CMSC 420: Short Reference Guide

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

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

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

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

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

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

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

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

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

Sorting: The following algorithms sort a set of n keys over a totally ordered domain. Let $[m]$ denote the set $\{0, \ldots, m\}$, and let $[m]^k$ denote the set of ordered k-tuples, where each element is taken from $[m]$.

A sorting algorithm is *stable* if it preserves the relative order of equal elements. A sorting algorithm is *in-place* if it uses no additional array storage other than the input array (although $O(\log n)$ additional space is allowed for the recursion stack). The *comparison-based algorithms* (Insertion-, Merge-, Heap-, and QuickSort) operate under the general assumption that there is a *comparator function* $f(x, y)$ that takes two elements x and y and determines whether $x < y$, $x = y$, or $x > y$.

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

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

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

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

Unordered Dictionary: (by randomized 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-key, 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) Any sequence of n insert, extract-min, union, decrease-key can be done in $O(1)$ amortized time each. (That is, the sequence takes $O(n)$ total time.) Extract-min and delete take $O(\log n)$ amortized time. Make-heap from n keys in $O(n)$ time.

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 2021

Programming Assignment 0: Tour and Locator

Handed out: Tue, Feb 2. Due: **Sun, Feb 14 (11:00pm)**. (Fair warning: Don't wait until too late, since TA support will be limited over the weekends.) See the course syllabus for the late policy. Will be discussed in class on Tue, Feb 2.

Overview: Our programming project this semester will involve implementing algorithms for efficiently computing transportation tours for a set of points in 2-dimensional space. A *tour* is a cycle that visits all the points of some set. Computing efficient tours is fundamental to many transportation applications. This assignment will be a short warm-up exercise, which is designed to (re)familiarize yourself with Java programming, and to gain practice with the submission and grading process. You will not need to implement any fancy data structures.

Tours: Let $P = \{p_1, \ldots, p_n\}$ be a set of n points in two-dimensional space $\mathbb{R}^2$ (see Fig. 1(a)). A *tour* is defined to be a cycle that visits each point of P exactly once (see Fig. 1(b)). The most famous example is the *travelling salesperson problem* (TSP), a famous NP-hard problem that involves computing the tour of minimum length (see Fig. 1(c)).

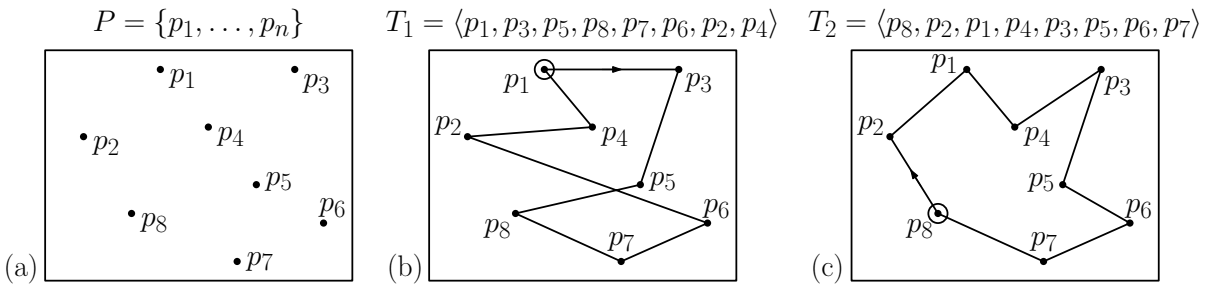


Figure 1: (a) A point set P, (b) a tour of P starting at p_1, and (c) another tour starting at p_8.

Representing a tour: Given a set of points P, we can represent a tour simply as a list (or more conveniently in Java as an `ArrayList`) of points. The associated tour is defined to be the cycle defined by visiting each point of the list in order, returning finally from the last point to the first.

Modifying through reversals: One way to modify any tour is by reversing an arbitrary sublist. For example, consider the tour shown in Fig. 2(a). Let us assume that the points of the tour are indexed from 0 to $n-1$, and let i and j be any two indices, where $0 \leq i < j \leq n-1$. We can form a new tour by reversing the sublist running from indices i through j (see Fig. 2(b)). This has the effect of replacing two edges $(i-1, i)$ and $(j, j+1)$ with the edges $(i-1, j)$ and $(i, j+1)$, and reversing all the edges in between.

Tour Object: In this assignment, you will implement a simple data structure, called `Tour`, that will maintain a tour for a set of points. Among other things, it will support sublist reversals as shown in Fig. 2. It will support other operations, such as the ability to add new points to tour, and various methods for listing out the points in the current tour.

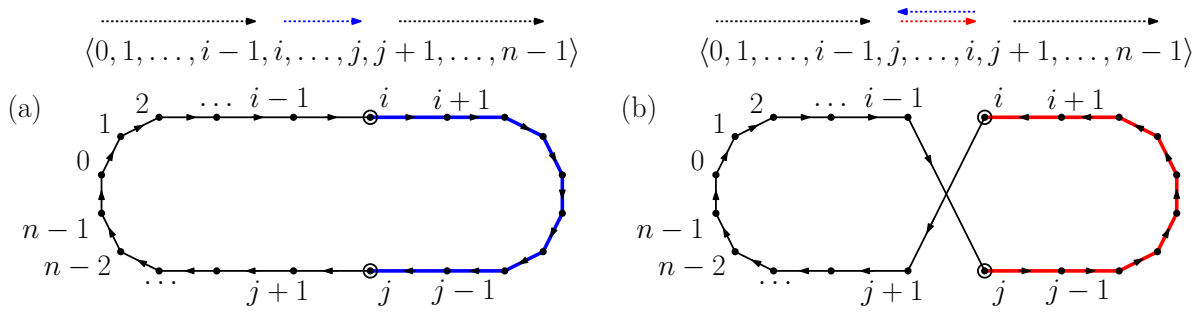


Figure 2: (a) A tour and (b) result of reversing the subtour from i to j.

Labeled Points: In order to refer to points in our data structure, we will associate each one with a string label. For example, if the points are airports, it will be convenient to identify each by its 3-letter IATA code. For example, Los Angeles International airport is given by the code "`LAX`" and Dulles International Aiport by "`IAD`". Its coordinates are given by its longitude and latitude. Each airport will also be associated with other information (its name, city, country, latitude and longitude), but we will not be using these for this assignment.

Of course, it would be too restrictive to build a data structure that works only on airport objects. The `Tour` class will be designed so that it can be applied to any generic type, called a *labeled point*. Such an object is defined by its (x, y) coordinates (of type `float`) and its *label* (of type `String`). A labeled point is any Java class that implements the following Java interface:

```
public interface LabeledPoint2D {
    public float getX(); // get point's x-coordinate
    public float getY(); // get point's y-coordinate
    public String getLabel(); // get the label
    // ... (and a few other methods, which we won't worry about now)
}
```

The `Tour` class (which you will implement) will be a generic Java object based on a type `Point`. This type can be any Java object that implements the `LabeledPoint2D` interface. In particular, our `Airport` class (which we will provide you) does this.

```
Airport.java: (We will provide this)
    public class Airport implements LabeledPoint2D { // An Aiport is a labeled point
        // ...
    }

Tour.java: (You will fill in the details here)
    public class Tour<Point extends LabeledPoint2D> { // A Tour stores labeled points
        // ... (You will fill in the rest of this)
    }

SomeApplication.java: (We will also provide this)
    ...
    Tour<Airport> theTour; // This stores a tour of Airports
```

I'm really confused! This is a lot to take in, so don't worry too much if this is a bit confusing. The program you need to write is actually pretty short. We will provide you with a skeleton implementation containing all of the above. All that you will need to do is fill in one file, `Tour.java`, which implements all the operations we ask of you. We will even provide the function declarations, and you just need to fill in their contents.

Public Interface: Here is a formal definition of the public interface of the `Tour<Point>` class:

`Tour():` This constructor performs whatever initializations are needed to create an empty tour. For our purposes, a tour can be represented as an expandable array, say a Java array-list, containing objects of type `Point`, that is, `ArrayList<Point>`. Thus, your constructor might create a new (empty) array-list object.

`String append(Point pt):` This appends a labeled point `pt` to the end of the current tour (e.g., by appending it to the aforementioned array-list).

It returns a string that summarizes the result of the operation. This string starts with the prefix `"append(XXX):␣"`, where "`XXX`" is the label associated with the point and "␣" denotes a single space. If the tour already contains a point with this same label, this is followed by the string `"Error - Label exists (operation ignored)"`. Otherwise, it adds this point to the end of the current tour. Letting i denote the index where the new point is placed, the prefix is followed by the string `"Added to tour at index` i`"`. (In standard Java style, we assume that indexing starts at zero.) An example is shown in Table 1. (Input lines have been shortened.)

Table 1: Example of commands and output strings.

Input:	**Output:**
`append:IAD: ...`	`append(IAD): Added to tour at index 0`
`append:BWI: ...`	`append(BWI): Added to tour at index 1`
`append:LAX: ...`	`append(LAX): Added to tour at index 2`
`append:IAD: ...`	`append(IAD): Error - Label exists (operation ignored)`
`list-tour`	`list-tour: 0:IAD 1:BWI 2:LAX`
`list-labels`	`list-labels: BWI:1 IAD:0 LAX:2`
`index-of:LAX`	`index-of(LAX): 2`

`String listTour():` This operation returns a string containing all the labels of the points in tour order. Each label is preceded with the index of this point in the tour. The output string starts with the prefix `"list-tour:␣"`, and it followed with a blank-separated sequence of the form "i`:XXX`", where i is the index (ranging from 0 up to $n-1$), and "`XXX`" is the label associated with this point. An example is given above.

`String listLabels():` This operation returns a string containing all the labels of the points in the tour in alphabetical order of the labels. Each label is succeeded with the index of this point in the tour. The output string starts with the prefix `"list-labels:␣"`, and it followed with a blank-separated sequence of the form "`XXX:`i", where "`XXX`" is the label associated with this point, and i is its index in the tour. An example is given above.

`String indexOf(String label):` This operation finds the point with the given label and returns its index in the tour. The output string starts with the prefix `"index-of(XXX):␣"`,

where "XXX" is the given label. If no point with the given label appears in the tour, this is followed with the string "Not-found". Otherwise, it is followed with the index of the point in the tour.

`String reverse(String label1, String label2)`: This operation reverses the subtour between the two given labels. The output starts with the prefix "reverse(XXX,YYY):␣", where "XXX" is the first label and "YYY" is the second label. Assuming that there are two distinct points in the tour with these labels, let i and j denote their indices in the tour. Swap i and j if needed so that $i < j$. Then reverse the order of points in the sublist from i to j, as shown in Fig. 2. Following the prefix, output "Successfully reversed subtour of length k", where k is the number of points in the sublist that was reversed. Some examples are shown in Table 2.

There are a few error cases to consider, which are processed in the following order:

- If no point of the tour has label XXX, then the prefix is followed by "Error - Label XXX does not exist (operation ignored)"
- If no point of the tour has label YYY, then the prefix is followed by "Error - Label YYY does not exist (operation ignored)"
- If XXX and YYY are the same, then the prefix is followed by "Error - Labels are equal (operation ignored)"

Table 2: Example of reverse operations.

