus S$. Initially, all of these lists are empty. Whenever we add an entry (p_i, p_j) is added to the priority queue, we add p_i to $\text{dep}(p_j)$ as well.

So, when p_j is added to the spanning tree, we iterate through the members of $\text{dep}(p_j)$ and compute its new nearest neighbor. But now the question emerges, how to we access this dependency list efficiently? We can do this by creating one more data structure:

Hash Map: to store the points of $P \setminus S$. Each element of the map is associated with its dependents list. This can be done using a Java `HashMap`.

In the parlance of this assignment, each point is a labeled point, say `LPoint`. An array list of points is of type `ArrayList<LPoint>`. A hash map that maps a point to its associated dependents list is therefore `HashMap<LPoint, ArrayList<LPoint>>`. If we create such an object, called, say, `dependents`, and given a point `pt`, we can access dep(pt) with: `ArrayList<LPoint> dep = dependents.get(pt)`.

Redundant Priority Queue Entries: There is a subtle issue with our algorithm as described. Whenever we compute a new nearest-neighbor pair (p_i, p_j) to add to our priority queue, it is possible that there was already a nearest neighbor pair (p_i, p'_j) in the queue. Ideally, we should remove this from the priority queue, but most priority queues (including our QuakeHeap) do not support deletion.

There is an easy fix, however. The only reason that one pair (p_i, p_j) overrides another (p_i, p'_j) is that p'_j was added to the spanning tree. Whenever we remove a pair (p_i, p'_j) from the priority queue, we check whether p'_j is in the tree. We can do this efficiently by accessing our set data structure for S. If so, we ignore this edge and go on to the next one.

Summary: That's a lot of data structures! But this is typical of many efficient algorithms. We need to access the various structures as efficiently as possible, and the best way to do this is to store them in an appropriate data structure. Fig. 3 demonstrates the iterations of the algorithm. For further information about the algorithm, consult the lecture notes.

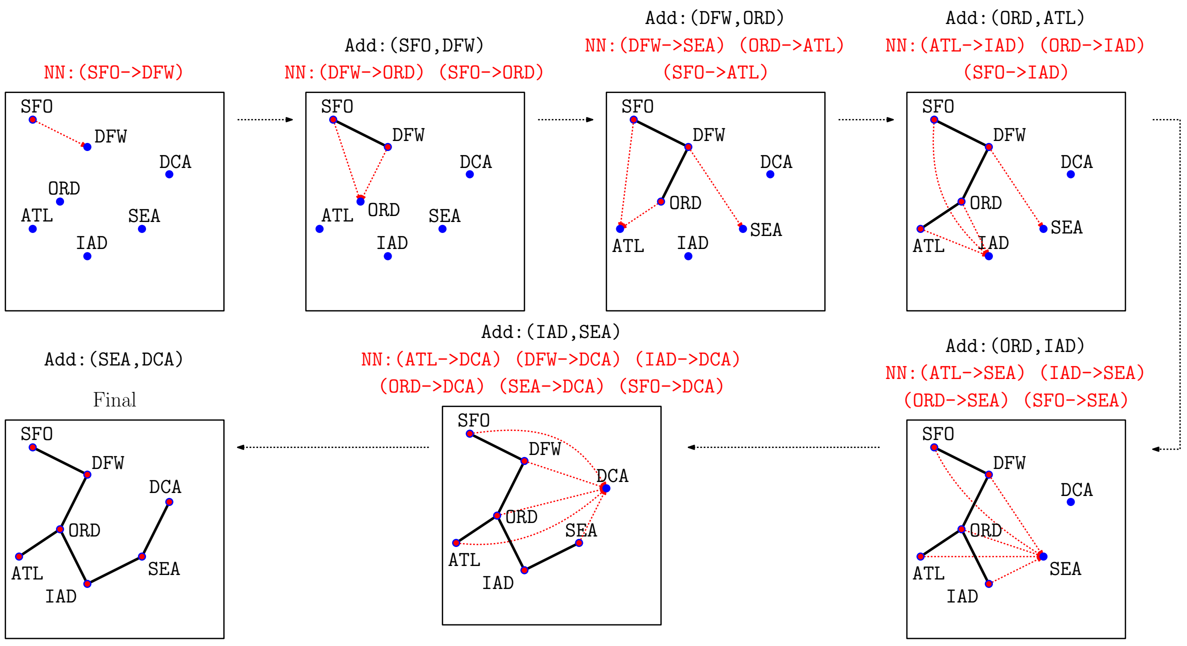


Figure 3: Prim's EMST algorithm walkthrough.

Requirements: Your program will implement the following functions for the `EMSTree`. While you can implement the operations internally however you like (subject to the style and efficiency requirements given below), the following function signatures should not be altered. As part of the skeleton code, we will provide you with the `LabeledPoint2D` interface, and two useful classes, `Point2D` and `Rectangle2D`. (If you wish to modify these objects, do not alter them. Instead, create your own copy, say `MyPoint2D`, and make modifications there.)

`EMSTree(Rectangle2D bbox):` This initializes by creating the basic objects (set, kd-tree, heap, hashmap) needed by the EMST algorithm. The bounding box can be passed into your constructor for your kd-tree. You may set the other parameters for the kd-tree and quakeheap as you like. (We would suggest using a max-height-difference of 2 for your kd-tree and a number of levels of 10 for your quakeheap.)

`void addPoint(LPoint pt):` This inserts a point into the set P of points that will form the EMST. The EMST is *not* constructed at this point, and no error checking is done.

We would recommend doing two things. First, insert `pt` into your set of points P. Second, create a dependents list for this point. If you are using a hash map of array lists for dependents, this can be done with `dependents.put(pt, new ArrayList<LPoint>())`.

`void clear():` This removes all the points from your points set P, clears the edge list for the EMST. You can also clear the kd-tree, the priority queue, and the hash map used for the dependents lists.

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

`ArrayList<String> buildEMST(LPoint start) throws Exception:` Builds the EMST for the current point set P, where `start` is the starting vertex (called s_0 above). For the purposes of testing, it returns a Java `ArrayList` of strings (described below) that provides a summary of the construction process.

This function checks for the validity of the point set. If any point lies outside the bounding box, it throws an `Exception` with the error message `"Attempt to insert a point outside bounding box"`. If there are two or more points with the same coordinates, it throws an `Exception` with the message `"Attempt to insert a duplicate point"`. (Update: 4/30) If multiple points fail these conditions, the first to be added triggers the exception. If an exception is thrown, the EMST that results is empty.

This function computes the entire EMST. It begins by clearing out the point set S, the list of tree edges, the kd-tree, and the priority queue. It clears the dependents lists for all the points. It inserts all the points into the kd-tree except the start point. And then it starts Prim's algorithm.

`ArrayList<String> listTree():` This lists the edges of the EMST. If the tree has not yet been built (e.g., points were added but `buildEMST` was not called) then it returns an empty list. Otherwise, it returns the edges in the order that they were added by Prim's algorithm. Assuming that you represent your each edges as `Pair<LPoint>`, the edge `e` could be listed using:

```
"(" + e.getFirst().getLabel() + "," + e.getSecond().getLabel() + ")"
```

For example, the tree shown in Fig. 3 would generate an array list with the following 6 entries

```
"(SFO,DFW)" "(DFW,ORD)" "(ORD,ATL)" "(ORD,IAD)" "(IAD,SEA)" "(SEA,DCA)"
```

Summary of `buildEMST`: The array list returned by your `buildEMST` function (see above) summarizes the algorithm's processing. It contains a line for every new point inserted into the EMST. The first line just gives the pair consisting of the start vertex (`start`) and its nearest-neighbor (`nn`),

```
"new-nn: (" + start.getLabel() + "->" + nn.getLabel() + ")"
```

(This is the first entry that is placed in your priority queue.)

For each remaining point added to the tree, it prints the newly added edge. If you store your edge as a `Pair` object, you can just rely on the `toString` method provided by this object. For example, if `edge` denotes the newly created pair, of type `Pair<LPoint>`, the added edge is reported with:

```
"add: " + edge + " new-nn:"
```

Following this, on the same line, you will output all the updated nearest-neighbor pairs. For each new nearest-neighbor pair consisting of a point `pt` from S and its nearest neighbor `nn` from $P \setminus S$, the output consists of the labels of these two points:

```
"(" + pt.getLabel() + "->" + nn.getLabel() + ")"
```

For example, when we added the edge `(DFW,ORD)` in Fig. 3, we generated the following new nearest-neighbor pairs:

```
(DFW->SEA) (ORD->ATL) (SFO->ATL)
```

There is one final issue. For testing purposes, we need your nearest-neighbor list to match ours exactly. For this reason, you should sort the entries of this list. You can do this easily using `Collections.sort`. (There is no need to design a special comparator.)

Below, we given an example of the seven entries in the array list returned by `buildEMST` on the example from Fig. 3. (The "`... more, omitted`" is not really in the string. We just ran out of space!)

```
new-nn: (SFO->DFW)
add: (SFO:(12.0,88.0)--DFW:(30.0,84.0)) new-nn: (DFW->ORD) (SFO->ORD)
add: (DFW:(30.0,84.0)--ORD:(19.0,58.0)) new-nn: (DFW->SEA) (ORD->ATL) (SFO->ATL)
add: (ORD:(19.0,58.0)--ATL:(5.0,51.0)) new-nn: (ATL->IAD) (ORD->IAD) (SFO->IAD)
add: (ORD:(19.0,58.0)--IAD:(32.0,41.0)) new-nn: (ATL->SEA) (IAD->SEA) (ORD->SEA) (SFO->SEA)
add: (IAD:(32.0,41.0)--SEA:(51.0,53.0)) new-nn: (ATL->DCA) (DFW->DCA) ... more, omitted
add: (SEA:(51.0,53.0)--DCA:(65.0,68.0)) new-nn:
```

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You should replace the `EMSTree.java` file with your own. We have provided a number of utility objects for you, include the `Point2D`, `Rectangle2D`, and `Airport` from the previous assignment. We have also provided `Pair` for storing edges.

You must use the package "`cmsc420_s22`" for all your source files. (This is required for the autgrader to work.) As usual, we will provide a driver program (`Tester.java` and `CommandHandler.java`) that will input a set of commands. Here is a portion of the class's public interface (and of course, you will add all the private data and helper functions). You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

```
package cmsc420_s22;

import java.util.ArrayList;

public class EMSTree<LPoint extends LabeledPoint2D> {

    public EMSTree(Rectangle2D bbox) { ... }
    public void addPoint(LPoint pt) { ... }
    public void clear() { ... }
    public int size() { ... }
    public ArrayList<String> buildEMST(LPoint start) throws Exception { ... }
    public ArrayList<String> listEMST() { ... }
    // ... and so on
}
```

Hints: (Update: 4/30) Here are some of the data objects that were referred to in the above descriptions and how they might be implemented. These are best implemented as private members of your `EMSTree` structure. (The following are only suggestions, and you may implement these how you, subject to the efficiency requirements below.)

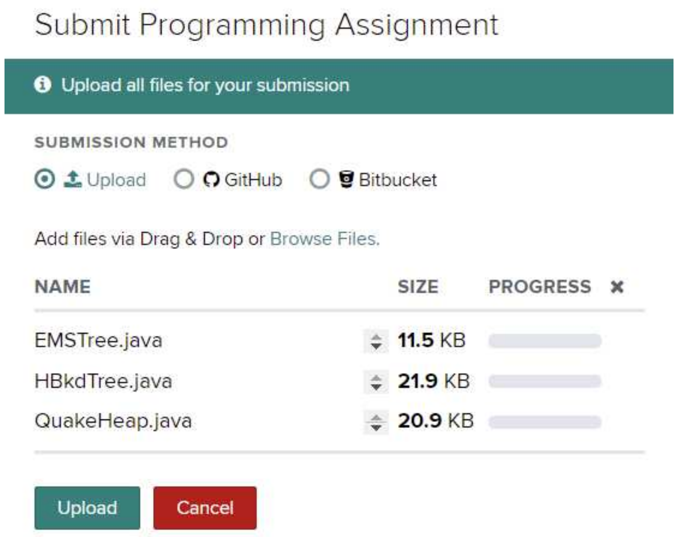
`ArrayList<LPoint>`	`pointList`	All the points (P)
`HashSet<LPoint>`	`inEMST`	Subset of points in the tree (S)
`ArrayList<Pair<LPoint>>`	`edgeList`	List of edges in the EMST
`HBkdTree<LPoint>`	`kdTree`	The kd-tree for $P \setminus S$
`QuakeHeap<Double, Pair<LPoint>>`	`heap`	priority queue of NN pairs
`HashMap<LPoint, ArrayList<LPoint>>`	`dependents`	dependents lists

We will provide the same objects as in Programming Assignment 2, including `Point2D`, `Rectangle2D`, and a simple class `Pair` for storing pairs. Although we have included skeletons of `QuakeHeap.java` and `HBkdTree.java` in the skeleton code, you can replace them with any objects you wish.

Efficiency requirements: (Update 4/30) While you are encouraged to use your own QuakeHeap and HBkdTree data structures, you are allowed to use any comparably efficient structure (e.g., you may use Java's built-in `PriorityQueue` data structure). You may substitute other structures for the ones we have recommended (e.g., `HashSet`, `HashMap`, `ArrayList`) provided that the required operations can be performed efficiently, say in $O(\log n)$ time.

Testing/Grading: (Update 4/30) We will use the standard Gradescope-based grading process that we have used in previous assignments.

The autograder will provide files `Point2D.java`, `LabeledPoint2D.java`, `Rectangle2D.java`, `Pair.java`, `Tester.java`, and `CommandHandler.java`. You will need to upload all other files as needed by your program. At a minimum, this will consist of `EMSTree.java`. If you use any other files, such as `QuakeHeap.java` and `HBkdTree.java`, these will be uploaded as well. (Update 5/1) An example of what your Gradescope submission window might look like is shown in Fig. 4


Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

[✓ Upload] GitHub Bitbucket

Add files via Drag & Drop or Browse Files.

NAME	SIZE	PROGRESS
EMSTree.java	11.5 KB	
HBkdTree.java	21.9 KB	
QuakeHeap.java	20.9 KB	

Figure 4: Possible Gradescope submission window.

CMSC 420: Spring 2022

Homework 1: Basic Data Structures and Trees

Handed out Tue, Feb 8. Due at **11:59pm, Tue, Feb 15**. Point values given with each problem may vary. **Please see the notes at the end about submission instructions**.

Problem 1. (25 points) Answer the following questions involving the rooted trees shown in Fig. 1.

(a) (4 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the "first-child/next-sibling" form.

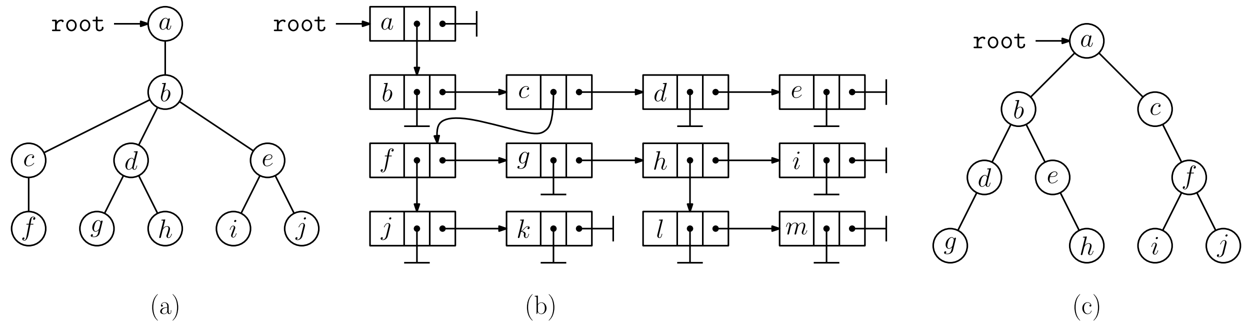


Figure 1: Rooted trees.

(b) (4 points) Consider the rooted tree of Fig. 1(b) represented in the "first-child/next-sibling" form. Draw a figure showing the equivalent rooted tree.

(c) (6 points) List the nodes of the tree of Fig. 1(a) in *preorder*. List them in *postorder*.

(d) (9 points) List the nodes of the tree of Fig. 1(c) in *preorder*. List them in *inorder*. List them in *postorder*.

(e) (2 points) Consider the binary tree of Fig. 1(c). Draw a figure showing the tree with inorder threads (as in Fig. 7 from the latex lecture notes for Lecture 3).

Problem 2. (5 points) Present pseudocode for a procedure `int getHeight(Node root)`, which is given the root of a tree represented using the first-child/next-sibling representation, and returns the height of the tree. For full credit your procedure should run in time proportional to the number of nodes in the tree. (For example, given the tree shown in Fig. 1(b), your function would return three.)

Give a short explanation in English how your procedure works. **Hint:** Use recursion.

Problem 3. (5 points) You have a binary trees in which each node, in addition to having links `left`, `right`, has a link `parent`, which points to the node's parent (and note that `root.parent == null`).

Present pseudocode for a function `Node inorderSuccessor(Node p)`, which returns `p`'s inorder successor, that is, the node that follows `p` in an inorder traversal. If `p` is the last node in the inorder traversal, your function should return `null`. (For example, given the tree shown in Fig. 1(c), `inorderSuccessor(c)` would return the node labeled "`i`" and `inorderSuccessor(h)` would return the node labeled "`a`".)

Give a short explanation in English how your procedure works.

Hint: You should **not** assume that this is a binary search tree. If you want to know whether a node `p` is the left or right child of its parent, you can do "`if (p == p.parent.left)`". Of course, beware of dereferencing `null` pointers.

Problem 4. (5 points) Suppose that you have a rooted tree, where all the leaves are at the same depth. We partition the nodes of the tree into levels as follows. The leaves are at level 0, their parents are at level 1, their grandparents are at level 2, and so on up to the root, which is at some level L (see Fig. 2). Let n denote the number of leaf nodes (all at level 0), and generally, for $0 \leq i \leq L$, let n_i denote the number of nodes on level i of this tree.

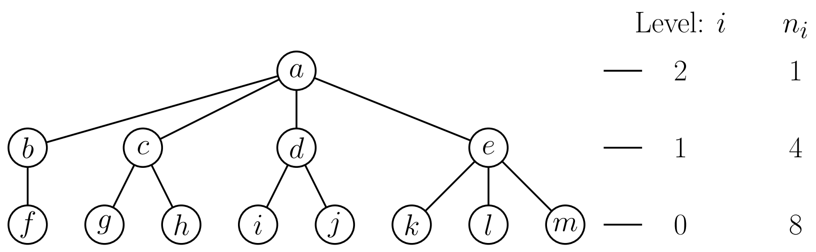


Figure 2: A tree with maximum level $L = 2$.

Suppose that the number of nodes in each level decreases by at least some constant factor, that is, suppose that there is a fixed real number $0 < \alpha < 1$ (which does not depend on n or L) such that $n_{i+1} \leq \alpha n_i$, for $0 \leq i < L$. Prove that there exists a constant c (depending on α) such that $L \leq c \lg n$. (Recall that lg means logarithm base 2.) You should derive the smallest value of c such that this holds.

Hint: If you have difficulty solving this, you can solve the following more concrete version for half credit. Suppose that the number of nodes in each level decreases by at least one third, that is, $n_{i+1} \leq n_i/3$. Prove that $L \leq (\lg n)/(\lg 3)$. (Recall that lg means logarithm base 2.)

Problem 5. (10 points) In this problem, we will consider modification and generalization of the amortized analysis of the dynamic stack algorithm from Lecture 2. We will make two changes: (1) we will slightly change the algorithm and cost model when expanding the stack, and (2) we will allow the stack to contract when the number of elements gets too small.

Throughout, let n denote the number of elements in the stack, and let m denote the size of the current array. Here is a formal description of our new dynamic stack and the actual cost of the two stack operations. We assume that we start with an array of size $m = 1$ containing $n = 0$ elements. Throughout, we maintain the condition that (unless the stack is empty) $\frac{m}{4} < n < m$.

`push(x)`: Add x to the top of the stack and increase n by one. (This is always possible, by our assumption that $n < m$).

If $n < m$ (normal case), we are done, and the actual cost is $+1$. On the other hand, if $n = m$ (overflow case), we double the array size (setting $m \leftarrow 2m$), allocate a new array of this doubled size, copy the contents of the stack into the new array (see Fig. 3(a)). Letting n denote the number of elements *after* the push, the actual cost is $n + 1$ (+1 for the push, and n for the time to copy the elements). Observe that the new array is exactly half full.

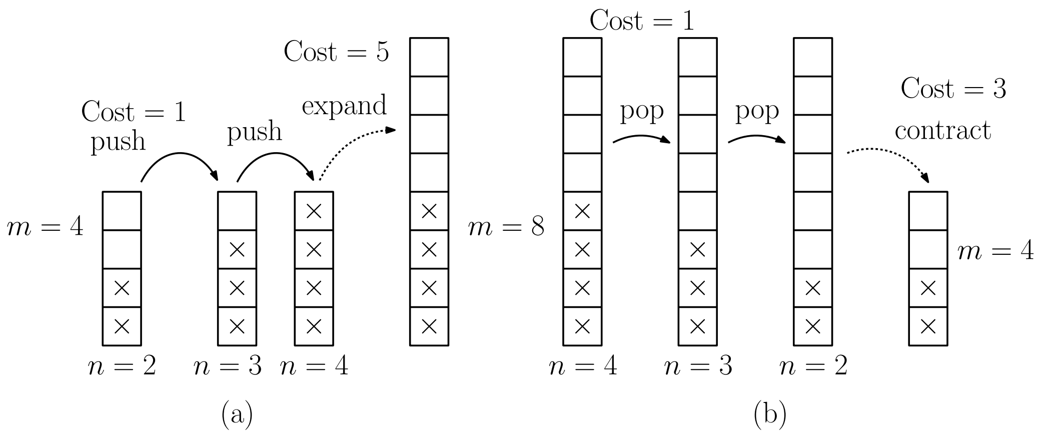


Figure 3: Expanding/Shrinking stack.

`pop()`: If $n = 0$, there is nothing to pop, and we return `null`, at an actual cost of +1. Otherwise, we pop the top element from the stack, and decrease n by one.

If after the pop, $n > \frac{m}{4}$ (normal case), we are done, and the actual cost is +1. On the other hand, if after the pop we have $n \leq \frac{m}{4}$ and $m \geq 2$ (underflow case), we halve the array size (setting $m \leftarrow \frac{m}{2}$), allocate a new array of this halved size, and copy the contents of the stack into the new array (see Fig. 3(b)). Letting n denote the number of elements *after* the pop, the actual cost is $n + 1$ (+1 for the pop, and n for the time to copy the elements). Observe that the new array is exactly half full.

The objective of this problem is to show that, over a long sequence of operations, the amortized cost (that is, the total actual cost divided by the number of operations) is some constant. We assume that we start within an empty stack ($n = 0$ and $m = 1$). Define *run* to be the sequence of operations starting just after the last array reallocation and running through the next array reallocation.

(a) (5 points) Suppose that the array size is m at the start of the run (and hence $n = m/2$), and the run ends with an *expansion* to size $2m$. Prove that there exists is a constant α_1 so that the amortized cost of the run (that is, the total cost of operations divided by the number of operations) is at most α_1.

(b) (5 points) Suppose that the array size is m at the start of the run (and hence $n = m/2$), and the run ends with a *contraction* to size $\frac{m}{2}$. Prove that there exists is a constant α_2 so that the amortized cost of the run (that is, the total cost of operations divided by the number of operations) is at most α_2.

For full credit, in each case compute the smallest value of α that works. You may assume that n is very large, so small additive constant terms do not matter.

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: Consider the setup in Problem 4 but suppose that the number of nodes decreases even faster. In particular, suppose that $n_{i+1} \leq \sqrt{n_i}$. Prove that there is a constant c such that $L \leq c \lg \lg n$ (that is, the log of the log of n). Since $\sqrt{1} = 1$, we could loop infinitely at the root level. Let's assume we end at level L, where $n_L = 2$.

General note regarding coding in homeworks: A common question at the start of the semester is "how much detail are you expecting?" You will figure this out as the semester goes on, but here are some basic guidelines.

Prove vs. Show: If we ask you to "prove" something, we are looking for a well structured proof. If you are applying induction, please be careful to distinguish your basis case(s) and indicate what your induction hypothesis is. If we ask you to "show," "explain," or "justify", we are usually just expecting a brief English explanation. If you are unsure, please check.

Algorithm vs. Pseudocode: When we ask for an "algorithm" we are expecting a high-level description of some computational process, usually in a combination of English and mathematical notation (e.g., "sort the n keys and locate x using binary search"). For the latter, we are expecting a more detailed step-by-step description that look much more like Java (e.g., "`Node q = p.left`").

Remember that you are writing your code to be read by a human, and not a Java compiler. Please omit extraneous details that are easily converted into Java. For example, it is easier to understand "`i =`$\lceil$`n/m`$\rceil$" than "`int i = (int) Math.ceil((double) n / (double) m))`".

Even if we do not explicitly ask for it, whenever you give an algorithm or pseudocode, **you should always provide a brief English explanation.** This helps the grader understand what your intentions are, and if there is a small error in your code, we can often use your explanation to understand what your actual intentions were. **Even if your solution is technically correct, we reserve the right to deduct points if it is not clear to us why it is correct.**

Submission Instructions: Please submit your assignment as a pdf file through Gradescope. Here are a few instructions/suggestions:

- You can typeset, hand-write, or use a tablet or any combination. We just need a readable pdf file with all the answers. Be generous with figures and examples. If there is a minor error in your pseudo-code, but the figure illustrates that you understood the answer, we can give partial credit.
- When you submit, Gradescope will ask you to indicate which page each solution appears on. **Please be careful in doing this!** It greatly simplifies the grading process for the graders, since Gradescope takes them right to the page where your solution starts. If done incorrectly,

the grader may miss your answer, and you may receive a score of zero. (If so, you can appeal. But hunting around for your answer is troublesome, and it is always best to keep the grader in a good mood!) This takes a few minutes, so give yourself enough time if you are working close to the deadline.

- Try to keep the answer to each subproblem (e.g. 5.2) on a single page. You can have multiple subproblems on the same page, but Gradescope displays one one page at a time. It is easiest to grade when everything needed is visible on the same page. If your answer spans multiple pages, it is a good idea to indicate this to alert the grader. (E.g., write "Continued" or "See next page" at the bottom of the page.)
- Most scanners (including your phone) do not take very good pictures of handwritten text. For this reason, write with dark ink on white paper. Use an image-enhancing app such as CamScanner or Genius Scan to improve the contrast.
- Writing can bleed through to the other side. To be safe, write on just one side of the paper.
- Students often ask me what typesetting system I use. I use LaTeX for text. This is commonly used by academicians, especially in math, CS, and physics, and is worth taking the time to learn if you are thinking about doing research. If you use LaTeX, I would suggest downloading an IDE, such as TeXnicCenter or TeXstudio. I draw my figures using a figure editor called IPE for drawing figures.

CMSC 420: Spring 2022

Homework 2: Search Trees

Handed out Tue, Feb 22. Due **Wed, Mar 2, 11:59pm**. Point values are tentative and subject to change.

Important! Solutions will be discussed in class on Thu, Mar 3, so **no late submissions will be accepted**. Turn in whatever you have completed by the due date.

Problem 1. (12 points) Consider the AVL tree shown in Fig. 1.