Input:	**Output:**
list-tour	list-tour: 0:IAD 1:BWI 2:LAX 3:DCA 4:JFK 5:ATL 6:SFO
reverse:BWI:ATL	reverse(BWI,ATL): Successfully reversed subtour of length 5
list-tour	list-tour: 0:IAD 1:ATL 2:JFK 3:DCA 4:LAX 5:BWI 6:SFO
reverse:IAD:IAD	reverse(IAD,IAD): Error - Labels are equal (operation ignored)
reverse:LAX:CDG	reverse(LAX,CDG): Error - Label CDG does not exist (operation ignored)
reverse:DFW:CDG	reverse(DFW,CDG): Error - Label DFW does not exist (operation ignored)
reverse:DFW:DFW	reverse(DFW,DFW): Error - Label DFW does not exist (operation ignored)

Locators: This completes the description of the input/output behavior of the program. There is, however, an issue related to the program's efficiency. Consider the operation `indexOf` described above. Based on our description so far, this operation would take worst-case time $O(n)$ to implement on a tour of length n, since it would involve searching through the entire tour to find the point with the given label. We would like to do better.

A common issue arising in data structure design is that we insert an object in the data structure at one time and later we wish to locate where this object appears our the data structure. We would like the `indexOf` operator to run in at most $O(\log n)$ time. Our suggestion on how to achieve this is to employ a Java `TreeMap` to store a collection of key-value pairs, where the key is the point's label and the value is the point's index in the tour. For example, given the tour from Table 1, this map would store the pairs $\{(\texttt{BWI}, 1), (\texttt{IAD}, 0)(\texttt{LAX}, 2)\}$. Now, to perform the operation `indexOf`, we can search the `TreeMap` for the given label to retrieve the associated index in $O(\log n)$ time.

Note that whenever the index of a point changes, as can happen during a `reverse` operation, you will need to look up the point in the `TreeMap`, and modify its associated value based on the point's new index.

Running-time Requirements: Our grading of your program will involve an inspection of your code for the manner in which you implement the above operations. Assuming that the current tour contains n points, we require that the above operations run in the following worst-case asymptotic times:

`append:` $O(\log n)$ (amortized) time, assuming you can add an item to Java's `ArrayList.add()` in $O(1)$ amortized time and can add an entry to a tree map in $O(\log n)$ time.

`listTour:` $O(n)$ time.

`listLabels:` $O(n)$ time, assuming Java's `TreeMap.entrySet()` function returns the entries sorted by key value in this time.

`indexOf:` $O(\log n)$ time, assuming this is the search time of Java's `TreeMap`.

`reverse:` $O(k \log n)$ time, where k is the length of the tour being reversed. It should take only $O(k)$ time to perform the reversal, but k locator values in the `TreeMap` need to be updated.

If you prefer, you can use a `HashMap` instead of a `TreeMap`. Note that the run time of `listLabels` will go up to $O(n \log n)$, since the strings will need to be sorted. This is acceptable for full credit.

Program structure: We will provide a driver program that will input a set of commands. You need only implement the `Tour` class and the functions listed above. Here is the public interface:

```
package cmsc420_s21;

public class Tour<Point extends LabeledPoint2D> {
    public Tour() { }                           // Constructor
    public String append(Point pt) { ... }      // Append point to tour
    public String listTour() { ... }            // List in tour order
    public String listLabels() { ... }          // List in alpha order
    public String indexOf(String label) { ... } // Index of label
    public String reverse(String label1, String label2) { ... } // Reverse
}
```

Skeleton Code: We will provide you with some skeleton code to start with. This consists of the following:

`Tour.java:` **This is the only file you need modify.** A skeletal version of the main class for the extended binary search tree.

`Airport.java:` A class that stores information about airports.

`LabeledPoint2D.java:` The interface for the labeled point type.

`Point2D.java:` A small utility class for storing (x, y) coordinates.

`Tester.java`: Main program for testing your implementation. It inputs commands either from a file or standard input and sends output to another file or standard output. **(You may modify this file to select different input/output files.)**

`CommandHandler.java`: A class that processes commands that are read from the input file and produces the appropriate function calls to the member functions of your `Tour` class.

You may submit additional files as well, but it is not necessary. Other than `Tour.java` avoid modifying or reusing any of the above files, since we will overwrite them with our own when testing your program. Use the package "`cmsc420_s21`" for all your source files.

Testing/Grading: We will be using Gradescope's autograder and JUnit for testing and grading your submissions. All the tests and the expected results are visible. We will provide a link to the final test data on the class Projects page. We will check style and efficiency manually, and this will constitute 20% of the final score.

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. 3). Drag your file `Tour.java` into the window. If you generated other files, zip them up and submit them all. (You do not need to include the files from the skeleton code, included `Airport.java`, `LabeledPoint2D.java`, `Point2D.java`, `Tester.java`, and `CommandHandler.java`.) Select "`Upload`".

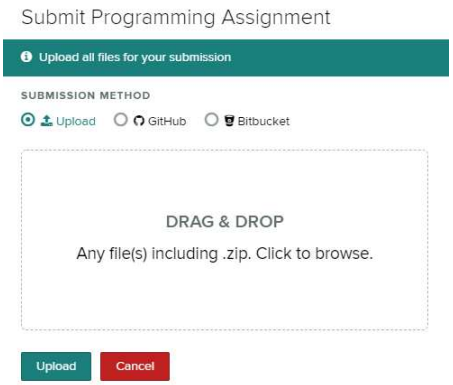


Figure 3: Gradescope submission.

After a few minutes, Gradescope will display the results (see Fig. 4). In this case, 20 out of 25 points are determined by the autograder, and we will assign the final 5 points based on inspecting the source code of your program for style and efficiency.

On the top-right of the page, it shows 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,

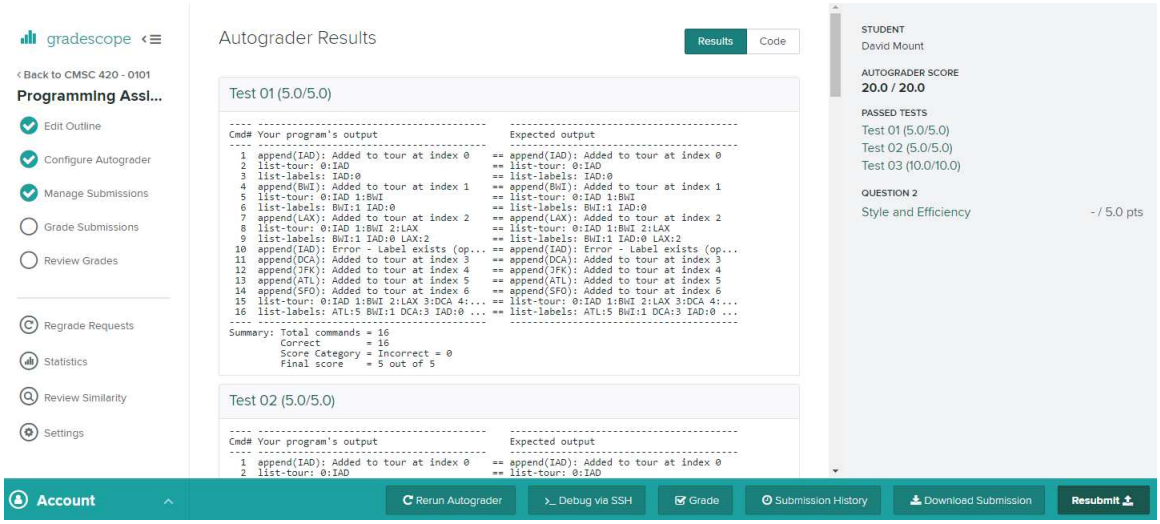


Figure 4: Gradescope autograder results (correct).

these will be highlighted (see Fig. 5). 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.

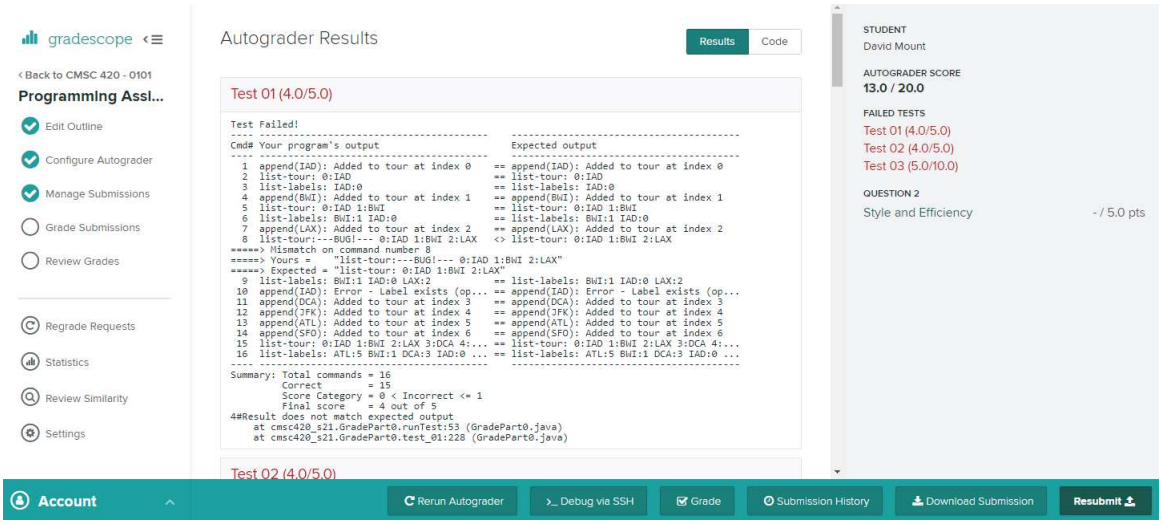


Figure 5: Gradescope autograder results (incorrect).

CMSC 420: Spring 2021

Programming Assignment 1: Extended AA Trees

Handed out: Tue, Mar 2. Part-1a due: **Wed, Mar 10, 11pm** and Part-1b due: **Mon, Mar 29, 11pm**.

Overview: In this assignment you will implement an extended variant of the AA Tree. Recall that an extended binary tree there are two different node types, *internal nodes* and *external nodes*. Extended trees are used in many applications. By separating node types, we can better tailor each node to its particular function. Data (that is, the key-value pairs) are stored only in the external nodes, which internal nodes only store keys, called *splitters*. Together, the internal nodes serve as an index, directing the search to the appropriate external node where the data is stored.

Our extended AA tree, or `AAXTree`, will be templated by two types `Key` and `Value`. Our only assumption regarding these types is that the `Key` type implements the Java `Comparable` interface, meaning that it defines a function `compareTo()` for comparing keys. In our test data, keys will be of type `String` and values of type `Airport`.

Extended AA Tree: Recall that a traditional AA tree is a binary variant of the 2-3 tree and is a close relative of red-black trees. Specifications on how to implement an extended version of this tree are given in the Supplemental Lecture on Extended AA Trees, which will be posted on the class's Projects Page. An example of such a tree is shown in Fig. 1(a).

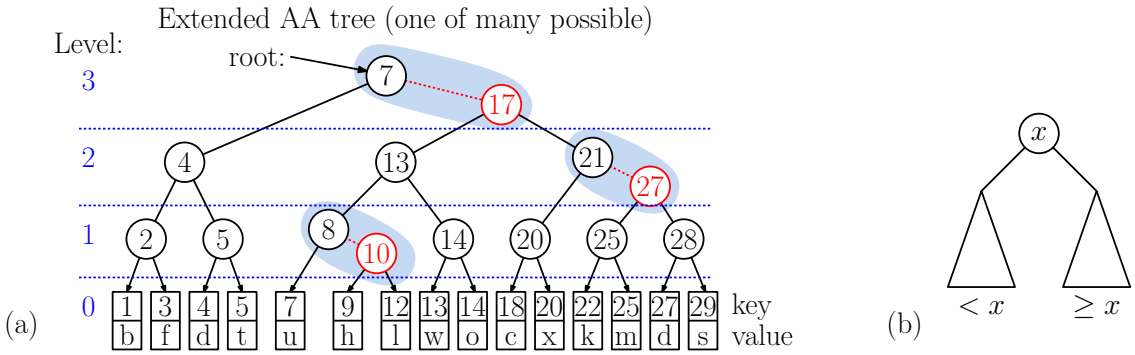


Figure 1: (a) An extended AA Tree storing the 15 key-value pairs $\{(1, \text{b}), (3, \text{f}), (4, \text{d}), \ldots\}$, (b) ordering convention. See the Projects Page for further information.