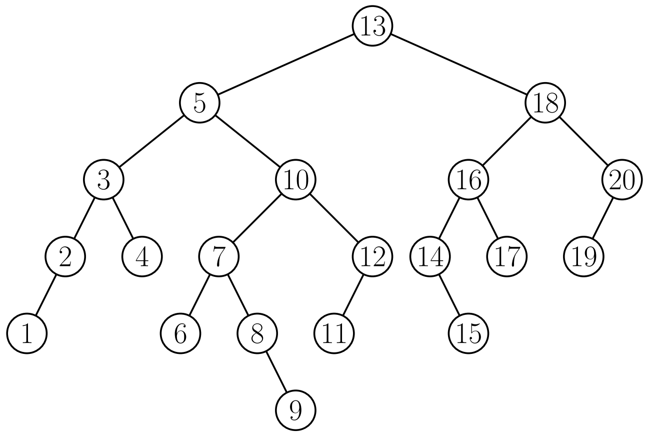
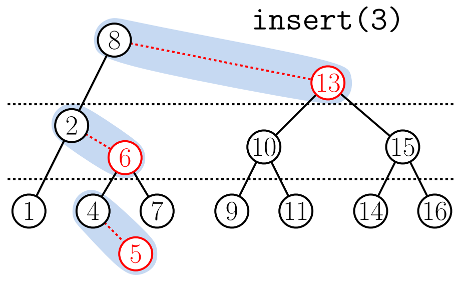


Figure 1: AVL Trees.

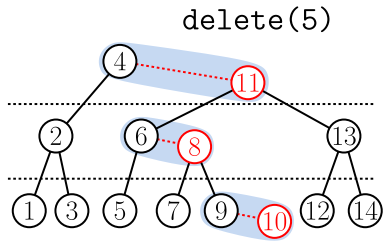
(a) (5 points) Draw the tree again, indicating the balance factors associated with each node.

(b) (7 points) Show the tree that results from the operation `delete(19)`, after all the rebalancing has completed. (We only need the final tree. You can provide intermediate results for partial credit.)

Problem 2. (12 points) Consider the AA trees shown in Fig. 2.



(a)



(b)

Figure 2: AA Trees.

(a) (6 points) Show the result of performing the operation `insert(3)` into the tree in Fig. 2(a).

(b) (6 points) Show the result of performing the operation `delete(5)` from the tree in Fig. 2(b).

(In both cases, we only need the final tree. You can provide intermediate results for partial credit. If you don't have two different colored pens, you can indicate red nodes with dashed edges and/or encircle circle groups of nodes as we do in our figures.)

Problem 3. (7 points) Recall the right rotation operation for a binary tree (given in Lecture 5).

```
Node rotateRight(Node p) {
    Node q = p.left
    p.left = q.right
    q.right = p
    return q
}
```

Suppose that we wish to apply this to a *threaded binary tree* using inorder threads (defined in Lecture 3). Explain what modifications (if any) are needed to perform a right rotation at node `p` so that after your modified function executes, all child links and threads are properly set. You may assume that the call is valid, in particular, `p` is non-null and `p`'s left-child link is standard parent-child pointer, and not a thread. Present your modified pseudocode and briefly explain why it is correct.

Note 1: Recall that the boolean's `leftIsThread` and `rightIsThread` are used to indicate that the left/right child link is a thread. These values may also need to be updated.

Note 2: It is possible that the rotation operation cannot be defined because it involves global knowledge of the tree structure beyond what is accessible through node `p`. If this is so, please explain why this is the case.

Problem 4. (7 points) Recall the code (shown below) for the operations `skew` and `split` for AA-trees (from Lecture 7).

```
AANode skew(AANode p) {                 |   AANode split(AANode p) {
    if (p == nil) return p              |       if (p == nil) return p
    if (p.left.level == p.level) {      |       if (p.right.right.level == p.level) {
        AANode q = p.left               |           AANode q = p.right
        p.left = q.right                |           p.right = q.left
        q.right = p                     |           q.left = p
        return q                        |           q.level += 1
    }                                   |           return q
    else return p                       |       }
}                                       |       else return p
                                        |   }
```

Also recall that each invocation of the `insert` function, the last line is "`return split(skew(p))`". There is an interesting phenomenon that sometimes occurs. It is illustrated in Fig. 3.

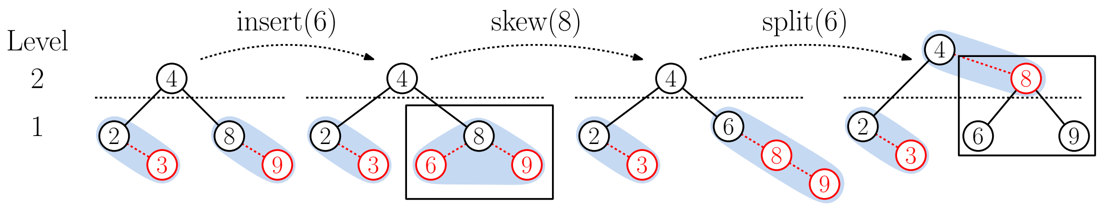


Figure 3: Ineffective skew-split combination.

Observe that the invocation of `skew(8)` results in a right rotation at `8` and returns a reference to node `6`. The subsequent invocation of `split(6)` results in a left rotation at `8` (thus undoing the previous rotation). It then promotes `8` to the next higher level. These two rotations effectively undo each other, and we call this skew-split combination *ineffective*.

In an effort to improve the efficiency of the AA tree, your task is to write an "effective" variant of skew-split. Your function, called `effectiveSkewSplit(p)` must be functionally equivalent to `split(skew(p))`. That is, it must have the same effect on the tree's structure, and it must return the same result. The only difference is that, it detects when an ineffective skew-split is about to occur and avoids doing the two rotations.

Present pseudocode for your function and explain why it is correct. As with skew and split, your function should run in $O(1)$ time.

Problem 5. (12 points) 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.

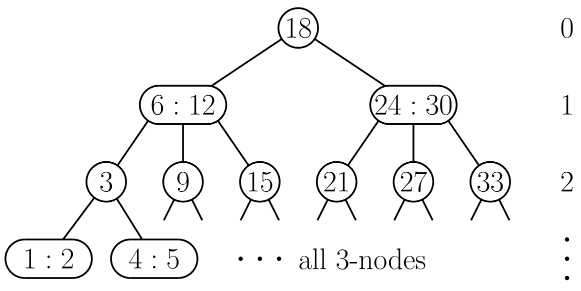


Figure 4: Alternating 2-3 tree.

(a) (6 points) 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) (6 points) 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).)

Challenge Problem 1: Continuing Problem 5 on the alternating 2-3 tree, for $i \geq 0$, define $N(i)$ to be the total number of nodes in all depths from 0 through i, and define $K(i)$ to be the total number of keys in all depths from 0 through i. Derive a closed-form mathematical formula for $N(i)$ or $K(i)$ (your choice). Present your formula and briefly explain how you derived it. As before, your formula should stand on its own and not make reference to $n(i)$ or $k(i)$.)

Challenge Problem 2: This problem is actually quite simple, but the "challenge" is to familiarize yourself with the section of the Latex lecture notes, Lecture X01, that discusses the amortized analysis of quake heaps.) In this problem we are going to do a walk-through of the amortized analysis of Quake Heaps for a single example of `extract-min` shown in Fig. 5.

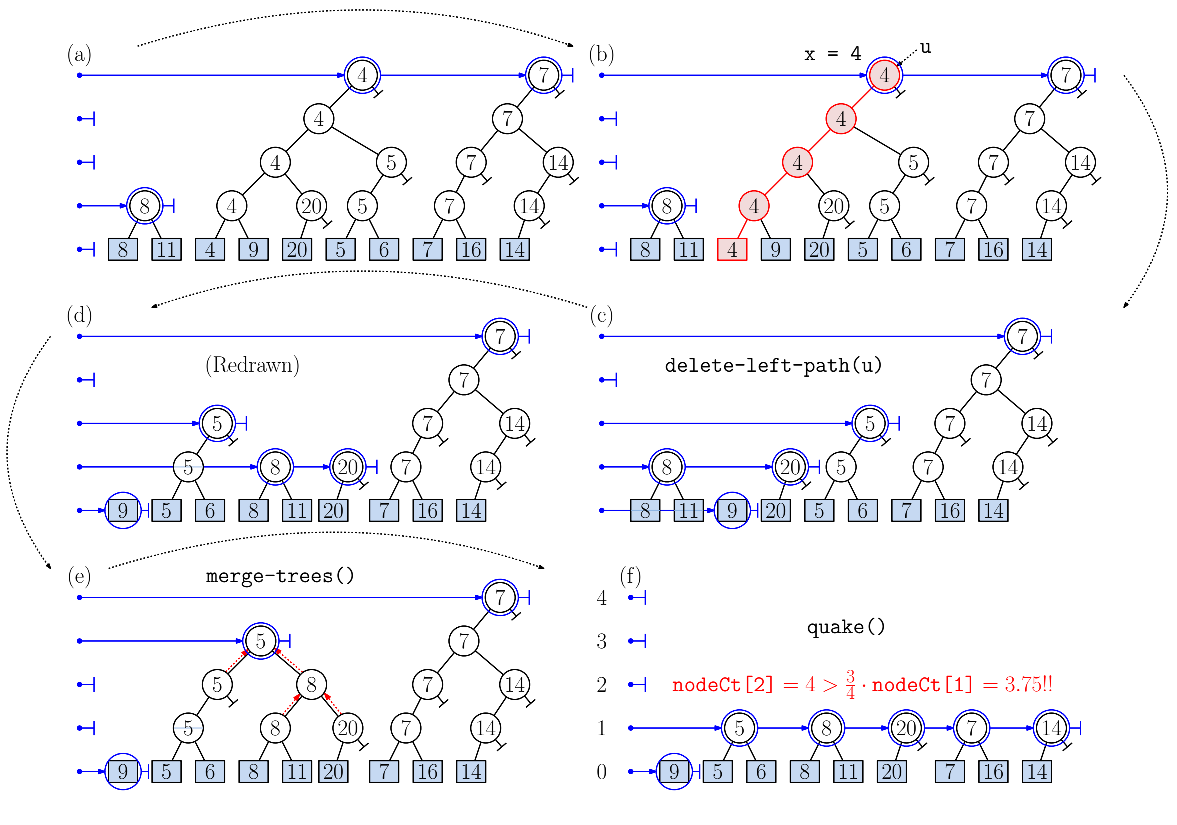


Figure 5: Quake Heap amortized analysis.

Recall that the *potential* for the data structure is defined to be $\Psi = N + 2R + 4B$, where N is the total number of nodes (both internal nodes and roots), R is the number of root nodes, and B is the number of nodes that have exactly one child (so called "bad nodes").

(a) Consider the initial quake heap shown in Fig. 5(a). For this tree, what are the values of N, R, B, and the potential Ψ?

(b) First, the algorithm searches all the roots to find the smallest key. Let T_b denote the actual cost, namely number of roots roots visited. What is T_b? Has the potential changed at all?

(c) Second, we delete all the nodes along the left path leading to the minimum root 4 (see Fig. 5(b)-(c)). Let T_c denote the actual number of nodes deleted. What is the value of T_c, and what is the net change to N, R, and B as a result? (Note that we have created some new roots in the process. If the number decreases, then the net change is negative.) Let $\Delta\Psi_c$ denote the net change to the potential. What is $\Delta\Psi_c$ and what is the total contribution $T_c + \Delta\Psi_c$ to the amortized cost?

(d) Next, we perform merge-trees (see Fig. 5(e)). Let T_d denote the actual cost of the number of new nodes created by the merging process. What is the value of T_d, and what is the net change to N, R, and B as a result? Let $\Delta\Psi_d$ denote the net change to the potential. What is $\Delta\Psi_d$ and what is the total contribution $T_d + \Delta\Psi_d$ to the amortized cost?

(e) Finally, we perform the quake operations (see Fig. 5(f)). Let T_e denote the actual cost of the number of new nodes deleted by the quake operations. What is the value of T_e, and what is the net change to N, R, and B as a result? Let $\Delta\Psi_e$ denote the net change to the potential. What is $\Delta\Psi_e$ and what is the total contribution $T_e + \Delta\Psi_e$ to the amortized cost?

(f) In summary, what is the total actual costs from this operation $T = T_b + \cdots + T_e$, what is the total change in potential $\Delta\Psi = \Delta\Psi_c + \cdots \Delta\Psi_e$, and what is the final amortized cost $T + \Delta\Psi$? (Note that total amortized cost may be negative, since we have improved the structure more than the actual amount of work needed to perform the operations.)

CMSC 420: Spring 2022

Practice Problems for Midterm 1

The exam will be held in class on **Tue, Mar 8**. It is close-book, closed-notes, but you will be allowed one sheet of notes, front and back.

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: AVL trees, 2-3 trees, AA trees, treaps, skiplists). Here is an example from last semester.

(a) Consider the 2-3 tree shown the figure below. Show the **final tree** that results after the operation `insert(6)`. When rebalancing, use only splits, *no adoptions* (key rotations).

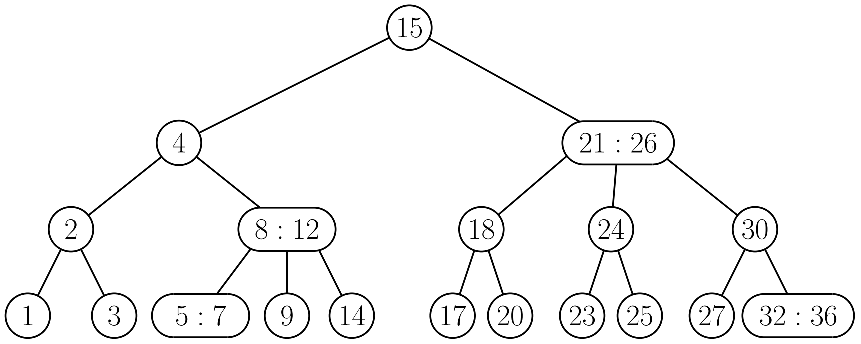


Figure 1: 2-3 tree insertion and deletion.

(b) Returning to the original tree, show the **final tree** that results after the operation `delete(20)` When rebalancing, you may use *both merge and adoption* (key rotation). If either operation can be applied, give priority to adoptions.

In both cases, you should use the algorithm presented in class. (You will receive partial credit if you produce a valid 2-3 tree, but not using the algorithm from class.)

Hint: Don't waste too much time showing intermediate results. You can return to this if you have spare time.

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) **True or false?** Let T be extended binary search tree (that is, one having internal and external nodes). In an inorder traversal, internal and external nodes are encountered in *alternating order*. (If true, provide a brief explanation. If false, show a counterexample.)