Part 1a Requirements: (Due Mon, Mar 8, 11pm - 30%) This first part involves the basic functionality to find, insert, list the dictionary's contents, and clearing the dictionary. Because our autograding program will check that your tree matches ours exactly, it is important that you follow the implementation described in the Supplemental Lecture on extended AA Trees.

`Value find(Key x)`: Determines whether there is a key-value pair (x, v), and if so returns a reference to v. Otherwise, it returns `null`.

`void insert(Key x, Value v) throws Exception`: Inserts key value (x, v), throwing an `Exception` with the message `"Insertion of duplicate key"` if there is a key-value pair in the dictionary with key x.

`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. The `key` and `value` refer to the key and value stored in the node. We assume that both provide a `toString` method. The `level` value refers to the node's level in the AA tree.

- Internal nodes: `"(" + key + ")␣" + level` (where "␣" is a space character)
- External node: `"[" + key + "␣" + value + "]"`

Our autograder program is sensitive to both case and whitespace. For example, given the tree of Fig. 1, a partial list of the `ArrayList` contents are shown in Fig. 2.

Index	Contents	Index	Contents	Index	Contents
0	`(7) 3`	5	`(5) 1`	10	`(8) 1`
1	`(4) 2`	6	`[4 d]`	11	`[7 u]`
2	`(2) 1`	7	`[5 t]`	12	`(10) 1`
3	`[1 b]`	8	`(17) 3`	13	`[9 h]`
4	`[3 f]`	9	`(13) 2`	14	`...`

Figure 2: Partial result from `getPreorderList` for the tree shown in Fig. 1(a).

`void clear()`: This removes all the entries of the tree (e.g., by setting the root pointer to `null`).

Part 1b Requirements: (Due Wed, Mar 24, 11pm - 70%) In this part you will implement the remaining operations.

`int size()`: Returns the number of key-value pairs in the dictionary. For example, for the tree of Fig. 1(a), this would return 15.

`void delete(Key x) throws Exception`: Deletes the entry with key x. If there is no such entry, it throws an `Exception` with the error message `"Deletion of nonexistent key"` if there is no key-value pair in the dictionary with key x.

`Value getMin()`: This returns the value associated with the smallest key of the dictionary. For example, on the tree of Fig. 1(a), this returns "`b`". If the dictionary is empty, these both return `null`.

`Value getMax()`: Same as `getMin`, but for the largest key of the dictionary.

`Value findSmaller(Key x)`: This returns the value associated with the entry having the largest key that is *strictly smaller* than x. (Note that x may or may not be in the dictionary.) If the dictionary is empty or if there is no key smaller than x, this returns `null`.

For example, on the tree of Fig. 1(a), `findSmaller(18)` would return "`o`", the value associated with the next smaller key 14. More generally, `findSmaller(x)` for $14 < x \leq 18$ returns "`o`". Finally, `findSmaller(1)` or generally `findSmaller(x)` for any $x \leq 1$ would return `null`.

`Value findLarger(Key x)`: Same as `findSmaller`, but for the next strictly larger key of the dictionary.

`Value removeMin()`: Deletes the entry from the dictionary associated with the smallest key, and returns its associated value. If the dictionary is empty, this returns `null`, and the dictionary is unchanged. As with any deletion operation, the tree should be rebalanced after the deletion. For example, on the tree of Fig. 1(a), `removeMin()` would delete the entry $(1, b)$ and return the value "`b`".

`Value removeMax()`: Same as `removeMin`, but for the largest key of the dictionary.

Skeleton Code: As in the first assignment, we will provide skeleton code on the class Projects Page. The only file that you should expect to modify is `AAXTree.java`. Use must use the package "`cmsc420_s21`" 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_s21;

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

    public AAXTree() { ... } // you fill these in
    public Value find(Key x) { ... }
    public void insert(Key x, Value v) throws Exception { ... }
    public void delete(Key x) throws Exception { ... }
    // ... and so on
}
```

Efficiency requirements: Except for `getPreorderList`, all operations should run in $O(\log n)$ time, where n is the number of entries in the data structure. The operation `getPreorderList` should run in time $O(n)$. We will check this by a manual inspection of your code.

Testing/Grading: Submissions will be made through Gradescope (you need only upload your modified `AAXTree.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. We will be checking the following items:

- You should use Java's class inheritance to implement your internal and external nodes.
- All operations (except `getPreorderList`) should be implemented so they run in $O(\log n)$ time.
- We will not check in detail for adherence to coding standards, but we may deduct points if your code is unusually complex of messy.

CMSC 420: Spring 2021

Programming Assignment 2: Wrapped k-d Trees

Handed out: Tue, Apr 6. Due: **Wed, Apr 21, 11pm**. (Submission via Gradescope.)

Overview: In this assignment you will implement a variant of the kd-tree data structure, called a *wrapped kd-tree* (or `WKDTree`) to store a set of points in 2-dimensional space. This data structure will support insertion, deletion, and a number of geometric queries, some of which will be used later in Part 3 of the programming assignment.

The data structure will be templated with the point type, which is any class that implements the Java interface `LabeledPoint2D`, as described in the file `LabeledPoint2D.java` from the provided skeleton code. A labeled point is a 2-dimensional point (`Point2D` from the skeleton code) that supports an additional function `getLabel()`. This returns a string associated with the point.

In our case, the points will be the airports from our earlier projects, and the labels will be the 3-letter airport codes. The associated point (represented as a `Point2D`) can be extracted using the function `getPoint2D()`. The individual coordinates (which are `floats`) 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 `WKDTree` will contain the following public class:

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

Wrapped kd-Tree: Recall that a kd-tree is a data structure based on a hierarchical decomposition of space, using axis-orthogonal splits. A *wrapped kd-tree* involves two modifications to the standard kd-tree (see Fig. 1).

Extension: As with the previous assignment, our tree will be an extended binary tree involving *internal nodes* and *external nodes*. Each external node stores a single point.

Each internal node stores the splitting information, consisting of a *cutting dimension* and a *cutting value*. The cutting dimension (or `cutDim`) indicates which axis (0 for x and 1 for y) is to be split, and the cutting value (or `cutVal`) indicates where the cut occurs along this axis (see Fig. 1). For example, if the cutting dimension is 0 (for x) and the cutting value is z, then a point $p = (p_x, p_y)$ will be put in the left subtree if $p_x < z$ and in the right subtree if $p_x \geq z$. Note that the cutting value does *not* need to be the coordinate of any point in the tree.[1]

Wrapping: In normal kd-trees, each node is associated with an axis-aligned rectangular cell, which is based on the splits that have been made by this node and its ancestors. In a wrapped kd-tree each internal node explicitly stores a *wrapper*, which is defined to be a

[1] In general, this is a useful feature because we can select splitting lines to optimize query processing. Although we will not take advantage of this fact in this project, this can be important in applications in high dimensional spaces as occurs in machine-learning applications.

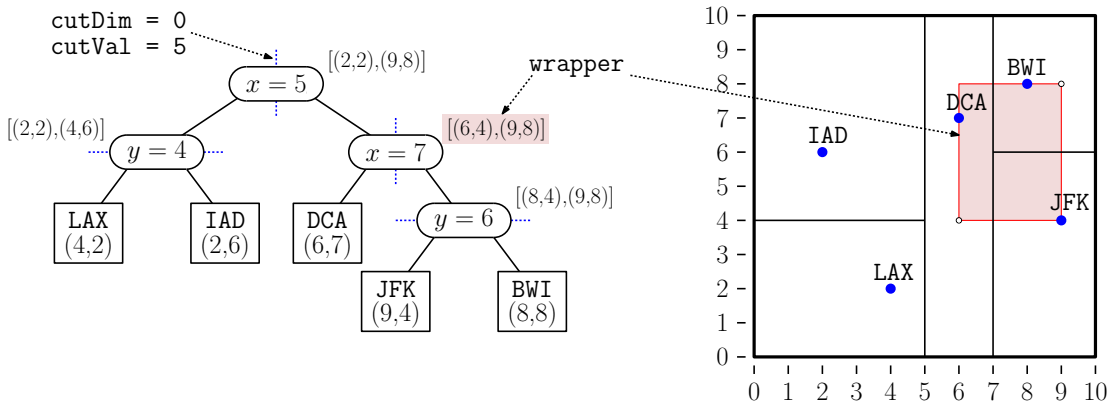


Figure 1: A wrapped kd-tree and the associated spatial subdivision.

minimum axis-aligned bounding box for the points in the subtree associated with this node. (In Fig. 1, the wrapper for the internal node "$x = 7$" is highlighted. A wrapper can be represented as any axis-parallel rectangle, say its lower-left and upper-right corner points.) We will provide a class `Rectangle2D` in the skeleton code, and a node's `wrapper` is of this type.

The use of wrappers modestly increases the storage requirements of the data structure, but query processing can be much faster because a wrapper can be significantly smaller than the associated cell. This means that query processing can do a better job of filtering out subtrees that cannot contribute to the search result. This feature becomes more significant as the dimension of the space increases. As points are inserted and deleted from the tree, the wrappers associated with nodes of the tree need to be updated accordingly.

Wrappers are only computed for internal nodes. (Each external node has an "implicit" wrapper consisting of the trivial rectangle that contains the associated point.) We can define a node's wrapper recursively as the smallest axis-aligned rectangle that contains the wrappers of the node's left and right children. (The class `Rectangle2D` provides a function `union` to perform this operation.)

Specifications on how to implement will be given in the Supplemental Lecture on Wrapped kd-Trees, which will be posted on the class's Projects Page.

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

In addition to the wrapped kd-tree, you will also need to provide a working version of the `AAXTree` from the previous assignment. We will use the `AAXTree` data structure as a *locator*. Whenever we insert a labeled point (e.g., airport "`DCA`" at coordinates $(6, 7)$), we will insert the key-value pair $\langle$`DCA`$, (6, 7)\rangle$ in the `AAXTree`. This way, if we need to determine the coordinates of any airport, we can look it up using its three-letter code. You do not need to do anything

other than provide the file, however. Our command handler will automatically insert each airport into both data structures.

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

`void insert(LPoint pt) throws Exception`: Inserts point `pt` in the tree. It throws an `Exception` with the message `"Insertion of point with duplicate coordinates"` if there is a point in the dictionary with the same coordinates. The insertion process is described in the supplementary lecture. (If there is a tie for the choice of the cutting dimension, x is preferred over y.)

`void delete(Point2D pt) throws Exception`: Deletes the entry whose coordinates match those of `pt`. If there is no such point, it throws an `Exception` with the error message `"Deletion of nonexistent point"`. The deletion process is described in the supplementary lecture.

Note that the above exception will *usually not* appear in your output. The reason is that the default deletion command is given an airport code (e.g., "`delete:DCA`"), and our command handler checks first whether this code exists within your `AAXTree`. If so, it invokes the appropriate deletion command (e.g., "`delete(Point2D(6,7))`"). There is a variant form of the delete command (`delete-point`) which calls your `WKDTree` delete function directly, but we use it sparingly since it causes the `AAXTree` and `WKDTree` to go out of sync with each other.

`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. The output for internal and external nodes has the following form (which must be matched exactly):

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

  ```
  "(x=" + cutVal + "): " + wrapper.toString()
  "(y=" + cutVal + "): " + wrapper.toString()
  ```

  where "␣" is a space character. The function `wrapper.toString()` is defined in `Rectangle2D.java`, which is part of the skeleton code. It outputs the lower-left and upper-right corners of the rectangle.
- External node: Letting `point` denote the labeled point stored in this node, this generates `"[" + point.toString() + "]"`. The function `point.toString()` is defined in `Airport.java`.

For example, here is the result for the tree of Fig. 1.