(c) **True or false?** In every extended binary tree having n external nodes, there exists an external node of depth $\leq \lceil \lg n \rceil$. Explain briefly. **Hint: Read this carefully before answering.**

(d) 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.)

(e) 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: $\sum_{i=0}^{m-1} c^i = (c^m - 1)/(c - 1)$.

(f) 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.

(g) 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?

(h) You have a valid AVL tree with n nodes. You insert two keys, one smaller than all the keys in the tree and the other larger than all the keys in the tree, but you do no rebalancing after these insertions. **True or False**: The resulting tree is a valid AVL tree. (Briefly explain.)

(i) By mistake, two keys in your treap happen to have the same priority. Which of the following is a possible consequence of this mistake? (Select one)

(i) The `find` algorithm may abort, due to dereferencing a `null` pointer.

(ii) The `find` algorithm will not abort, but it may return the wrong result.

(iii) The `find` algorithm will return the correct result if it terminates, but it might go into an infinite loop.

(iv) The `find` algorithm will terminate and return the correct result, but it may take longer than $O(\log n)$ time (in expectation over all random choices).

(v) There will be no negative consequences. The find algorithm will terminate, return the correct result, and run in $O(\log n)$ time (in expectation over all random choices).

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

(k) You are given a skip list storing n items. What is the expected number of nodes that are at level 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. You are given a degenerate binary search tree with n nodes in a left chain as shown on the left of Fig. 2, 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. 2).

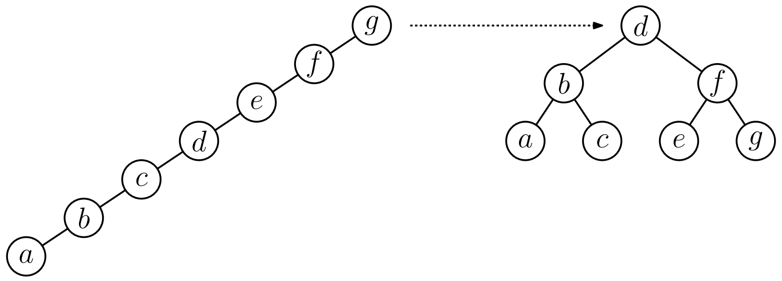


Figure 2: Rotating into balanced form.

(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 3. You are given a binary search tree where, in addition to the usual fields `p.key`, `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 pseudo-code for a function

```
Node preorderPred(Node p)
```

which is given a non-null reference `p` to a node of the tree and returns a pointer to `p`'s *preorder predecessor* in the tree (or `null` if `p` has no preorder predecessor). Your function should run in time proportional to the height of the tree. Briefly explain how your function works.

Problem 4. Recall that in a binary tree the *depth* of a node is defined to be the number of edges from the root to the node. The *height* of a node is defined to be the height of the subtree rooted at this node, that is, the maximum number of edges on any path from this node to one of its leaves.

In this problem, we will consider some questions involving nodes of a particular depth and height in an AVL tree. Let us assume (as in class) that an `AVLNode` stores its `key`, `value`, `left`, `right`, and `height`, and let us assume that the `AVLTree` stores a pointer to the `root` node.

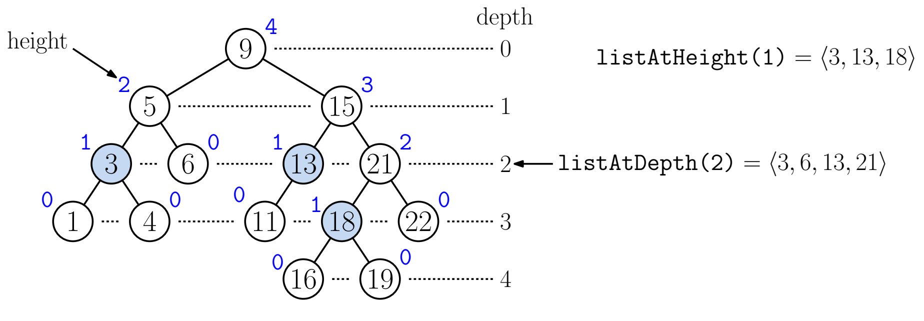

height
depth
9
4
5
2
15
3
3
1
6
0
13
1
21
2
1
0
4
0
11
0
18
1
22
0
16
0
19
0
0
1
2
3
4
listAtHeight(1) = ⟨3, 13, 18⟩
listAtDepth(2) = ⟨3, 6, 13, 21⟩

Figure 3: AVL tree heights and depths.

(a) Present an algorithm `listAtHeight(int h)`, which is given an integer $h \geq 0$, and returns a list (e.g., a Java `ArrayList`) containing all the keys, in increasing order, associated with all the nodes that are at height h in the AVL tree. If there are no nodes at height h, the function returns an empty list.

For example, in Fig. 3, the call `listAtHeight(1)` would return the list $\langle 3, 13, 18 \rangle$.

Briefly explain how your algorithm works, present a description (pseudocode preferred or a clear English explanation), and briefly explain its running time. The running time should be proportional to the number of nodes at height $\geq h$. (For example, in the case of `listAtHeight(1)`, there are 7 nodes of equal or greater height.)

(b) Present an algorithm `listAtDepth(int d)`, which is given an integer $d \geq 0$, and returns a list (e.g., a Java `ArrayList`) containing all the keys, in increasing order, associated with all the nodes that are at depth d in the AVL tree. If there are no nodes at depth d, the function returns an empty list. **Note:** Nodes do *not* store their depths, only their heights.

For example, in Fig. 3, the call `listAtDepth(2)` would return the list $\langle 3, 6, 13, 21 \rangle$.

In each of the coding problems, briefly explain how your algorithm works, present a description (pseudocode preferred or a clear English explanation), and briefly explain its running time. The running time of your algorithm should be proportional to the number of nodes at depths $\leq d$. (For example, in the case of `listAtDepth(2)`, there are 7 nodes of equal or lesser depth.)

(c) Prove that in any AVL tree, the maximum number of nodes that there are can be at depth $d \geq 0$ is 2^d. (**Hint:** This is intended to be easy. Even so, please give a short proof, even you think the observation is "obvious".)

(d) Given any AVL tree T and depth $d \geq 0$, we say that T is *full at depth d* if it has 2^d nodes at depth d. (For example, the tree of Fig. 3 is full at depths 0, 1, and 2, but it is not full at depths 3 and 4.) 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. 3 has height 4, and is full at levels 0, 1, and 2.)

Problem 5. 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. 4, 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.)

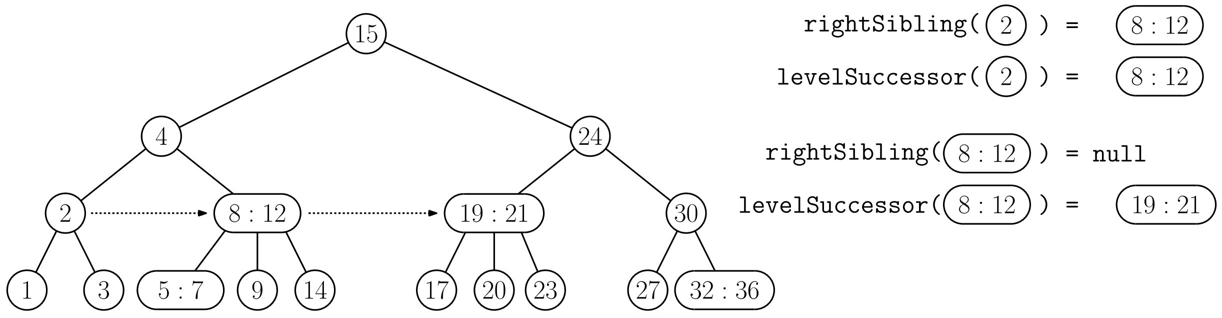

rightSibling(2) = 8 : 12
levelSuccessor(2) = 8 : 12
rightSibling(8 : 12) = null
levelSuccessor(8 : 12) = 19 : 21

Figure 4: Sibling and level successor in a 2-3 tree.

Your function should run in $O(1)$ time.

(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. 4, 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 6. A *social-distanced bit vector* (SDBV) is an abstract data type that stores bits, but no two 1-bits are allowed to be consecutive. It supports the following operations (see Fig. 5):

- `init(m)`: Creates an empty bit vector `B[0..m-1]`, with all entries initialized to zero.
- `boolean set(i)`: For $0 \leq i \leq m$ (where m is the current size of `B`), this checks whether the bit at positions i and its two neighboring indices, $i-1$ and $i+1$, are all zero. If so, it sets the ith bit to 1 and returns `true`. Otherwise, it does nothing and returns `false`. (The first entry, `B[0]`, can be set, provided both it and `B[1]` are zero. The same is true symmetrically for the last entry, `B[m-1]`.)

  For example, the operation `set(9)` in Fig. 5 is successful and sets `B[9] = 1`. In contrast, `set(8)` fails because the adjacent entry `B[7]` is nonzero.

There is one additional feature of the SDBV, its ability to *expand*. If we ever come to a situation where it is impossible set any more bits (because every entry of the bit vector is either nonzero or it is adjacent to an entry that is nonzero), we *reallocate* the bit vector to one of three times the current size. In particular, we replace the current array of size m with an array of size $3m$, and we copy all the bits into this new array, compressing them as much as possible. In particular, if k bits of the original vector were nonzero, we set the entries $\{0, 2, 4, \ldots, 2k\}$ to 1, and all others to 0 (see Fig. 5).

The cost of the operation `set` is 1, unless a reallocation takes place. If so, the cost is m, where m is the size of the bit vector *before* reallocation.

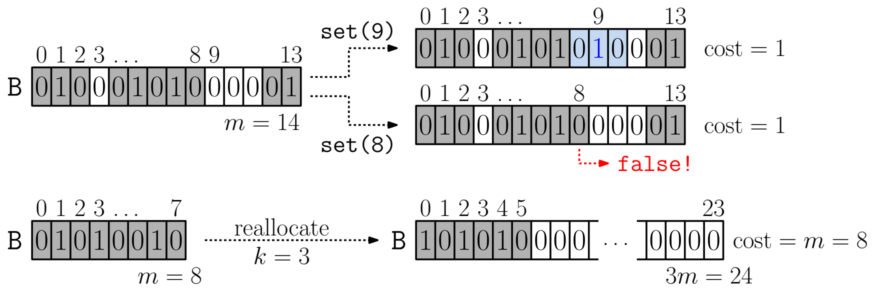


Figure 5: Social-distanced bit vector. (Shaded entries cannot be set to one, due to social-distancing.)

Our objective is to derive an amortized analysis of this data structure.

(a) Suppose that we have arrived at a state where we need to reallocate an array of size m. As a function of m, what is the minimum and maximum number of bits of the SDBV that are set to 1? (Briefly explain.)

(b) Following the reallocation, what is the minimum number of operations that may be performed on the data structure until the next reallocation event occurs? Express your answer as a function of m. (Briefly explain.)

(c) As a function of m, what is the cost of this next reallocation event? (Briefly explain.)

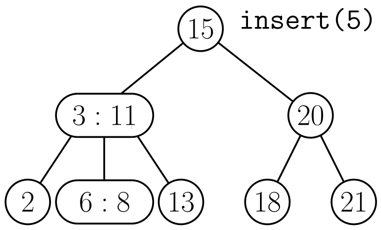
(d) Derive the amortized cost of the SDBV. (For full credit, we would like a tight constant, as we did in the homework assignment. We will give partial credit for an asymptotically correct answer. Assume the limiting case, as the number of operations is very large and the initial size of the bit vector is small.)

Throughout, if divisions are involved, don't worry about floors and ceilings.

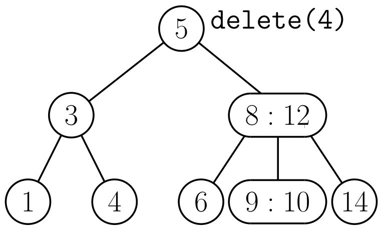
CMSC 420 (0101) - Midterm Exam 1

Problem 1. (10 points) For this problem, use the algorithms presented in class for 2-3 trees. (You will receive partial credit if you produce a valid 2-3 tree, but not using the algorithm from class.) Intermediate results are not required.

(a) (5 points) Show the **final tree** that results after the operation `insert(5)` to the 2-3 tree in the figure below.



(b) (5 points) Show the **final tree** that results after the operation `delete(4)` to the 2-3 tree in the figure below.



Problem 2. (30 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. Without knowing the tree structure, is it possible to know how many of the links in this tree are threads (versus normal parent-child links)? If so, indicate how many as a function of n.

(b) (6 points) 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.)

(1) It is definitely a full binary tree
(2) It is definitely **not** a full binary tree
(3) It is definitely a complete binary tree
(4) It is definitely **not** a complete binary tree
(5) Its height is larger than the original by exactly 1
(6) Its height is larger than the original, but the amount of increase need not be 1

(c) (5 points) You have an AA tree that contains an even number of keys n. As a function of n, what is the minimum and maximum number of red nodes that might be in this tree?

(d) (5 points) Suppose that you insert 13 keys into a quake heap and then perform the merge-trees operation. How many roots are there at each of the levels 0–4 in the resulting structure?

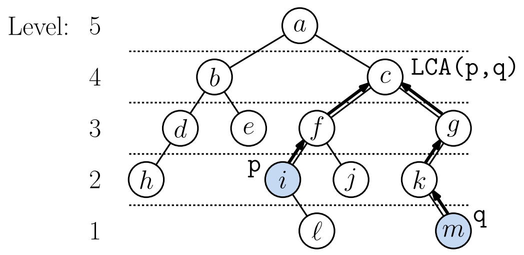
(e) (4 points) You insert a node x into a treap having at least three entries, and you observe that after the insertion, x is at the root of the tree. What can you say about the random priority assigned to x?

(1) It is the smallest
(2) It is the largest
(3) It is the median
(4) You can't infer anything about x's priority

(f) (6 points) A hacker tries to mess with your skip list as follows. First, they insert a large number of keys. After this, they delete all the keys with nodes that contribute to levels 1 and higher, effectively reducing your skip list to a standard linked list.