```
(x=5.0): [(2.0,2.0),(9.0,8.0)]
(y=4.0): [(2.0,2.0),(4.0,6.0)]
[LAX: (4.0,2.0)]
[IAD: (2.0,6.0)]
(x=7.0): [(6.0,4.0),(9.0,8.0)]
[DCA: (6.0,7.0)]
(y=6.0): [(8.0,4.0),(9.0,8.0)]
[JFK: (9.0,4.0)]
[BWI: (8.0,8.0)]
```

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

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

`int size()`: Returns the number of points in the tree. For example, for the tree of Fig. 1(a), this would return 5.

`LPoint getMinX()`: This returns a reference to the labeled point that is associated with the smallest x-coordinate in the tree. If two or more points have the same minimum x-coordinate, then the one with the smallest y-coordinate is returned. If the dictionary is empty, this returns `null`.

Analogously, there are functions `getMaxX()`, `getMinY()`, and `getMaxY()`, which perform the analogous operations. For the max versions, if there are ties for the principal coordinate, return the point with the largest "other" coordinate.

For example for the tree shown in Fig. 2(a), the calls `getMinX()` `getMaxX()`, `getMinY()`, and `getMaxY()` would return the labeled points "`SEA`", "`JFK`", "`LAX`", and "`SFO`" (respectively).

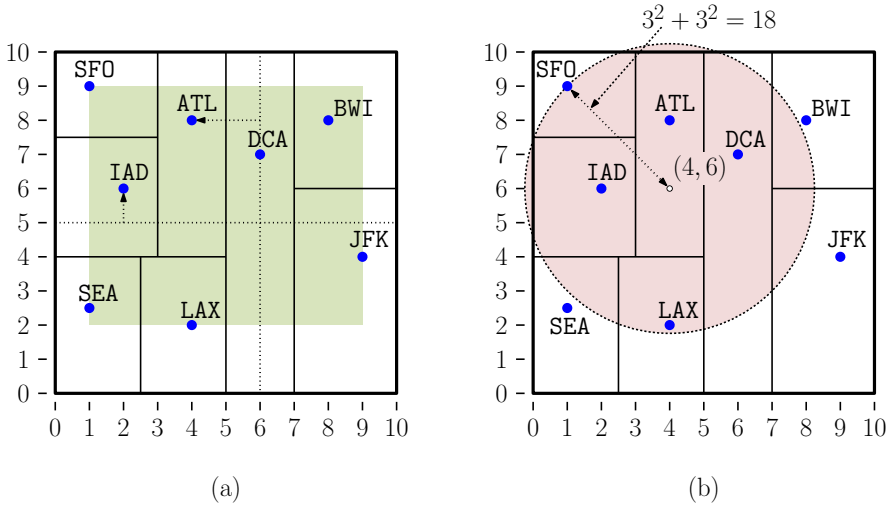


Figure 2: Queries on a wrapped kd-tree.

`LPoint findSmallerX(float x)`: Among all the points whose x-coordinates are *strictly smaller* than x, this returns a reference to the labeled point having the largest x-coordinate. (Note that there need not be a point having the coordinate x.) If the tree is empty or if there is no point whose x-coordinate is smaller than x, this returns `null`. If there are ties for the largest x-coordinate, return the point with the largest y-coordinate.

Analogously, there are functions `findLargerX()`, `findSmallerY()`, and `findLargerY()`. For the larger versions, if there are ties for the point with the smallest principal coordinate, return the point with the smallest "other" coordinate.

For example, for the tree of Fig. 2(a), `findSmallerX(6)` would return the labeled point "`ATL`", and `findLargerY(5)` would return the labeled point "`IAD`".

`ArrayList<LPoint> circularRange(Point2D center, float sqRadius)`: This function is given a circular disk, expressed as a center point `center` and a squared radius of

`sqRadius`. (So, to represent a disk of radius 5, we would set `sqRadius` to $5^2 = 25$.) This function returns an array-list containing all the points whose squared distance from the center point is at most `sqRadius`. (Thus if a point lies on the boundary of the disk, it will be included.) If there are no points in the disk, it should return an empty array-list (not `null`). The order of elements in the list does not matter (because we will sort it before outputting), but there should be no duplicates in the list.

To assist you, the `Point2D` and `Rectangle2D` classes both provide a utility function `distanceSq(Point2D pt)`, which computes the squared distance from the current object to point `pt`.

For example, in the tree of Fig. 2(b), `circularRange(Point2D(4,6), 18)` would return an `ArrayList` with the labeled points "`ATL`", "`DCA`", "`IAD`", "`LAX`", and "`SFO`" (in any order).

Why squared radius? The advantage of using squared distances over standard Euclidean distances is that we can avoid invoking the square-root function required by the standard distance (from the Pythagorean Theorem). Since our input points have integer coordinates, squared distances can be computed exactly as integers, which avoids floating-point round-off errors due to limited precision. (For example, the squared distance between $(4, 6)$ and $(1, 9)$ in Fig. 2(b) is 18, whereas the Euclidean distance is $4.242640687119285\ldots$.)

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You should replace the `AAXTree.java` file with your own, and you should add the implementation of the above functions to `WKDTree.java`. You should not modify any of the other files, but you can add new files of your own. For example, if you wanted to add additional functions to any of the classes, such as `Point2D` or `Rectangle2D`, it would be preferable to create an entirely new class (e.g., `MyRect2D`), which you will upload with your submission.

You must use the package "`cmsc420_s21`" 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_s21;

public class WKDTree<LPoint extends LabeledPoint2D> {

    public WKDTree() { ... } // you fill these in
    public LPoint find(Point2D pt) { ... }
    public void insert(LPoint pt) throws Exception { ... }
    public void delete(Point2D pt) throws Exception { ... }
    // ... and so on
}
```

Efficiency requirements: Unlike the AA-Tree, there are no worse-case guarantees on the running times of the above functions. Nonetheless, you should make a reasonable effort to implement

your functions in an efficient manner. (See the supplemental lecture notes for suggestions.) In particular, if it can be inferred (e.g., from the wrapper) that a node's subtree cannot possibly contribute to the query result, then the processing should immediately return, without visiting its children. Also, if there is one subtree that is obviously better to visit first than the other, your code should take advantage of this. Up to 10% of the final score will be based on this manual inspection of your code.

Testing/Grading: As before, we will be using Gradescope's autograder and JUnit for testing and grading your submissions. Because you will be submitting multiple files for this part, you should produce a zip file with the two principal files (`AAXTree.java` and `WKDTree.java`). You may include other files, but note that the files given in the skeleton code (e.g., `Point2D.java`, `Rectangle2D.java`, and so on) will be overwritten by the autograder. So, there is nothing to be gained by modifying these files.

As always, we will provide some sample test data and expected results along with the skeleton code. Note that some portion (up to 20%) of the final grade will be based on hidden tests.

CMSC 420: Spring 2021

Programming Assignment 3: Efficient TSP Heuristics

Handed out: Tue, Mar 27. Due: **Tue, May 11, 11pm**. (Submission via Gradescope.)

Overview: In this assignment we will combine our extended AA tree and wrapped kd-tree data structures to implement a data structure for maintaining traveling salesperson (TSP) tours. We are given a discrete set of points P in $\mathbb{R}^2$. Recall from Programming Assignment 0 that a *tour* of P is a cycle that visits all the points of P exactly once.

Squared Measure: The (standard) Euclidean TSP problem involves computing the tour over P of the minimum total Euclidean length. Euclidean distances involve square roots, and this results in rather unpredictable round-off errors. We will instead consider a variant of this problem, for the sake of easier testing.

Let T be a tour of P, and let $\langle p_0, p_1, \ldots, p_{n-1}\rangle$ denote the sequence of points along the tour. Define the *squared measure* of the tour, denoted $D^{[2]}(T)$ to be the sum of the squared distances of the edges of the tour, that is,

$$D^{[2]}(T) \;=\; \sum_{i=0}^{n-1} \text{dist}^2(p_i, p_{i+1}).$$

Throughout, indices are taken modulo n, so $p_n = p_0$. (Note that this is different from taking the square of the standard TSP length.) The squared measure has the advantage that if the coordinates P's points are all integers, then $D^{[2]}(T)$ is an integer. This is not generally true for the standard TSP measure.

Modifying Tours: Recall from Programming Assignment 0 that we can represent any tour as a list of points, and we can modify a tour by reversing a sublist. Given any two indices i and j, where $0 \le i < j \le n-1$. We can modify a tour by reversing the sublist from indices $i+1$ through j. (**Note:** We have changed the indexing slightly from Programming Assignment 0. This is a bit cleaner and more consist with established practice .) Let's call this operation reverse(i, j). This has the effect of replacing two edges $(i, i+1)$ and $(j, j+1)$ with the edges (i, j) and $(i+1, j+1)$, and reversing the path from $i+1$ through j (see Fig. 1).

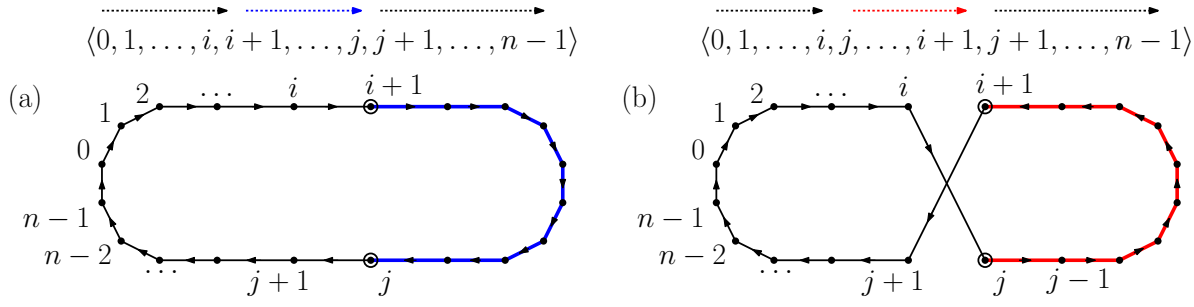


Figure 1: The operation reverse(i, j).

Note that this operation is not defined when $i = j$, but we can generalize it to any pair $i \neq j$ by performing reverse$(\min(i, j), \max(i, j))$.

Because the reversal only changes two edges, the change in cost is the difference between the squared lengths of the two new edges minus the square lengths of the original edges. Define the change in the measure to be:

$$\Delta(i, j) \;=\; (\text{dist}^2(p_i, p_j) + \text{dist}^2(p_{i+1}, p_{j+1})) - (\text{dist}^2(p_i, p_{i+1}) + \text{dist}^2(p_j, p_{j+1}))$$

Using this, we define a few other heuristics for modifying a tour:

2-Opt: Some reversals reduce the overall cost and some do not. Given a pair $0 \leq i, j \leq n-1$, where $i \neq j$, the operation 2-Opt(i, j) that conditionally performs a reversal if the squared measure decreases strictly. In particular, it first checks whether $\Delta(i, j) < 0$, and if so, it performs reverse(i, j). Otherwise, the tour is unchanged.