Is this an issue that the skip-list designer needs to worry about? Take a position (either "*This is an issue*" or "*This is not an issue*") and briefly justify your position. (You may assume the hacker does *not* have access to your random number generator.)

Problem 3. (20 points) Given two nodes `p` and `q` in a binary tree, their *lowest common ancestor*, denoted `LCA(p,q)`, is the common ancestor of these two nodes that is closest to both. (For example, in the figure below `LCA(p,q)` is the node labeled "c".) If `q` is an ancestor of `p` (possibly `p` itself), then `LCA(p,q) = q`.



In this problem, we assume that we are given a binary tree. Each node `p` is associated with the usual child pointers `p.left` and `p.right` as well as a parent pointer, `p.parent`.

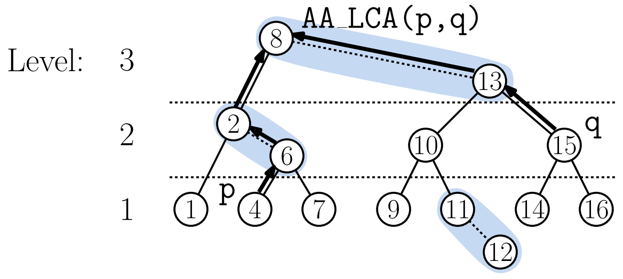
(a) (10 points) Each node `p` is associated with a *level number*, `p.level`, which grows by 1 as we go from child to parent. Present pseudocode for a function

```
Node LCA(Node p, Node q)
```

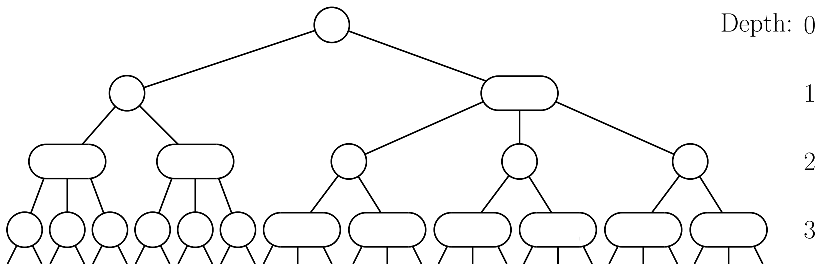
which returns the LCA of `p` and `q`. (For full credit, it should run in time proportional to the height of the tree and should not modify the tree.)

Hint: Move `p` and `q` up in a coordinated manner until they converge at the LCA.

(b) (10 points) Repeat part (a), but with the following twist. The tree is a (valid) AA-tree. This means that a node and its parent might be at the same level, according to the rules of AA trees. Call this function `AA_LCA`.



Problem 4. (20 points) This is a variant of the HW 2 problem called a *skewed alternating 2-3 tree*. The root is a 2-node. Its left child is a 2-node, and its right child is a 3-node. For each successive level the nodes alternate. The children of each 2-node are 3-nodes, and the children of each 3-node are 2-nodes.



(a) (10 points) For $i \geq 0$, define $n(i)$ to be the number of nodes at depth i in a skewed alternating 2-3 tree. Derive a **recurrence** for $n(i)$. Present your recurrence and briefly explain how you derived it.

Hint: I believe that the recurrence is simplest when you work two levels at a time, for example, try to express $n(i)$ in terms of $n(i-2)$. Be sure to give the base case(s).

(b) (10 points) Derive a **closed-form mathematical formula** (exact, not asymptotic) for $n(i)$. Present your formula and briefly explain how you derived it.

As in the homework, your formula should *not* involve summations or recurrences, but can involve multiple cases. You do *not* need to use the result of (a), and you do *not* need to give a formal proof of correctness.

Problem 5. (20 points) 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.

(a) (10 points) 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.

(You can do the special case $\gamma = 2$ for half-credit.)

(b) (10 points) Your computer has a *hardware accelerator* that copies a block of memory of size k in time $k/(\lg k)$. When the stack overflows, we allocate a new array of size $2m$, copy the elements from the old array over to the new array. The total cost is 1 (for the push) plus $m/(\lg m)$ (for copying). Derive a tight bound on the amortized cost, which holds in the limit as $m \to \infty$. Explain your answer.

CMSC 420: Spring 2022

Homework 3: More Search Trees and kd-Trees

Handed out Wed, Mar 30. Due **Wed, Apr 6, 11:59pm**. (Solutions will be discussed in class on Thu, Apr 7, so submissions will not be accepted after 9:30am, Apr 7.)

Problem 1. (15 points) Show the result of executing the operation `splay(8)` on the tree in Fig. 1.

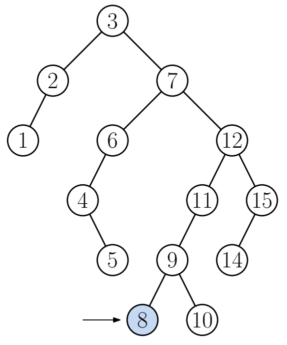


Figure 1: Splaying.

Problem 2. (15 points) Consider the scapegoat tree shown in Fig. 2. We will trace through the insertion of the key "p" into this tree (which fits between "o" and "q" alphabetically). For this problem, you may assume that `m == n` and both are equal to the number of nodes in the tree.

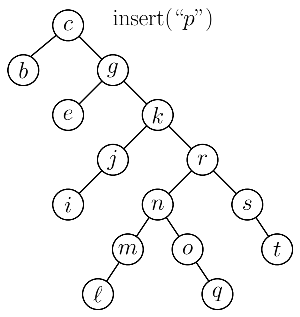


Figure 2: Insertion into a scapegoat tree.

(a) Following the insertion of "p", explain why a rebuild event will be triggered.

(b) Tracing the search path towards the root, which is the scapegoat node?

(c) Rebuild the subtree rooted at the scapegoat, and show the final tree with the rebuilt subtree inserted into its place. (Use the same algorithm given in the lecture notes for building the tree, with ties broken in the same manner.)

Problem 3. (10 points) In our lecture on Scapegoat Trees, we proved that if the tree has a node p at depth at least $\log_{3/2} n$, then somewhere along the search path to this node, there must exist a scapegoat node u. Recall that a node u is a *scapegoat* if

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

where $\text{size}(u)$ denotes the number of nodes in the subtree rooted at node u, and u.child denotes the child of u along the search path.

In this problem, we will ask you to generalize this result. Let α be any real constant, such that $\alpha > 1$. Complete the following lemma, and present a proof for it.

Lemma: Given a binary search tree of n nodes and any constant $\alpha > 1$, if there exists a node p such that $\text{depth}(p) > \log_\alpha n$, then p has an ancestor (possibly p itself) such that

$$\frac{\text{size}(u.\text{child})}{\text{size}(u)} > [\textit{“You fill this in”}]$$

For fullest credit, your answer to [*“You fill this in”*] should be as small as possible in the limit as $n \to \infty$. (**Hint:** Modify the proof from the Scapegoat tree lecture notes. You may also assume that `m == n`.)

Problem 4. (10 points) Throughout this problem 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)).

In a *vertical line-sliding query*, you are given an (infinitey) vertical line specified by its coordinate x_0 (see Fig. 3(b)). The query returns the first point $p_i \in P$ that is first hit if we slide the segment to its right. If no point of P are hit, the query returns `null`.

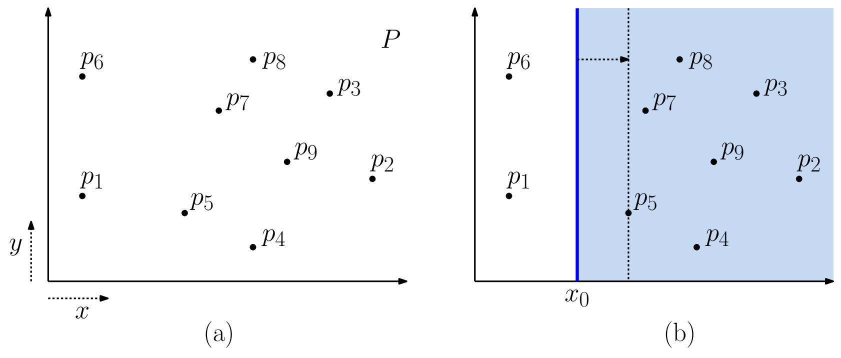


Figure 3: Segment-sliding and minimum box queries.

Present pseudo-code for an efficient algorithm, `Point vertLineSlide(scalar x0)`, which given x_0, the x-coordinate of the vertical line.

You may assume the standard 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. You may assume that there are no duplicate coordinate values among the points of P and no point of P lies on the vertical line $x = x_0$.

Briefly explain your algorithm and derive its execution time.

Hint: For fullest credit, your program should run in time $O(\sqrt{n})$, where n is the number of points in the structure. You may assume that the cutting dimensions alternate between x and y and that the tree is perfectly balanced, which means that if p's subtree contains m points then its children's subtrees each contain at most $m/2$ points.

Challenge Problem: All modern text editors provide an *undo* command (e.g., "control-Z"). In this problem, we would like to implement such an operation for a splay tree. Each time the undo operation is applied, the most recent splay operation is undone (see Fig. 4). If undo is performed multiple times, we step backwards undoing multiple splay operations until there are no more.

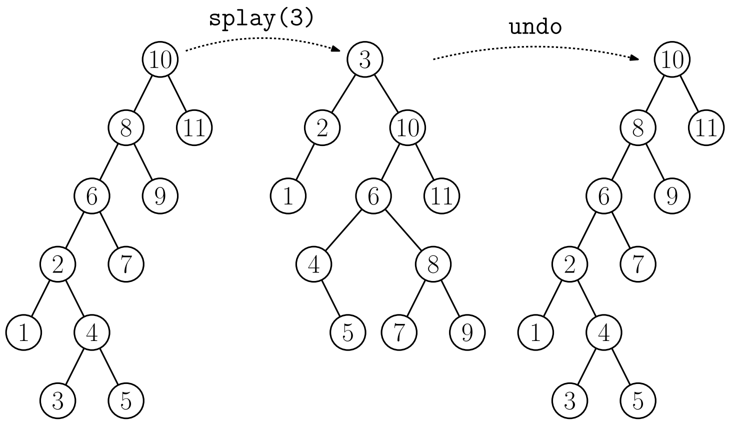


Figure 4: Undo operation on a splay tree.

The question to explore is how to do this efficiently. Obviously, we could just store the entire contents of every tree after each operation, but this would be hopelessly inefficient if the tree is very large. We would like to store the minimum amount of information in order to efficiently undo each splay.

(a) Here is an idea. There are essentially six different operations that a splay tree may perform at a given node p. These are named based on the relationship between p and its grandparent (see Fig. 5):

LL: Zig-zig, when p is a left-left grandchild (and the symmetrical **RR** operation)

LR: Zig-zag, when p is a left-right grandchild (and the symmetrical **RL** operation)

L: Zig, when p is the left child of the root (and the symmetrical **R** operation)

Each time we perform one of these operations as part of splaying, we push one of the symbols $\{LL, RR, LR, RL, L, R\}$ onto a stack S. Since each splay may consist of a variable number of rotations, before pushing these symbols, we push a special stack entry, $\perp$, which indicates the end of the rotations that constitute the splay operation.

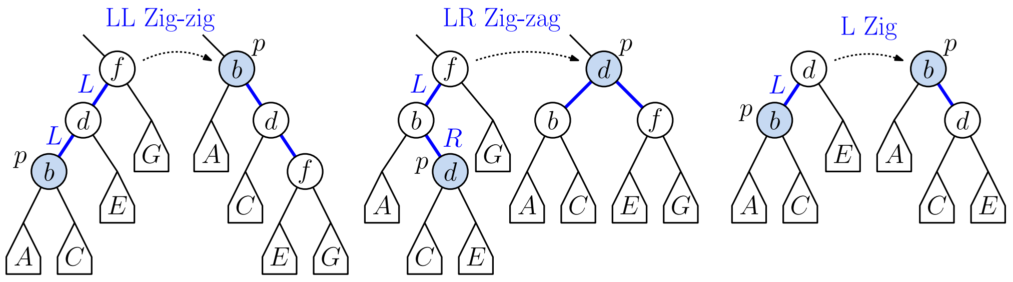


Figure 5: Undo operation on a splay tree.

For example, the splay operation of Fig. 4 involves the three rotations RL, LL, and L (from bottom to top). So the last four symbols in our stack would be $\langle \ldots, \perp, RL, LL, L \rangle$.) Present an algorithm which given such a stack S containing a (valid) combination of the 7 symbols $\{LL, RR, LR, RL, L, R, \perp\}$, applies the appropriate modifications to restore the previous splay tree in the sequence. In particular, as each symbol is popped off the stack, explain exactly what operation is performed on the tree to undo the rotation.

(b) Taking this one step farther, suppose that we whenever the operation `splay(x)` is performed, there is a node containing `x`, and this node is a leaf node in the current tree. Show that under this assumption, it is possible to implement the undo operation *without* the need of the special symbol $\perp$.

CMSC 420: Spring 2022

Practice Problems for Midterm 2

The exam will be held in class on **Tue, Apr 12**. It is close-book, closed-notes, but you will be allowed one sheet of notes, front and back.

Disclaimer: These practice problems have been extracted from old homework assignments and exams. Material changes from semester to semester. These do **not** necessarily reflect the actual coverage, difficulty, or length of the midterm exam.

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.

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

(a) What is the purpose of the *next-leaf* pointer in B+ 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) Both scapegoat trees and splay trees provide $O(\log n)$ amortized time for standard dictionary operations (insert, delete, and find). Suppose that your application involves many more find operations than insertions or deletions. Which of these two structures would you prefer and why?

(d) 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 $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1}) - 1)/(c-1)$.)