2-Opt-NN: There are clearly $O(n^2)$ possible 2-Opts that could be attempted on a tour. If the tour is close to optimum, the vast majority of these operations will not have any effect on the tour. How can we focus attention to 2-Opts that are most likely to reduce the tour's cost? There many different heuristics that could be applied. One idea for identifying 2-Opts that are likely to be effective is to let p_j be the closest point to p_i, assuming that it is closer than the neighbor p_{i+1} it replaces.

This gives rise to an important geometric query called a *fixed-radius nearest neighbor*. We will define this operation in the strict sense. Given a point set P and a query point q and a radius r, the problem is to compute the closest point of $p_j \in P$ to q, assuming that $\text{dist}(q, p_j) < r$ (see Fig. 2(a)). If there is no point of P within this distance bound, the query returns `null` (see Fig. 2(b)).

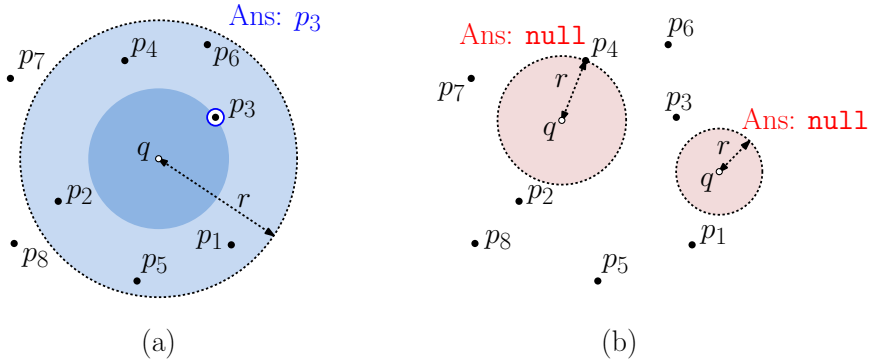


Figure 2: Fixed-radius nearest-neighbor queries.

In a 2-Opt-NN operation, we perform the operation 2-Opt(i, j), where p_j is the closest point to p_i. However, we only want to do this if the point p_j is closer than p_i's current successor p_{i+1} (indices taken modulo n). To find p_j we perform a fixed-radius nearest neighbor query for the query point $q = p_i$ and search radius.[1] $r = \text{dist}(p_i, p_{i+1})$. If we

[1]You might wonder why we need the radius constraint as part of the query. Why not just compute q's nearest neighbor, and then check afterwards whether the distance is smaller than r? The reason is that r is typically quite small, and there may be very few points lying within the query range. Thus, the radius constraint can significantly improve the query's efficiency.

receive a non-null result, p_j, we then perform 2-Opt(i, j). If the result is `null`, the tour is unchanged.

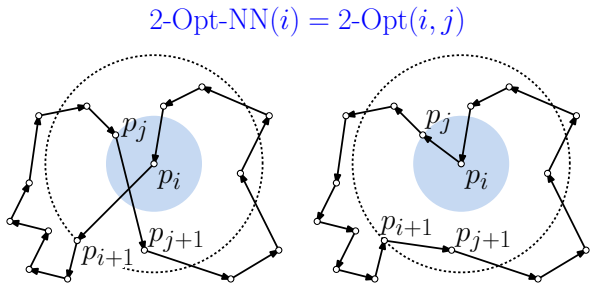


Figure 3: 2-Opt-NN operation.

We need to make one modification to the fixed-radius NN query. Clearly, the closest point p_i is p_i itself (not very useful!). So, the operation 2-Opt-NN(i) is defined formally as follows. First, apply a fixed-radius NN query with $q = p_i$, maximum distance $r = \text{dist}(p_i, p_{i+1})$, and ignoring p_i itself (or equivalently, any points at distance 0). Then perform 2-Opt(i, j).

What if there are multiple candidates for the nearest neighbor? For the sake of consistency, let's agree that the point to be selected is the one that is lexicographically smallest with respect to its coordinates.[2] That is, among all nearest neighbors, its x-coordinate should be the smallest, and among all that have the same x-coordinate, the y-coordinate should be the smallest.

All 2-Opt: In contrast to 2-Opt-NN, which performs 2-Opt on a judiciously chosen pair, this operation is pure brute force. It iterates through all indices i from 0 to $n-1$, and for all j from $i+1$ to $n-1$, and performs 2-Opt(i, j) for each pair. Thus, in total there are $\binom{n}{2} = O(n^2)$ instances of 2-Opt being performed.

Tour Object: In this assignment, you will implement a data structure, called `Tour`, that will maintain a tour for a set of points. It supports a number of operations, as described below. The data structure will be templated with the point type, which is any class that implements the Java interface `LabeledPoint2D`, as described in the file `LabeledPoint2D.java` from the provided skeleton code. A labeled point is a 2-dimensional point (`Point2D` from the skeleton code) that supports an additional function `getLabel()`. This returns a string associated with the point.

Each tour object will store three principal data elements:

Tour: This is a tour itself, that is, a list (e.g., Java `ArrayList`) containing the points (`LPoint`) of the tour.

Locator: This structure is used for locating the index of an airport in the tour from its code (e.g., "`LAX`"). It is a dictionary (implemented as an `AAXTree`) storing key-value pairs, where the keys are strings and the values are indices (represented as a Java `Integer`).

[2] We do not expect many test cases to check for this condition, so in your first pass, you might ignore this issue. The input file `test05-input.txt` has an instance where there are multiple candidates for the nearest neighbor.

The index for a given string gives the index of the corresponding labeled point in the tour. As with Programming Assignment 0, whenever we insert a new point into our tour, we need to record its location, and whenever we move a point to a new index (e.g., through reversal), we need to update its location.

At a minimum, the locator will need to support the operations of `insert`, `find`, `clear`, and a new operation called `replace` (which was not required in Programming Assignment 1). The `replace` operation has the following signature:

```
void replace(Key x, Value v) throws Exception
```

It is given a key `x` (that is, an airport code) and an associated value `v` (that is, an index in the tour). It searches for `x`. If it is not found, it throws an exception with the error message `"Replacement of nonexistent key"`. Otherwise, the value associated with this entry is changed to `v`.

Spatial Index: This is a 2-dimensional spatial index (implemented as a `WKDTree`) storing the points (`LPoint`). At a minimum, it must support the operations `insert`, `find`, `clear`, and a new operation called `fixedRadNN` (which was not required in Programming Assignment 2). The `fixedRadNN` operation has the following signature:

```
LPoint fixedRadNN(Point2D q, double sqRadius)
```

It returns a reference to the fixed-radius nearest-neighbor query to q, where the squared radius of the disk is `sqRadius`. Among the points whose squared distance to q is strictly more than zero and strictly less than `sqRadius`, it returns the closest to q. If there is no such point in the disk, it returns `null`. If there are multiple points at the same distance, your function should return that is lexicographically smallest in terms of its x and y coordinates. (That is, if two points are at each distance to q, prefer the one with the smaller x-coordinate. If both have the same x-coordinate, prefer the one with the smaller y-coordinate.)

Tour Operations: The Tour object should support the following public functions.

`Tour()`: Initializes an empty tour, creating the tour, locator, and spatial index (all empty).

`void append(LPoint pt) throws Exception`: Appends the labeled point `pt` to the end of the tour. If there exists a point with this label, an exception with the error message "`Duplicate label`" is thrown. If there already exists a point with the same coordinates, an exception with the error message "`Duplicate coordinates`" is thrown. Otherwise, the point is added to the tour, its index is added to the locator, and the point is added to the spatial index.

`ArrayList<LPoint> list()`: This returns a Java `ArrayList` containing all the points of the tour in order.

`void clear()`: The clears everything: the tour, the locator, and the spatial index.

`double cost()`: The returns the current squared measure of the tour. For the sake of consistency and accuracy, you should perform all arithmetic operations using `double` variables, and use the `Point2D` function `distanceSq` to compute distances between points.

`void reverse(String label1, String label2) throws Exception`: This begins by locating the indices i and j for the tour points with labels `label1` and `label2`, respectively.

If either label is not found in the locator, an exception with the error message "`Label not found`" is thrown. If $i == j$ (or equivalently, if the labels are equal), an exception with the error message "`Duplicate label`" is thrown. Otherwise, the operation reverse(i, j) is performed on the tour. (It may be that $i < j$ or $j < i$. Your function should work correctly in either case.)

`boolean twoOpt(String label1, String label2) throws Exception:` This is the same as `reverse` above, but after checking the validity of the arguments, instead of reverse, the operation 2-Opt(i, j) is performed on the tour. That is, we check whether $\Delta(i, j) < 0$ (note that the inequality is strict), and if so, we perform reverse(i, j). If the reversal is performed, the operation is said to be *effective*. If the operation is effective, we return `true`, and otherwise we return `false`.

`LPoint twoOptNN(String label) throws Exception:` This first locates the index i for the tour point with label `label`. If this label is not found in the locator, an exception with the error message "`Label not found`" is thrown. Otherwise, the operation 2-Opt-NN(i) is performed on the tour. That is, we invoke `fixedRadiusNN(q,rsq)` where $q = p_i$ and $rsq = \text{dist}^2(p_i, p_{i+1})$ (where p_{i+1} is the point immediately following p_i in the tour). If it returns `null`, then we return `null`. Otherwise, let p_j denote the result. We invoke 2-Opt(i, j). If it is effective, then we return a reference to the point p_j. Otherwise, we return `null`.

`int allTwoOpt():` This performs the operation all-2-Opt() on the tour. (For consistency in testing, this should be done exactly as described for all i from 0 to $n - 1$ and all j from $i + 1$ to $n - 1$, performing 2-Opt(i, j).) Among the $\binom{n}{2}$ 2-Opts performed, return the number that were effective.

Hint on Helpers: In order to implement the above functions, you may define whatever local helper functions you like. The functions above take labels as inputs, but it is more natural to work with tour indices. We would recommend that for each of the above label-based functions, you have a local index-based function to perform the actual operation. For example, the helper for `reverse` might be called `void reverseHelper(int index1, int index2)`, where `index1` and `index2` are the indices in the tour for the respective labels. The advantage of doing this is that your other helper functions can easily invoke one another.

Doubles not Floats: For some of the larger test cases we are planning to use, the number of digits in the tour costs will be too large to store in a single float variable. For the sake of testing, we have converted all the instances of `float` in the supporting classes (e.g. `Airport.java`, `Point2D.java`, `Rectangle2D.java`) to be of type `double`. You may need to make a similar change in your `WKDTree.java` to keep the compiler from complaining.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. We will also provide canonical versions of our `AAXTree` and `WKDTree` implementations. (Note that you will still need to add the new functions `replace` and `fixedRadiusNN`.)

```
package cmsc420_s21;

public class Tour<LPoint extends LabeledPoint2D> {
    public Tour() { /* you fill these in */ }
```

```
    public void append(LPoint pt) throws Exception { /* ... */ }
    public ArrayList<LPoint> list() { /* ... */ }
    // ... and so on
}
```

Efficiency requirements: The new `AAXTree` operation `replace` should run in $O(\log n)$ time. The new `WKDTree` operation `fixedRadNN` should be reasonably efficient, in the sense that the code should check each node's wrapper. If the wrapper for some node does not overlap the query disk or is farther away than the closest point seen so far, it should not recursively visit a node's children.

Testing/Grading: As before, we will be using Gradescope's autograder and JUnit for testing and grading your submissions. You can just drag your files `AAXTree.java`, `WKDTree.java`, and `Tour.java` into the Gradescope upload window. You may include other files, but note that the files given in the skeleton code (e.g., `Point2D.java`, `Rectangle2D.java`, and so on) will be overwritten by the autograder. So, there is nothing to be gained by modifying these files.

As always, we will provide some sample test data and expected results along with the skeleton code. Note that some portion (up to 20%) of the final grade will be based on hidden tests.

CMSC 420: Spring 2021

Homework 1: Basic Data Structures and Trees

Handed out Fri, Feb 12. Due at **11:00pm, Mon, Feb 22**. Indicated point values are approximate. Before writing your answers, please see the notes at the end about submission instructions.

Problem 1. (8 points)

(1.1) (4 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the "first-child/next-sibling" form.

(1.2) (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.

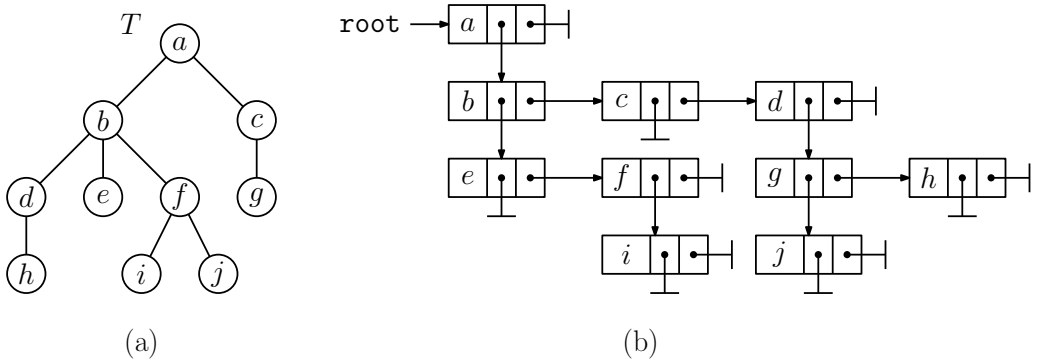


Figure 1: Rooted tree to first-child/next-sibling form and vice versa.

Problem 2. (3 points) Draw the binary tree of Fig. 2(a) with inorder threads added.

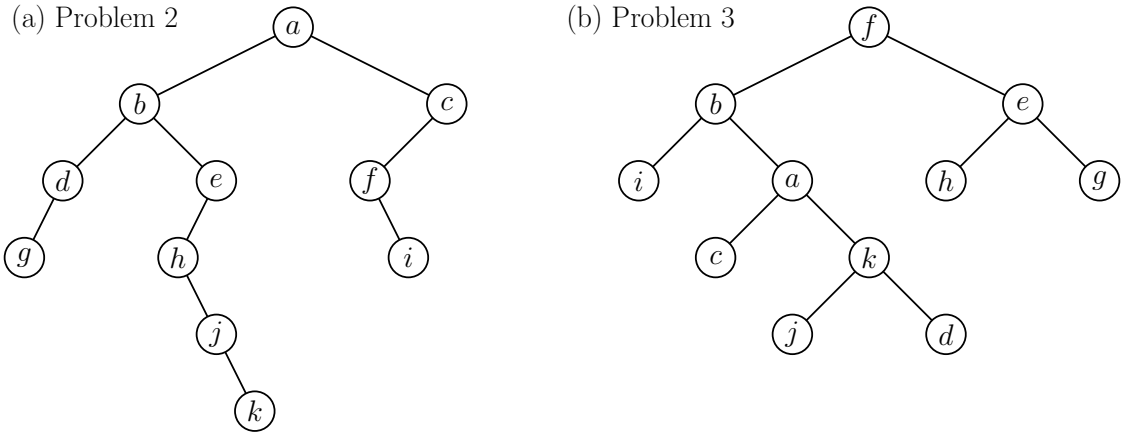


Figure 2: (a) Adding inorder threads to a binary tree and (b) a full binary tree.

Problem 3. (15 points) You have a full binary tree, where each node is labeled with a distinct letter. (Recall that a binary tree is *full* if each non-leaf node has exactly two children.) Throughout this problem, we restrict attention to full binary trees.

(3.1) (3 points) Someone has performed a *postorder* traversal, and gave you a list of the node labels. (For example, in the tree shown in Fig. 2(b), this is $\langle i, c, j, d, k, a, b, h, g, e, f\rangle$.) Is it generally possible to uniquely recover a full binary tree from its postorder sequence? If yes, explain how by presenting an algorithm for doing so. If no, draw two different labeled full binary trees where the postorder lists are the same.

(3.2) (3 points) Repeat question (3.1), but this time list has been modified so that each leaf node has been "flagged" to distinguish leaves from internal nodes. (For example, in the tree shown in Fig. 2(b), if we use "$*$" to indicate a leaf, this would be $\langle i^*, c^*, j^*, d^*, k, a, b, h^*, g^*, e, f\rangle$.)

(3.3) (3 points) Repeat (3.1), but this time for an *inorder* traversal of a full binary tree. (For example, in the tree shown in Fig. 2(b), this would be $\langle i, b, c, a, j, k, d, f, h, e, g\rangle$.)

(3.4) (3 points) Repeat (3.2), but this time for an *inorder* traversal of a full binary tree. (For example, in the tree shown in Fig. 2(b), this would be $\langle i^*, b, c^*, a, j^*, k, d^*, f, h^*, e, g^*\rangle$.)

(3.5) (3 points) We won't ask you to solve the remaining case (of a *preorder* traversal), but you suppose you discuss the unflagged case (3.1) with your best friend. (You both suspect that the evil Prof. Mount may put this question on a future exam.) This friend announces that the answer is "no" and tells you that there is a simple 6-node counterexample. Without even seeing the counterexample, you tell your friend this is wrong! How is this possible? (Assume for the sake of this problem that you are not a psychic.)

Problem 4. (8 points) You are given two $n \times n$ matrices A and B, where (following Java's convention) the rows and columns are indexed from 0 to $n-1$. Their product $A \cdot B$ is an $n \times n$ matrix C where for $0 \le i, j \le n-1$, $C[i,j] = \sum_{k=0}^{n-1} A[i,k] \cdot B[k,j]$.

(4.1) (5 points) Assume that A and B are represented sparse matrices (see Lecture 2 and Fig. 3). Present an efficient algorithm for computing the product $A \cdot B$.

To simplify things, you may assume that the output matrix C is represented as a standard $n \times n$ 2-dimensional array, which has been initialized to zero. To make it possible to generalize your solution to the sparse case, you should fill in the nonzero entries of C in sequential order (e.g., top to bottom and left to right).

(4.2) (3 points) Derive the running time of your algorithm in terms of the following quantities: n, N_A and N_B, where N_A and N_B are the numbers of nonzero entries in the matrices A and B, respectively. (That is, state what the asymptotic running time is and present a proof or convincing explanation of you bound. Hint: In the special case when the matrices are dense, that is $N_A = N_B = n^2$, the running time should be $O(n^3)$.)

Note: At the end of the handout, I present a sample solution for matrix addition, to give you some idea of the amount of detail I expect.

Problem 5. (10 points) This problem involves the AVL tree shown in Fig. 4.

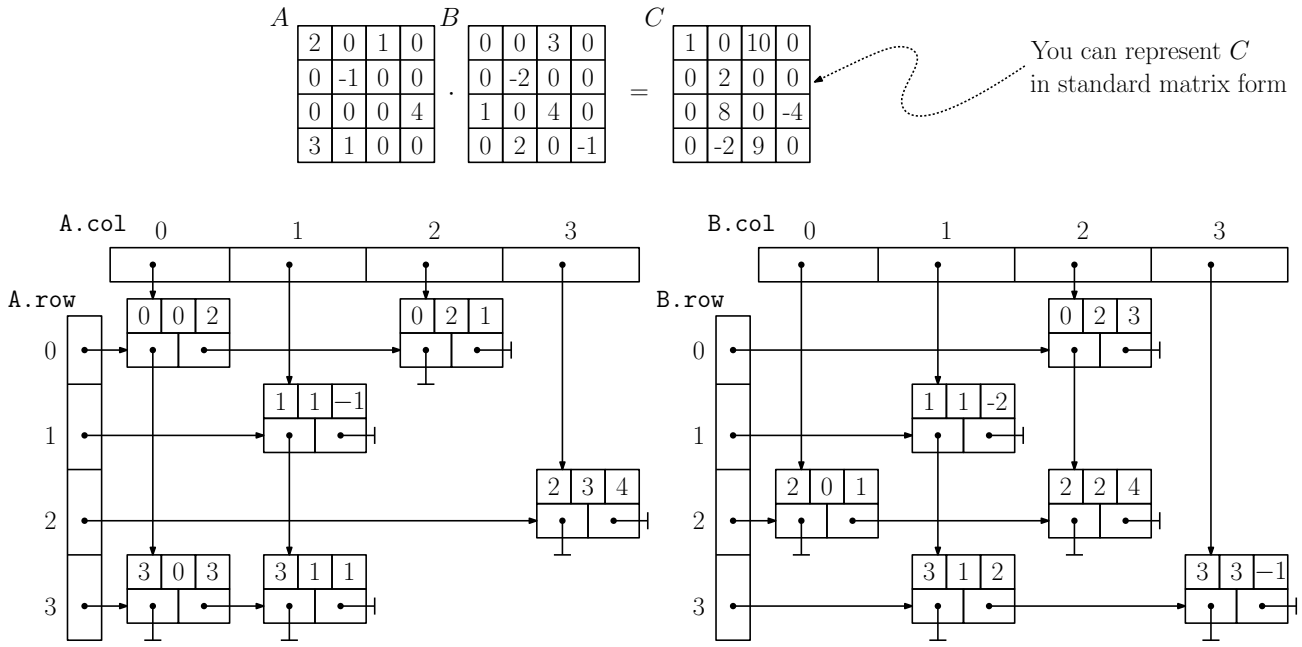


Figure 3: Sparse matrix multiplication.

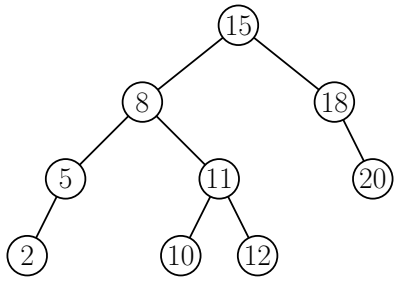


Figure 4: AVL Trees.

(5.1) (2 points) Redraw the tree, but label each node with the height of its subtree and its balance factor. Suggestion: For uniformity, use the convention from the Lecture 5 slides, where the height is written on the left of each node and the balance factor is written on the right.

(5.2) (4 points) Show the result of inserting the key 1 into this tree. First, show the result of inserting the new key into the tree (without any rebalancing) and show the updated balance factors working up from the inserted node to the first node where a rotation is needed. Second, show the final tree after rebalancing is done. Also show the final balance factors.

(5.3) (4 points) Repeat (5.2), but this time insert the key 13. (Do the insertion on the *original* tree from Fig. 4.)

Problem 6. (6 points) In our implementation of AVL search trees, we assumed that, in addition to the key and value, each node stored a pointer to its left and right child as well as the height of its subtree. Suppose that, in addition, we add a pointer to the node's *parent* in the tree. (The root node's parent pointer is set to `null`.) The new node structure is as follows, and the node constructor takes an additional argument specifying the parent:

```
class AVLNode {
   Key key;
   Value value;
   int height;
   AVLNode left, right, parent;

   AVLNode(Key x, Value val, int hgt, AVLNode lft, AVLNode rgt, AVLNode par) {
       // constructor - details omitted
   }
}
```

Present pseudocode for an `insert` function that inserts a new key, applies the appropriate rebalancing, and updates the parent pointers appropriately. As with standard AVL insertion, your function should run in time $O(\log n)$.