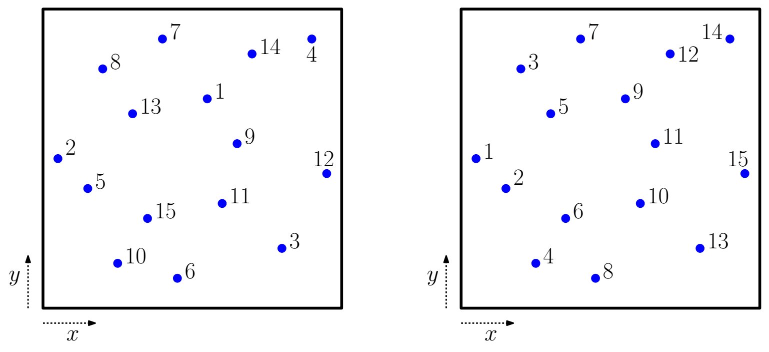


Figure 1: Height of kd-tree.

(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 (see the left side of Fig. 1). 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? (No explanation needed.)

(f) Same as the previous problem, but suppose that we insert points in *ascending* order of x-coordinates, but the y-coordinates are *random* (see the right side of Fig. 1). What is the expected height of the tree? (No explanation needed.)

(g) (**Note:** This problem is only applicable for the probing methods we discuss up to the time of the midterm.) 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

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

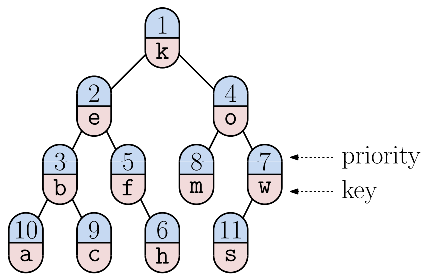


Figure 2: Treap node structure and an example.

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

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

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

Problem 3. 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 4. For this problem, assume that the structure of a node in a skip list is as follows:

```
class SkipNode {
    Key key;            // key
    Value value;        // value
    SkipNode[] next;    // array of next pointers
}
```

The height of a node (that is, the number of levels to which it contributes) is given by the Java operation `p.next.length`.

Often, when dealing with ordered dictionaries, we wish to perform a sequence of searches in sorted order. 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 want to start at `p`. Suppose that there are k nodes between x and y in the skip list. We want the expected search time to be $O(\log k)$, not $O(\log n)$.

(a) Present pseudo-code for an algorithm for a function `Value forwardSearch(p, y)`, which starting a node `p` (whose key is smaller than y), finds the node of the skip list containing key y and returns its value. (For simplicity, you may assume that key y appears within the skip list.) In addition to the pseudo-code, briefly explain how your method works.

Show that the expected number of hops made by your algorithm is $O(\log k)$, where k is the number of nodes between x and y. The proof involves showing two things:

(b) Prove that the maximum level reached is $O(\log k)$ in expectation (over random coin tosses).

(c) Prove that the number of hops per level is $O(1)$ in expectation.

Problem 5. 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. 3. 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.

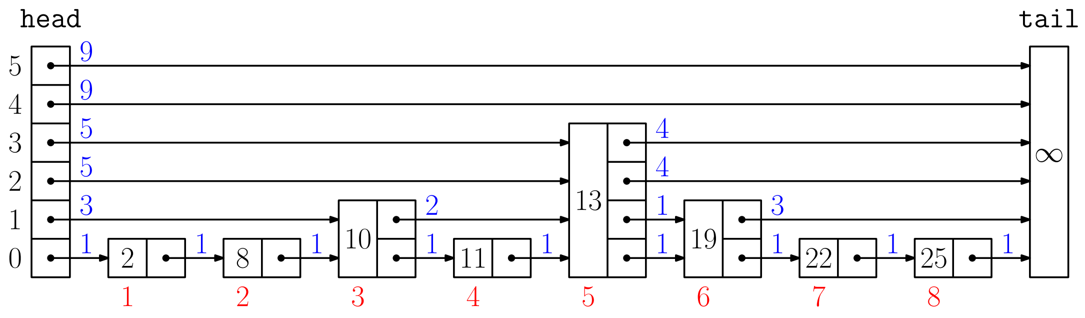


Figure 3: Skip list with span counts (labeled on each edge in blue).

(a) 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. 3, 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.

(b) Present pseudo-code for a function `Value getMinK(int k)`, which returns the value associated with the kth smallest key in the entire skip list. For example, in Fig. 3, the call `getMinK(5)` would return 13, since 13 is the fifth smallest element of the skip list. You may assume that $1 \leq k \leq n$, where n is the total number of nodes in the skip list.

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

Problem 6. It is easy to see that, if you splay twice on the same key in a splay tree (`splay(x); splay(x)`), the tree's structure does not change as a result of the second call.

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

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

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

Problem 7. Consider the following possible node structure for 2-3 trees (that is, a B-tree of order $m = 3$), 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. 4, 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.

(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

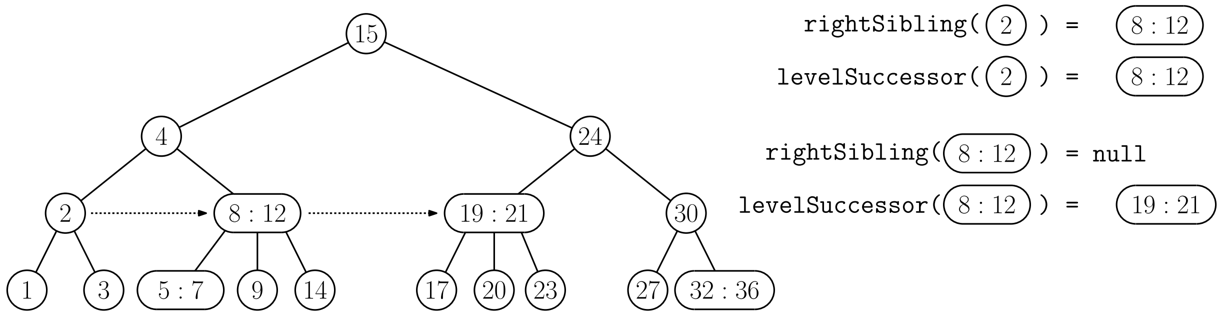

rightSibling(2) = 8 : 12
levelSuccessor(2) = 8 : 12
rightSibling(8 : 12) = null
levelSuccessor(8 : 12) = 19 : 21

Figure 4: Sibling and level successor in a 2-3 tree.

level (including the case where `p` is the root), this function returns `null`. (For example, in Fig. 4, 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. Given a set P of n points in the real plane, a *partial-range max query* is given two x-coordinates x_1 and x_2, and the problem is to find the point $p \in P$ that lies in the vertical strip bounded by x_1 and x_2 (that is, $x_1 \le p.x \le x_2$) and has the maximum y-coordinate (see Fig. 5).

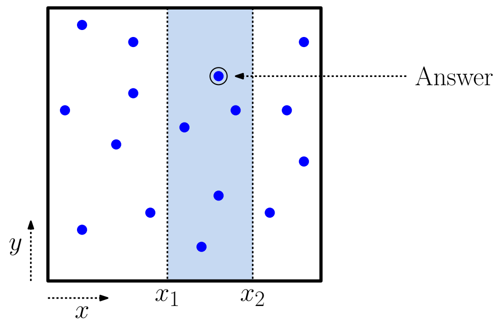


Figure 5: Partial-range max query.

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

```
Point partialMax(double x1, double x2, KDNode p, Rectangle cell, Point best)
```

Assuming the tree is balanced and the splitting dimension alternates between x and y, show that your algorithm runs in time $O(\sqrt{n})$.

Problem 9. 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.

The purpose of this problem is to show that does not apply to lines that are not axis-parallel. 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 10. 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. 6). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.

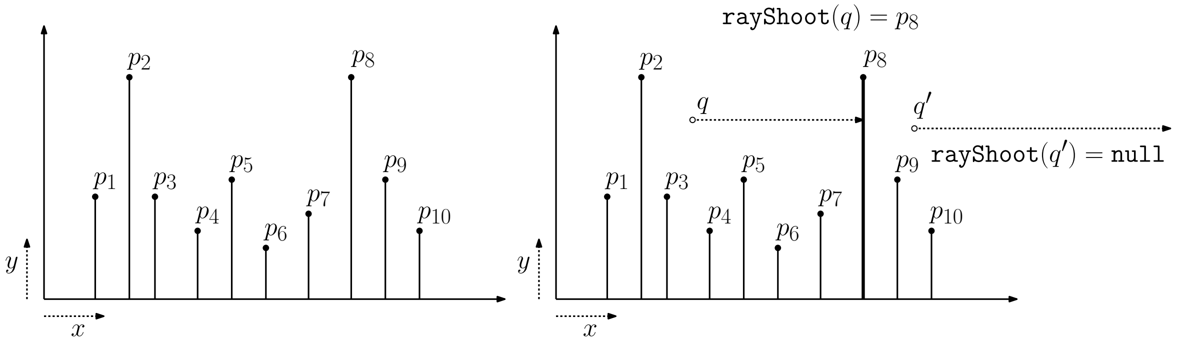


Figure 6: Ray shooting in a kd-tree.

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 11. (**Note:** This problem may not be applicable, depending on whether we discuss deletion in hash tables before the exam.) In class we demonstrated a simple idea for deleting keys from a hash table with open addressing. Namely, whenever a key is deleted, we stored a special value "`deleted`" in this cell of the table. It indicates that this cell contained a deleted key. The cell may be used for future insertions, but unlike "`empty`" cells, when the probe sequence searching for a key encounters such a location, it should continue the search.

Suppose that we are using *linear probing* in our hashing system. Describe an alternative approach, which does not use the "`deleted`" value. Instead it moves the table entries around to fill any holes caused by a deleted items.

In addition to explaining your new method, justify that dictionary operations are still performed correctly. (For example, you have not accidentally moved any key to a cell where it cannot be found!)

CMSC 420: Spring 2022

CMSC 420 (0101) - Midterm Exam 2

Problem 1. (20 points) This problem involves splay trees. In both cases, apply the splay-tree algorithm given in the lecture notes.

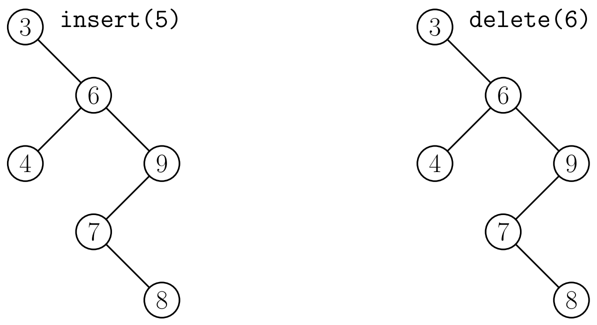


Figure 1: Splay tree operations

(a) (10 points) Show the steps involved in operation `insert(5)` for the tree in the figure. Indicate what splays are performed, and what trees result from each splay. Also indicate what node(s) are created, and show the final tree after insertion.

(b) (10 points) Show the steps involved in operation `delete(6)` for the tree in the figure. In particular, indicate what splays are performed, and what trees result from each splay. Also indicate what node(s) are removed, and show the final tree after deletion.

Problem 2. (30 points) Short answer questions. Unless requested, explanations are not required, but may always be given to help with partial credit.

(a) (4 points) By mistake, two keys in your treap happen to have the same priority. When you attempt to execute `find`, which of the following is a possible consequence? (Select all that apply.)

(1) It might go into an infinite loop

(2) It might return the wrong result

(3) It might take longer than $O(\log n)$ expected time

(4) There are no significant negative consequences

(b) (4 points) You have a splay tree storing n keys $x_1 < x_2 < \cdots < x_n$. Suppose that you perform n consecutive splay operations on these keys in sorted order:

$$\texttt{splay}(x_1),\ \texttt{splay}(x_2),\ \ldots,\ \texttt{splay}(x_n).$$

Which of the following is the *total* running time for all of these operations? (Give the tightest correct bound.)

(1) $O(n)$ (worst-case)

(2) $O(n \log n)$ (worst-case)
(3) $O(n)$ (expected case)
(4) $O(n \log n)$ (expected case)
(5) None of the above

(c) (6 points) Consider a B-tree of order $m = 13$. Answer the following questions for a single node that is *not* the root and *not* a leaf.

(a) What is the maximum number of children?
(b) What is the minimum number of children?
(c) What is the maximum number of keys?
(d) What is the minimum number of keys?

(d) (2 points) True or false: After performing an insertion in a scapegoat tree (but before any rebuilding), two or more nodes on the search path may satisfy the scapegoat condition (`size(p.child)/size(p)` $> 2/3$).

(e) (2 points) True or false: As part of performing an insertion in a scapegoat tree, two or more subtrees may be rebuilt.

(f) (4 points) In high dimensional spaces (say, dimensions greater than 10), kd-trees are preferred over quadtrees. Why is this?

(g) (4 points) Given a balanced kd-tree storing n points in 2-dimensional space, where we alternate the splitting axes, and given an *axis-parallel line* ℓ, what is the maximum number of nodes whose cell intersects ℓ?

(1) $O(\log n)$
(2) $O(\sqrt{n})$
(3) $O(n)$
(4) None of the above

(h) (4 points) Repeat (g), but this time ℓ may have an *arbitrary slope.*

Problem 3. (15 points) 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)` $\geq 2/3$ (see the figure below). Let T be a left-heavy tree that contains n nodes.

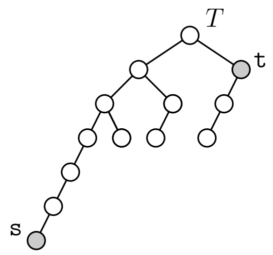


Figure 2: A left-heavy tree.

(a) (8 points) Consider any left-heavy tree T with n nodes, and let `s` be the leftmost node in the tree. Prove that `depth(s)` $\geq \log_{3/2} n$. (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.)

(b) (7 points) 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. (15 points) You are given a set $P = \{p_1, \ldots, p_n\}$ of n points in 2D space, all having positive x- and y-coordinates (see Fig. 3(a)). They have been stored in a perfectly balanced kd-tree using alternating cutting dimensions, $x, y, x, y, \ldots$. The kd-tree stores a bounding box `bbox` that contains all the points and a pointer `root` to the root of the kd-tree.

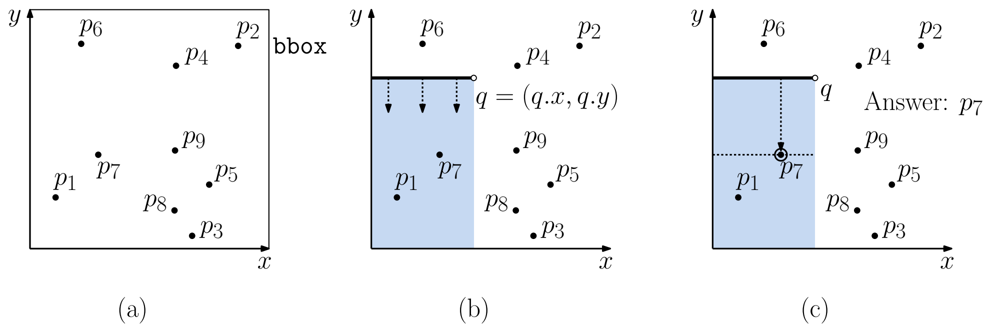


Figure 3: Platform dropping queries.

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 (see Fig. 3(b)). Our objective is to report the first point $p \in P$ that would be hit if we drop the platform. Formally, this point has the maximum y-coordinate such that $p_x \leq q_x$ and $p_y \leq q_y$ (see Fig. 3(c)).

In this problem we will derive an efficient algorithm to answer such queries. You will fill in the missing gaps to a recursive helper function. The helper is given the query point `q`, the current node `p` being visited (which might be `null`), the associated rectangular `cell` for this node, and `best`, the best answer seen so far. The function returns the new best point.

(a) (4 points) Give a condition on `cell` so that *no* point in `p`'s subtree can be hit by the falling platform? Express your answer in terms of `q = (q.x, q.y)` and the low and high cell corner points `cell.lo` and `cell.hi`.

(b) (3 points) What conditions must `p.point` satisfy to replace `best` as the new best answer to the query?

(c) (4 points) Suppose that `cell` lies entirely to the left of `q` (that is, `cell.hi.x` $\leq$ `q.x`), and further suppose that `p` has a cutting dimension of 1 (horizontal cut). Explain why only one of `p`'s children need be searched for the answer. How would you determine which child it is?

(d) (2 points) Based on your above answers, fill in the boxes boxes to complete the helper function for answering platform dropping queries. It is given the query point `q`, the current node `p` being visited (which might be `null`), the associated rectangular `cell` for this node, and `best`, the best answer seen so far. The function returns the new best point. (If there is not change from the above, you can just say "Same as (a)".)

`Note:` If the structure of our helper function does not make sense to you, you can optionally provide your own helper function in its entirety at the end of the exam. Please put in note in the first fill-in box that you are doing this.

```
Point platformDrop(Point q, KDNode p, Rectangle cell, Point best) {
    if (p == null) return best
                                                           (a)
    if (                                                 ) return best
                                                           (b)
    if (                                                 ) best = p.point
    Rectangle leftCell = cell.leftPart(p.cutDim, p.point)
    Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
    if (cell.hi.x <= q.x && p.cutDim == 1) {
                                                           (c)
        if (                                             )
            best = platformDrop(q, p.left, leftCell, best)
        else
            best = platformDrop(q, p.right, rightCell, best)
    } else {
        best = platformDrop(q, p.right, rightCell, best)
        best = platformDrop(q, p.left, leftCell, best)
    }
    return best
}
```

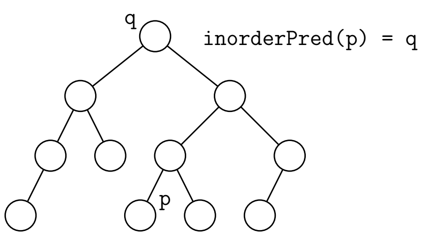
(e) (2 points) What is the initial call to the helper function?

Problem 5. (20 points)

(a) (10 points) Suppose that you are given a standard rooted binary tree, where in addition to left and right, each node has a parent link. (There are no keys.)

```
class Node {                    // a node in the tree
    Node      left              // left child
    Node      right             // right child
    Node      parent            // parent (null if root)
}
```

Present pseudocode for a function `Node inorderPred(Node p)`, which returns a pointer to inorder predecessor of `p`. If `p` has no inorder predecessor, it should return `null`.



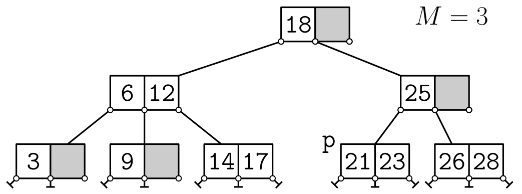
Briefly explain how your function works. If you wish, you may define additional helper functions. Your function should run in time proportional to the height of the tree.

(b) (10 points) Suppose that we are given a B-tree of order M, for some $M \geq 3$. Let us assume the following node structure:

```
class BNode {                     // a node in a B-tree
    int       nc                  // number of children
    BNode     child[M]            // children pointers
    Key       key[M-1]            // keys
    BNode     parent              // parent
}
```

The children are `child[0..nc-1]` and the keys are `key[0..nc-2]`. The entry `key[i]` sits between subtrees `child[i]` and `child[i+1]`.

Present pseudocode for a function `Key inorderPred(BNode p, int i)`, which returns the key of the B-tree that immediately precedes `p.key[i]` in sorted order.



```
inorderPred(p,0) = 18
inorderPred(p,1) = 21
```

Briefly explain how your function works. If you wish, you may define additional helper functions. Your function should run in time proportional to the height of the tree (assuming M is a constant).

Hint: There are a number of cases. You can get partial credit if you can solve some of the subcases correctly. For example, you might start by considering what happens when `p` is a leaf node.

CMSC 420: Spring 2022

Homework 4: Range Trees, Hashing, and Tries

Handed out Tue, May 3. Due **Tue, May 10, 11:00am**. (Solutions will be discussed in class on Tue, May 10, so turn in whatever you have finished by then.) You may drop the lowest of your four homework scores. Even if you do not attempt this assignment, note that the material will be covered on the final exam.

Problem 1. (15 points) In this problem, you will show the result of inserting a sequence of three keys into a hash table, using linear and quadratic probing and double hashing. In each case, at a minimum 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 initial access counts as a probe, so if there is no collision, the number of probes is 1.)
- 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. 1(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") = 13
insert("Y")  h("Y") = 5
insert("Z")  h("Z") = 14
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
▨	I	B	F	A	M	Q	▨	J	▨	C	L	▨	H	P	W

(b) Quadratic probing

```
insert("X")  h("X") = 4
insert("Y")  h("Y") = 8
insert("Z")  h("Z") = 3
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	▨	▨	F	▨	B	▨	L	N	P	▨	W	▨	▨

Figure 1: Hashing with linear and quadratic probing.

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

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

Problem 2. (15 points) In this problem we will build a suffix tree for $S =$ `"baabaababaa$"`.

(a) (7 points) Recall that the 12 suffixes of S are (in reverse order):

$$S_{11} = \texttt{"\$"}, \quad S_{10} = \texttt{"a\$"}, \quad S_9 = \texttt{"aa\$"}, \quad \ldots, \quad S_0 = \texttt{"baabaababaa\$"}.$$

(c) Double hashing