Hint: This can be messy if you don't approach it carefully. I believe that the cleanest solution is to find all places where a left or right child is changed (e.g., `p.left = q`) and fix the parent link right away. This works for *almost all* the cases where parent links need to be updated. Beware that `q` may be null, and you must never dereference `null`.

Note: Challenge problems are for fun. We grade them, but the grade is not used when grade cutoffs are determined. 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: You are given an array `A[1..n]` of real numbers (negative, positive, and zero). Design an efficient data structure to perform any sequence of the following two operations:

`void add(int i, float x)`: Given $1 \leq i \leq n$, and a strictly positive number x, this adds the value x to $A[i]$.

`float max(int m)`: Given $1 \leq m \leq n$, this returns the *maximum* value of the first m elements of A.

Notice that the number of elements remains fixed throughout (there are no insertions or deletions). Only the values may change. Each operation should take $O(\log n)$ time. (Hint: For full credit, as working storage, you can use an additional array `B[1..n]` of floats. If you don't see how to do this with just a single array `B`, for partial credit you can use any data structure of total space is $O(n)$. You shouldn't need any data structures beyond modifications of the ones we have seen so far in class.)

General note regarding coding in homeworks: When asked to present an algorithm or data structure, do **not** give complete Java code. Instead give a short, clean pseudocode description containing only the functionally important elements, along with an English description and a short example. (For example, I would prefer to see "$\lceil n/2 \rceil$" over "`(int) Math.ceil((double) n / 2.0)`".)

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. It is much easier to get an idea quickly from a figure than a segment of pseudo-code. I use Latex for text in conjunction with a figure editor called IPE for drawing figures.
- 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. 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, and 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 "PTO" 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 a scan 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.

Algorithm Solutions: Students often ask me how much detail am I expecting for questions that involve giving an algorithm. Whenever you are asked to present an "algorithm" your should present the following (even if I don't explicitly ask for all of this):

- A short English explanation
- Present the algorithm itself, typically in pseudocode
- If it is not obvious, briefly justify the algorithm's correctness
- Give a brief analysis of the running time

Below, I present a sample solution for the problem of matrix addition with the sparse-matrix representation. Let's assume that we are given two sparse matrices A and B (see Lecture 2), and our objective is to compute their matrix sum C. For simplicity, we will assume that C is given as a standard $n \times n$ matrix, which is initialized to zero, and all we need to do is to fill its the nonzero entries. Following Java's conventions, we assume that rows and columns are indexed from 0 to $n-1$. Recall that in the sparse-matrix representation, we are given two n-element arrays, call them `row[]` and `col[]`, where `row[i]` is the head of a linked list of the entries on row i, and `col[j]` is the head of a linked list of nodes in column j. These linked lists are sorted by increasing order of indices. Each nonzero matrix entry is represented by a node containing the following information:

```
class Node {
  int row      // this entry' row index
  int col      // this entry's column index
  float value  // the value of this matrix entry
  Node rowNext // the next entry (from left to right) in this row
  Node colNext // the next entry (from top to bottom) in this column
}
```

Sparse Matrix Addition: The algorithm iterates through each row $0 \leq i \leq n-1$, and then iterates in a coordinated manner through the two linked lists `A.row[i]` and `B.row[i]`. If both entries are in the same column, we compute their sum and store it in C. Otherwise, we copy the value that lies in the smaller column index. After processing an entry, we advance to the next element in the linked list. When we reach the end of either list, we simply copy the remaining entries of the other list to the appropriate entries in C.

Here is the pseudo-code. We'll omit braces and type specifications as much as possible.

```
void sparseAddition(SparseMatrix A, SparseMatrix B, float C[][])
    for (i = 0 to n-1)                      // iterate through all the rows
        ap = A.row[i]                       // head of A's ith row
        bp = B.row[i]                       // head of B's ith row
        while (ap != null && bp != null)    // while entries remain in both rows
            if (ap.col < bp.col)            // A's entry comes next?
                C[i][ap.col] = ap.value     //copy to C and advance
                ap = ap.rowNext
            else if (bp.col < ap.col)       // B's entry comes next?
                C[i][bp.col] = bp.value     // copy to C and advance
                bp = bp.rowNext
            else // (ap.col == bp.col)      // both in the same column?
                C[i][ap.col] = ap.value + bp.value // put sum in C and advance
                ap = ap.rowNext
                bp = bp.rowNext
      // At this point, only one list has elements remaining
      while (ap != null)                    // copy any remainder from A to C
          C[i][ap.col] = ap.value
          ap = ap.rowNext
      while (bp != null)                    // copy any remainder from B to C
          C[i][bp.col] = bp.value
          bp = bp.rowNext
```

The correctness follows from the fact that the two pointers `ap` and `bp` move in coordination, so one never gets too far ahead of the other. To obtain the running time, observe that we visit each of the nodes of the sparse matrix representations for A and B exactly once (when we are processing its row). Since there are N_A nodes in A and N_B nodes in B, this takes time $O(N_A + N_B)$. However, even if these quantities are both zero, will still need to access each of the n entries of `A.row` and `B.row`. Thus, the overall running time is $O(n + N_A + N_B)$.

CMSC 420: Spring 2021

Practice Problems for Midterm 1

Exam Logistics: Please read these now, so you don't have to waste time during the exam.

- This exam will be asynchronous and online. The exam will be made available through Gradescope for a 48-hour period starting at **12:00am the morning of Thu, Mar 11** and running through **11:59pm the evening of Fri, Mar 12**. The exam is designed to be taken over a 90-minute time period, but to allow time for scanning and uploading, you will have **2 hours** to submit the exam through Gradescope once you start it.
- The exam will be open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. (You may use a simple arithmetic calculator, but I don't expect that you will need one.)
- You may freely use any information from class, without citing it. Information from external sources (e.g., books or online sources) can be used without penalty, but you *must* cite your sources (e.g., "I found this on Stack Overflow") and express the answer in your own words.
- Please do not discuss any aspects of the exam with classmates during the exam's 48-hour time window, even if you have both submitted. This includes its content, difficulty, and length.
- If you have any questions during the 2-day exam window, please email me (mount@umd.edu) or make a *private* Piazza post. (Please post privately even if you have not yet started. Others in the class may be working on the exam.)
- If you are unsure about how to interpret a problem and I do not respond in a timely manner, please do your best. Write down any assumptions you are making. There will be no "trick" questions on the exam. Thus, if a question doesn't make sense or seems too easy or too hard, please check with me.
- Uploading a large image pdf will take time. Please allow sufficient time to submit your final work. While I do not want to penalize people for having slow network connectivity, *I reserve the right to deduct 2% of the final grade if you do not complete your upload within the 2-hour deadline.* (You can submit multiple times.)
- If you experience any technical issues while taking the exam, **don't panic**. Save you work (ideally in a manner that attaches a time stamp), and contact me by email (`mount@umd.edu`) as soon as possible. I understand that unforeseen events can occur, and I will attempt make reasonable accommodations.
- The exam will be long, and so be mindful of this. To get the most credit from each problem, on your first pass give just the answer and any required justification. If time permits, go back and fill in intermediate results and explanations to help with partial credit.

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

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. 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).

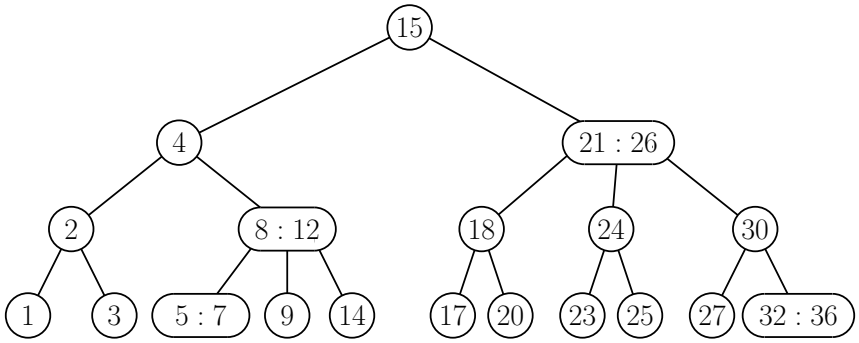


Figure 1: 2-3 tree insertion and deletion.

(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 may draw intermediate subtrees to help with partial credit, but don't waste too much time on this.

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 at most $\lceil \lg n \rceil$. **Explain briefly.**

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

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

(f) Repeat (e) in the case of *deletion* from an AVL tree. (You can give your answer as an asymptotic function of n.)

(g) You are given a 2-3 tree of height h, which you convert to 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?

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

(i) You have a set of n distinct keys, where n is a large even number. You randomly permute all the keys and insert the first $n/2$ of them into a standard (unbalanced) binary search tree. You then take the remaining $n/2$ keys, sort them in ascending order, and insert them into this tree. The final tree has n nodes. As a function of n, what is the expected height of this tree? (Select the best from the choices below.)

(i) $O(\log n)$
(ii) $O((\log n)^2)$
(iii) $O(\sqrt{n})$
(iv) $O(n)$

(j) 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.)

(k) 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).

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).

(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 a given an inorder threaded binary search tree T (not necessarily balanced). Recall that each node has additional fields `p.leftIsThread` (resp., `p.rightIsThread`). These indicate whether `p.left` (resp., `p.right`) points to an actual child or it points to the inorder predecessor (resp., successor).

Present pseudocode for each of the following operations. Both operations should run in time proportional to the height of the tree.

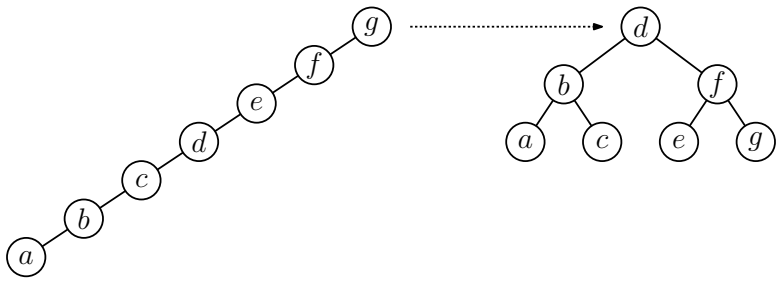


Figure 2: Rotating into balanced form.

(a) `void T.insert(Key x, Value v)`: Insert a new key-value pair (x, v) into T and update the node threads appropriately (see Fig. 3(a)).

(b) `Node preorderSuccessor(Node p)`: Given a non-null pointer to any node p in T, return a pointer to its *preorder successor*. (Return `null` if there is no preorder successor.)

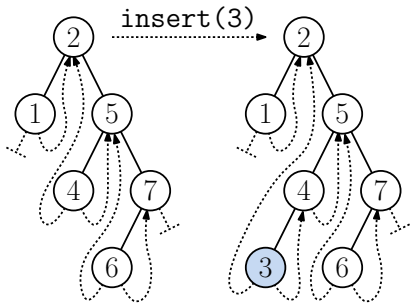


Figure 3: Threaded tree operations.

Problem 4. 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 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. **??**, the right sibling of the node containing "2" is the node containing "8:12". Since the node containing "8:12" is the rightmost node of its parent ("4"), it has no right sibling.)
Your function should run in $O(1)$ time.