```
insert("X")  h("X") = 13; g("X") = 6
insert("Y")  h("Y") = 4; g("Y") = 3
insert("Z")  h("Z") = 3; g("Z") = 4
```

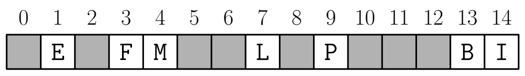


Figure 2: Hashing with double hashing.

Let id$_j$ denote the *substring identifier* for S_j. (Recall from Lecture 17 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 $\text{id}_0 \ldots \text{id}_{11}$).

(b) (8 points) Draw the suffix tree for S. Draw your tree in the same edge labeling style we used in Fig. 7 in Lecture 17 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. (14 points) 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. 3(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) (7 points) 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. 3(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) (7 points) 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. 3(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`.

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

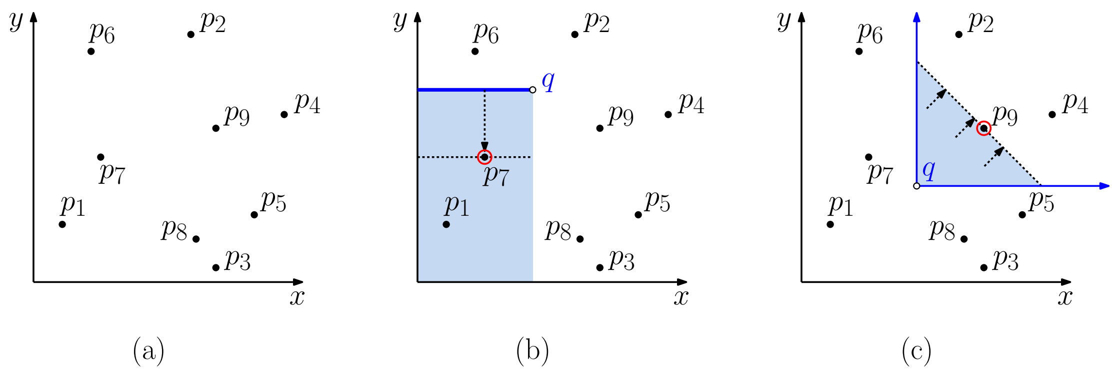


Figure 3: Platform-dropping and max empty-triangle queries.

Problem 4. (5 points) Consider the buddy-system memory allocation shown in Fig. 4, where shaded blocks are allocated and white blocks are free. Suppose that we deallocate the block of size 1 at address 22. Explain which blocks are merged together, and what single block replaces them all.

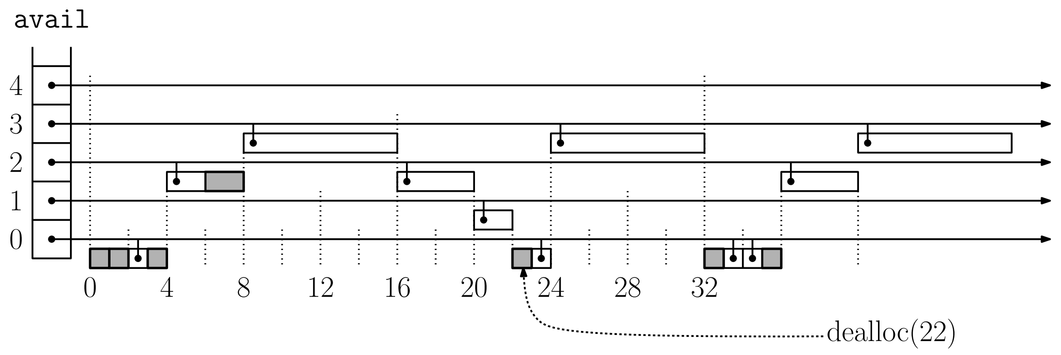


Figure 4: Buddy system memory allocation.

CMSC 420: Spring 2022

Practice Problems for the Final Exam

Our final exam will be held on **Fri, May 13, 4–6pm** in **Tydings Hall (TYD) 0130**. It is close-book, closed-notes, but you will be allowed three sheets of notes, front and back.

Disclaimer: The exam will be comprehensive, emphasizing material in the latter half of the semester. **These practice problems reflect the just material since the second midterm, but you should expect coverage of other topics as well.** They have been extracted from old homework assignments and exams. Material changes from semester to semester. These do **not** necessarily reflect the actual coverage, difficulty, or length of the midterm 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 *after* 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) 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.

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

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

(e) 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?

(f) Consider the following dictionary structures: (1) Unbalanced binary search trees, (2) AVL trees, (3) AA-trees, (4) quake-heaps, (5) treaps, (6) splay trees, (7) scapegoat trees. Suppose we insert one key into such a data structure that contains n keys. For which of these data structures can we assert that the worst-case number of structural changes to the tree is $O(\log n)$? (A *structure change* is any local alteration of the structure: creating/modifying node contents, rotation, node split, etc.)

(g) A scapegoat tree containing n keys has height $O(\log n)$ ... (select one):

  (i) Always—the height is guaranteed

  (ii) In expectation, over the algorithm's random choices

  (iii) In expectation, assuming that keys are inserted in random order

  (iv) In the amortized sense—the average height will be $O(\log n)$ over a long sequence of operations

  (v) Maybe yes, maybe no—there is just no way of knowing

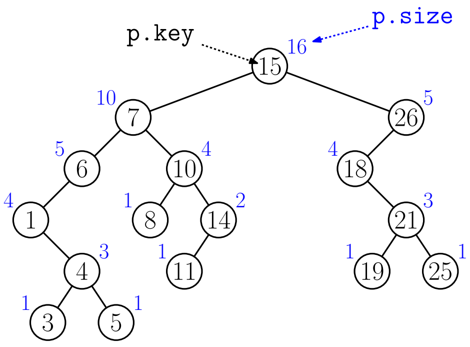
(h) Between the classical dynamic storage allocation algorithm (with arbitrary-sized blocks) or the buddy system (with blocks of size power of 2) which is more susceptible to *internal fragmentation*? Explain briefly.

(i) (Check out Problem 1(g) from the Practice Problems for Midterm 2 on hashing.)

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;
    Node left;
    Node right;
    int size; // number of nodes 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.)



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

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

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

Figure 1: The functions printMaxK and printOdd.

You should do this by traversing the tree. You are not allowed to "cheat" but storing an auxiliary list of sorted nodes.

For fullest credit, the keys should be printed in *ascending order*, and your algorithm should run in time $O(k + \log n)$ (see part (4.2) below). (Partial credit will be given otherwise, but an $O(n)$ time algorithm is not worth anything.)

You may assume that $0 \leq k \leq n$, where n is the total number of nodes in the tree. 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 $\langle x_1, x_3, x_5, \ldots, x_n\rangle$, if n is odd, and $\langle x_1, x_3, x_5, \ldots, x_{n-1}\rangle$, if n is even.

Beware: We are not printing the "odd-valued" keys, rather we are printing the odd numbered positions in the sorted order (see Fig. 1.)

Again, you should do this by traversing the tree. You are not allowed to "cheat" by storing auxiliary lists or using global variables. Your program should run in time $O(n)$. Briefly explain your algorithm.

Problem 3. Throughout this problem, assume that you are given a standard kd-tree storing a set P of n points in $\mathbb{R}^2$ (see Fig. 2(a)). Assume that the cutting dimension alternates between x and y. You may also assume that the tree stores a bounding box `bbox`, which is a 2-dimensional rectangle 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.

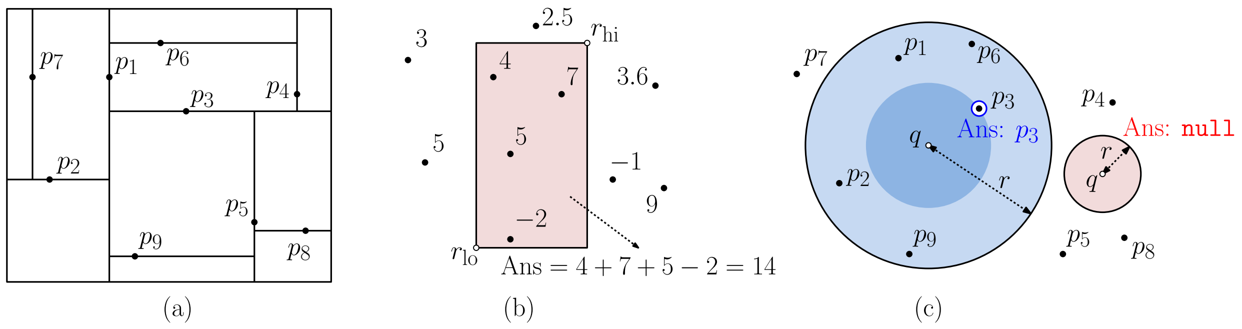


Figure 2: Queries on kd-trees.

(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.)

(c) In a *fixed-radius nearest neighbor query*, we are given a point $q \in \mathbb{R}^d$ and a radius $r > 0$. Let C denote the 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. Present (in pseudo-code) an efficient kd-tree algorithm that answers such a query.

You may handle the edge cases (e.g., multiple points at the same distance or points lying on the boundary of C) however you like. **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.

Briefly explain your algorithm, but you *do not* need to derive its running time.

Problem 4. In this problem we will build a suffix tree for the string $S =$ `baabaabababaa$`.

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

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

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

Problem 5. 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. 3(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. 3(b)). (As an application, imagine you are searching for the k highest rated restaurants in a rectangular region of some city.)

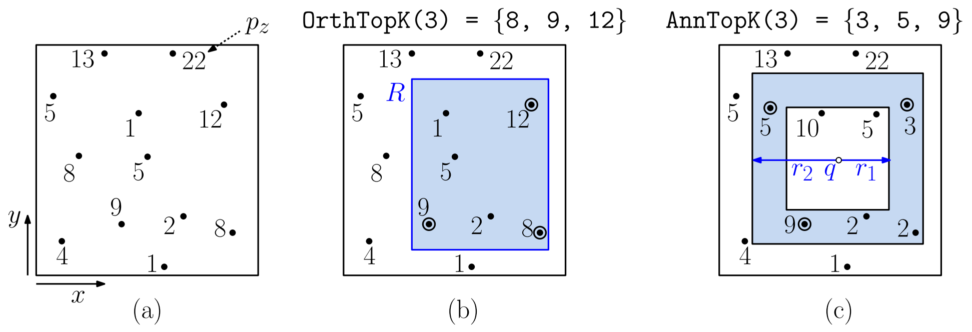


Figure 3: Orthogonal top-k queries and annulus top-k queries.

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. 3(c)).

Problem 6. Suppose you have a large span of memory, which starts at some address `start` and ends at address `end-1` (see Fig. 4). (The variables `start` and `end` are generic pointers of type `void*`.) As the dynamic memory allocation method of Lecture 15, this span is subdivided into blocks. The block starting at address `p` is associated with the following information:

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

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

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. 4). 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.

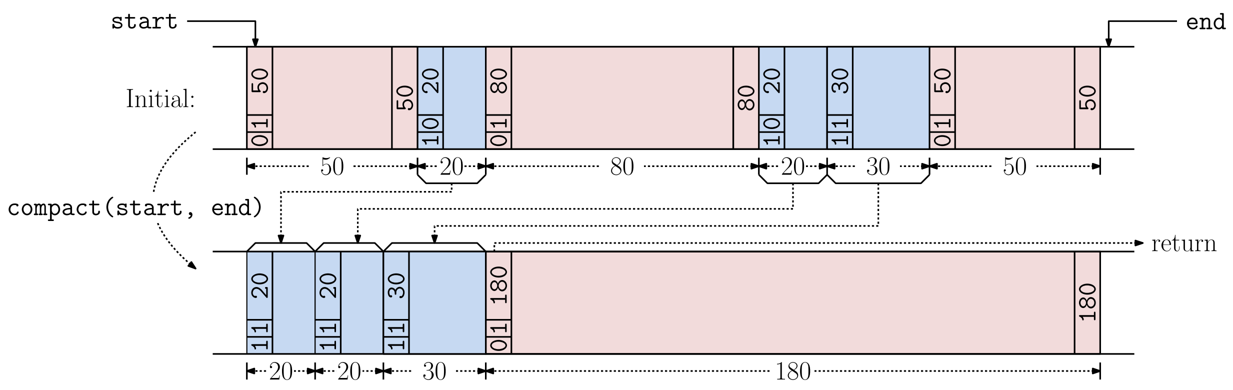


Figure 4: Memory compactor.

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 7. Recall the buddy system of allocating blocks of memory (see Fig. 5). 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)` $= 111_2 = 7$.

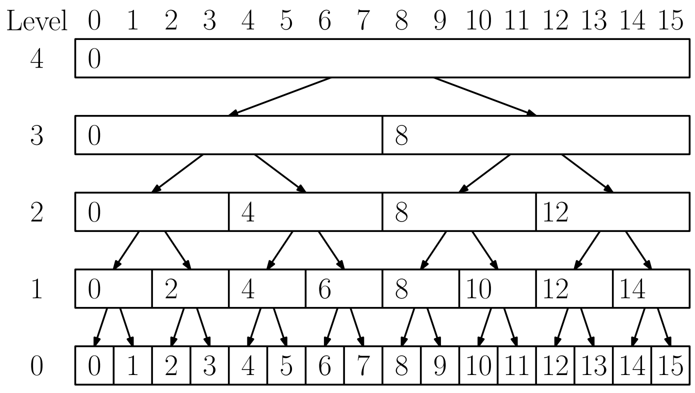


Figure 5: Buddy relatives.

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 8. 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 9. (Check out Problem 11 from the Practice Problems for Midterm 2 on deletion in open-addressing hashing.)

CMSC 420 (0101) - Final Exam

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

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

(a) (4 points) Early in the semester we saw that a 2-dimensional matrix could be represented using a *multilist structure*. What are the main advantages of the multilist over a standard array-based representation? (Select all that apply.)

(1) Matrix entries can be *modified faster* with the multilist
(2) Matrix operations (e.g., multiplication) are *generally faster* with the multilist
(3) If the matrix is sparse (few non-zero elements), the multilist *saves space*

(b) (6 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) It is best to take the replacement exclusively from `p`'s right subtree
(5) At most one replacement is needed for each deletion operation

(c) (2 points) 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?

(d) (4 points) Suppose that a subtree of height h in a Quake Heap has been constructed by applying some number of `link` operations (but no `cut`'s). As a function of h, how many leaves does this subtree have? (Select one.)

(1) Exactly 2^h
(2) At most 2^h, but possibly fewer
(3) At least 2^h, but possibly more
(4) We cannot put an exact upper or lower bound, but it will be $O(2^h)$
(5) None of the above

(e) (3 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 and why?

(f) (4 points) 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.

(g) (3 points) 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?

(h) (4 points) In our implementation of Prim's EMST algorithm, which of the following is true throughout the execution of the algorithm? (Select all that apply)

(1) The number of entries in the spatial index (kd-tree) is at least as large as the number of points in the current spanning tree

(2) The number of entries in the spatial index (kd-tree) is at least as large as the number of points that are *not* in the current spanning tree

(3) The number of entries in the priority queue (heap) is at least as large as the number of points in the current spanning tree

(4) The number of entries in the priority queue (heap) is at least as large as the number of points that are *not* in the current spanning tree

Problem 2. (10 points) Show the result of executing the operation `splay(5)` on the tree in Fig. 1. (Intermediate results may be given for partial credit.)

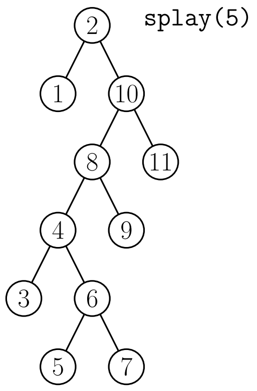


Figure 1: Splaying.

Problem 3. (15 points) Perform the following operations on a hash table. In each case, the operations are performed *as a sequence* and list the *number of probes.*

(a) Show the results of inserting the keys "X" then "Y" then "Z" into the hash table shown in Fig. 2(a) assuming *double hashing*, where `g()` is the jump size.

(b) Show the result of insertions and deletions in Fig. 2(b), assuming *linear probing.* (When deleting, indicate what your placeholding symbol is.)

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

(a) (4 points) List the substring identifiers for the 8 suffixes of S. For the sake of uniformity, list them in order (either back to front or front to back).

(a) Double hashing

```
insert("X")  h("X") = 1; g("X") = 4
insert("Y")  h("Y") = 7; g("Y") = 2
insert("Z")  h("Z") = 1; g("Z") = 5
```

0	1	2	3	4	5	6	7	8	9
▨	▨	▨	▨	M	▨	V	L	▨	P

(b) Linear probing/Deletion

```
insert("X")  h("X") = 4
delete("B")  h("B") = 2
insert("Z")  h("Z") = 3
```

0	1	2	3	4	5	6	7	8	9
W	▨	F	M	B	J	V	▨	▨	P

Figure 2: Hashing.

(b) (8 points) Draw S's suffix tree. List children alphabetically (`"a" < "b" < "$"`). Label each leaf with its suffix index (from 0 to 7). Also, label each internal node with its number of descendent leaves.

(c) (3 points) We want to know how many occurrences of the substring `"ba"` occur in S. Which node(s) of the suffix tree provide the answer to this query?

Problem 5. (20 points: 2–6 points for each part) 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. 3(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. 3(c) intersects 5 segments of L.)

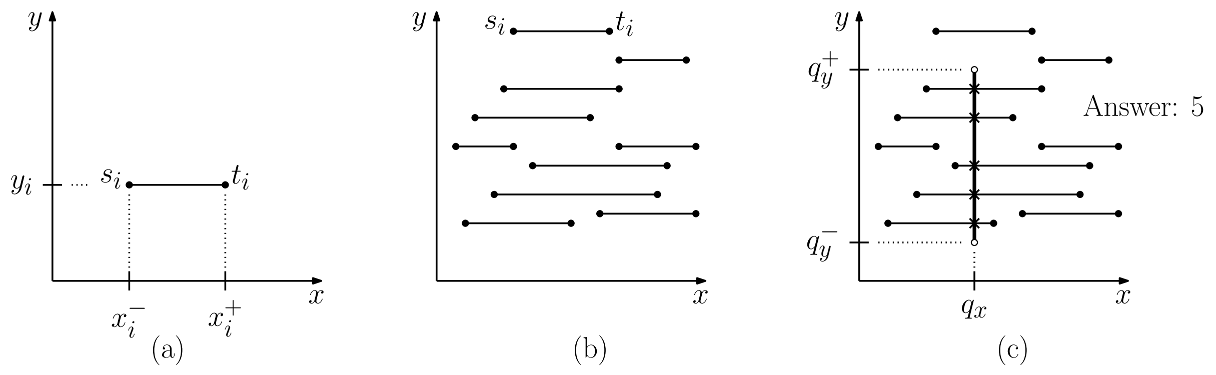


Figure 3: Segment stabbing queries.

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) What conditions must a horizontal segment's y-coordinate (y_i) satisfy to intersect the query segment?

(b) What conditions must a horizontal segment's left endpoint (x_i^-) satisfy to intersect the query segment?

(c) What conditions must a horizontal segment's right endpoint (x_i^+) satisfy to intersect the query segment?

(d) Based on parts (a)–(c), briefly explain the structure of range tree to answer segment stabbing queries. (Query processing comes later.)

(e) Briefly explain how to apply your structure from (d) to answer segment stabbing queries.

(f) Given that there are n segments, what is the space and query time of your data structure?

Problem 6. (15 points) 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) (10 points) 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) (5 points) 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)**

Problem 7. (15 points) In this problem you will write a program to check the validity of an AVL tree. The node structure is given below. All members are public.

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

In order to be valid, every node `p` of the tree must satisfy the following conditions:

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

Present pseudocode for a function `boolean validAVL(AVLNode root)`, which returns `true` if the tree structure at the given `root` node is a valid AVL tree and `false` otherwise. Explain how your function works. For full credit, your function should run in time $O(n)$, where n is the number of nodes in the tree. You may *not* assume the existence of complex utility functions (e.g., for sorting, performing tree traversals, or computing tree heights).

Hint: Use recursion. You may write additional helper functions.

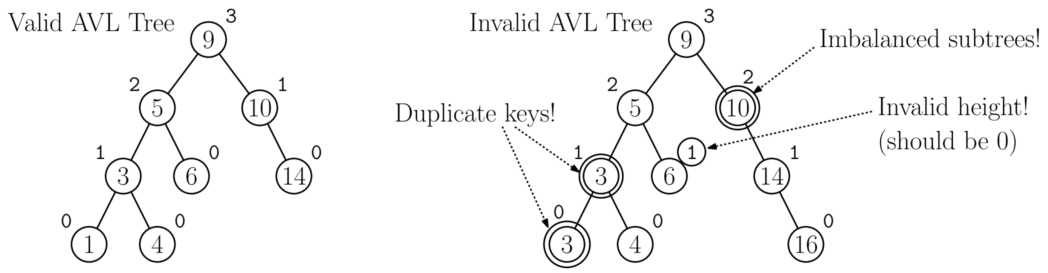


Figure 4: Valid and invalid AVL trees.