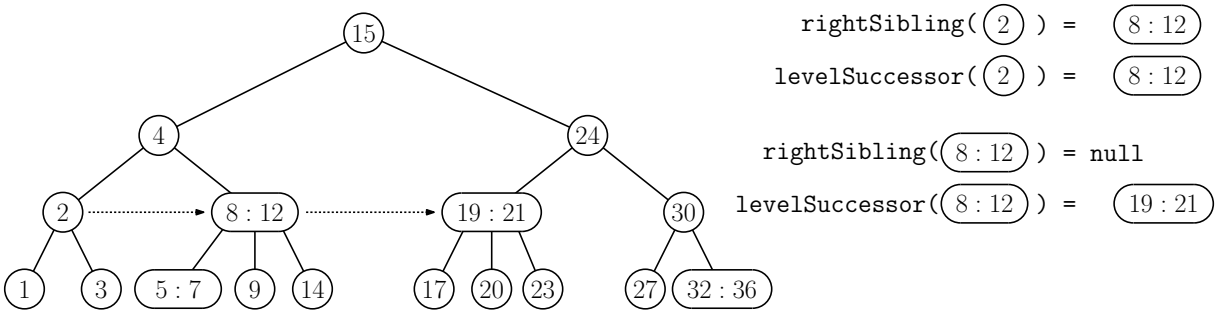


Figure 4: Sibling and level successor in a 2-3 tree.

(b) For a node `p` in a 2-3 tree, its *level successor* is the node to its immediate right at the same level. Give pseudocode for a function `Node23 levelSuccessor(Node23 p)`, which returns a reference to `p`'s level successor, if it exists. If `p` is the rightmost node on its level (including the case where `p` is the root), this function returns `null`. (For example, in Fig. 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 $0 \le i \le m$ (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

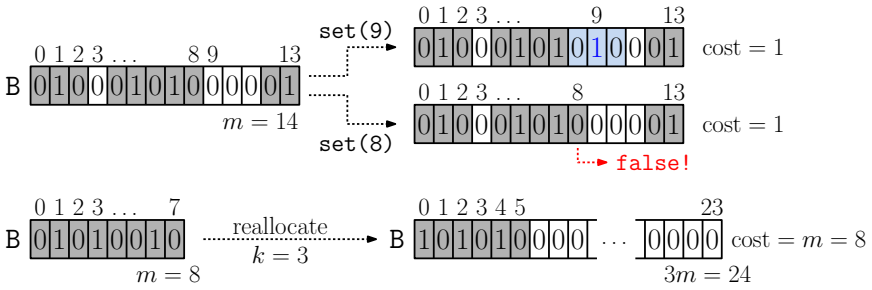


Figure 5: Social-distanced bit vector. (Shaded entries cannot be set to one, due to social-distancing.)

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.

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.)

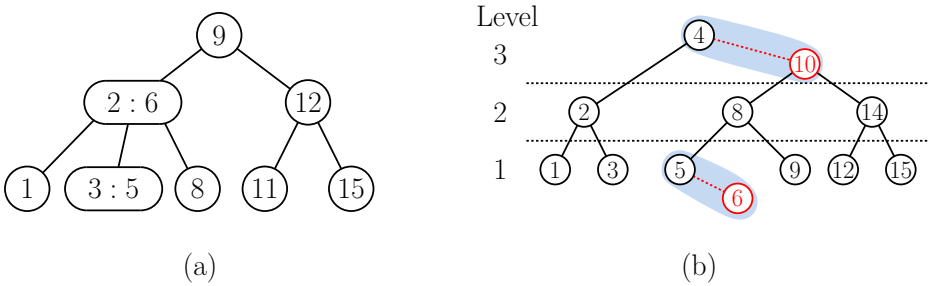
Throughout, if divisions are involved, don't worry about floors and ceilings.

CMSC 420: Spring 2021

Midterm Exam 1

The exam is asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. The total point value is 100 points. Good luck!

Problem 1. (20 points) Consider the trees shown in Fig. 1.



(a) (b)

Figure 1: 2-3 and AA Trees.

(1.1) (5 points) Draw a picture of the AA tree corresponding to the 2-3 tree shown in Fig. 1(a). (Similar to Fig. 1(b), indicate which levels the nodes are at and indicate black-red pairs by connecting them with a dashed line.)

(1.2) (5 points) Draw a picture of the 2-3 tree corresponding to the AA tree shown in Fig. 1(b).

(1.3) (10 points) Show the final AA tree that results by inserting 7 into the AA tree of Fig. 1(b). (Intermediate results may be given to help with partial credit.)

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

(2.1) (5 points) 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.)

(2.2) (5 points) Recall that the *depth* of a node in a tree is the number of edges along the path from the root to this node. In an inorder threaded binary tree, if `u.right` is a thread, what can be inferred about the relative depths of these two nodes? (Select one. No explanation needed.)

(a) `depth(u) < depth(u.right)`

(b) `depth(u) <= depth(u.right)`

(c) `depth(u) > depth(u.right)`

(d) `depth(u) >= depth(u.right)`

(e) We cannot infer anything. It depends on the tree's structure.

(2.3) (5 points) Suppose we have an inorder threaded binary tree, which is *full*. If `u.right` is a thread, which of the following are possible? (Select all that apply. No explanation needed.)

(a) Both `u` and `u.right` are internal.

(b) Both `u` and `u.right` are leaves.

(c) Node `u` is a leaf and `u.right` is internal.

(d) Node `u` is internal and `u.right` is a leaf.

(2.4) (5 points) 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)

(2.5) (5 points) Below we give the AA rebalancing operation `skew`, but we have explicitly commented out the check where `p == nil`). What will the modified function do if we were to call `skew(nil)`? (Select one and briefly explain your answer.)

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

(a) It has no effect and returns a reference to `nil`.

(b) It will alter the contents of `nil`, but it will not modify any of the other nodes of the tree.

(c) It may alter both `nil` and other nodes the tree as well.

(d) It will abort due to an attempt to dereference a `null` pointer.

(2.6) (5 points) Under what circumstances will the priority value of a treap node change? (Select all that apply. No explanation needed.)

(a) Once created, it will never change (until the node is deleted).

(b) It will be changed periodically according to a random process.

(c) It may change if the node is involved in a rotation.

(d) It may change if the node is used as a replacement for a deleted node.

(2.7) (5 points) You are given a skip list storing n items. What is the expected number of nodes that will contribute to level 3 of 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 3. (25 points) 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.

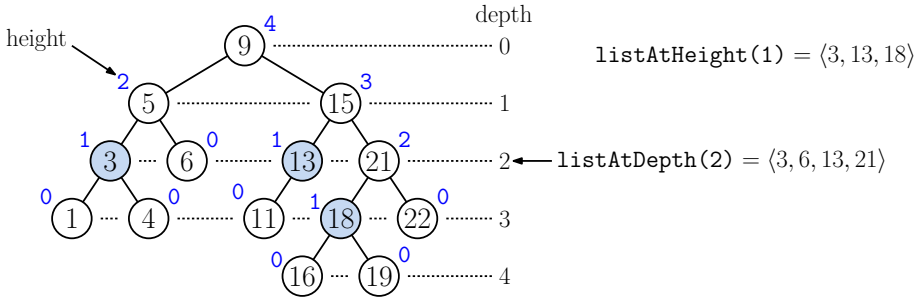


Figure 2: AVL tree heights and depths.

(3.1) (5 points) 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. 2, 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.)

(3.2) (5 points) 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. 2, 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.)

(3.3) (5 points) 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".)

(3.4) (10 points) 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. 2 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. 2 has height 4, and is full at levels 0, 1, and 2.)

Problem 4. (20 points) In this problem, we will consider generalization of the amortized analysis of the dynamic stack algorithm from Lecture 2. Recall that in the dynamic stack from the class, whenever we run out of space using an array of size k, we *expand* by allocating a new array of size $2k$, copy the elements over, and remove the old array. In this problem, we will also consider *contracting*, by halving the size of the array when we have too few elements. Below we give a formal description of our new dynamic stack and the cost of each operation. Throughout, assume that k is a power of 2.

Initialization: We allocate an array of size $k = 1$, which is empty. (Actual Cost = 0)

`push(x)`: If $n < k$ (standard case), we push the element onto the stack and increment n. (Actual Cost = 1) If $n = k$ (overflow case), we double the array size by setting $k \leftarrow 2k$, allocate a new array of this size, copy the contents of the old array into the new array, and remove the old array. We then do the standard-case push. (Actual Cost = $2k + 1$)

`pop()`: If $n = 0$, we return `null`. (Actual Cost = 1) Otherwise, we pop the stack and decrement n. If (after decrementing) $n \leq \frac{k}{4}$, we halve the array size by setting $k \leftarrow \frac{k}{2}$, allocate a new array of this size, copy the contents of the old array into the new array, and remove the old array. (Actual Cost = $1 + \frac{k}{2}$)

Note that in either case, just after reallocation, roughly half of the current array is being used. In this problem, you will derive the amortized cost of operations on this data structure. Consider a long sequence of pushes and pops, starting from any empty stack. We will break this sequence into runs, with each run ending just after each reallocation (expanding or contracting).

(4.1) (5 points) Suppose that when the run starts, the current array size is k, and the run ends with an *expansion* to size $2k$. Prove that there is a constant c (which you may choose) so that the amortized cost is c. (That is, show that the sum of actual costs in the run is at most c times the number of operations.) **Hint:** This is pretty much what we did in class. You can just adapt the proof from the lecture, but put it in your own words.

(4.2) (10 points) Suppose that when the run starts, the current array size is k, and the run ends with a *contraction* to size $\frac{k}{2}$. Prove that there is a constant c (which you may choose) so that the amortized cost is c.

(4.3) (5 points) We triggered a contraction when $n \leq \frac{k}{4}$. Suppose instead that we triggered a contraction (halving the size of the array) whenever $n \leq \frac{k}{2}$. Would the amortized cost still be a constant? Briefly justify your answer.

Because we are most interested in asymptotic bounds, you may focus mainly on the case where n and k are both large numbers.

CMSC 420: Spring 2021

Homework 2: Hashing and Geometric Search

Handed out Thu, Apr 15. Due **Mon, Apr 26, 11pm**. (Solutions will be discussed in class on Tue, Apr 27, so late submissions will not be accepted after the start of class.)

Problem 1. (12 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. (Each is inserted one after the other.) In each case, at a minimum you should indicate the following:

- Was the insertion successful? (The insertion fails if the probe sequence loops infinitely without finding an empty slot.)
- Show contents of the hash table after inserting all three keys.
- For each case, give a count of the number of *probes*, that is, the number of entries in the hash table that were accessed in order to find an empty slot in which to perform the insertion. (The initial access counts as a probe, so this number is at least 1. For example, in Fig. 3 in the Lecture 14 LaTeX lecture notes, `insert("z")` makes 1 probe and `insert("t")` makes 4 probes.)

For the purposes of assigning partial credit, you can illustrate the actual probes that were performed, as we did in Fig. 4 from Lecture 14. But be sure to also list the probe count.

(1.1) (6 points) Show the results of inserting the keys "`X`", "`Y`", and "`Z`" into the hash table shown in Fig. 1(a), assuming that conflicts are resolved using *linear probing*.

(a) Linear probing

```
insert("X")  h("X") = 13
insert("Y")  h("Y") = 15
insert("Z")  h("Z") = 9
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	I	A	K		M		G		L	W	H	C	E	J	

(b) Quadratic probing

```
insert("M")  h("M") = 3
insert("D")  h("D") = 6
insert("Q")  h("Q") = 9
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A			G			L	N		P				R	

Figure 1: Hashing with linear and quadratic probing.

(1.2) (6 points) Show the results of inserting the keys "`M`", "`D`", and "`Q`" into the hash table shown in Fig. 1(b) using *quadratic probing*. (Hint: If you are unsure whether quadratic probing has gone into an infinite loop, it may be useful to note that for any positive integer i, $i^2 \bmod 15 \in \{0, 1, 4, 6, 9, 10\}$.)

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

Problem 2. (8 points) You have a hash table of size m into which you insert n keys using separate chaining (as described in class). The keys are integers from the set $U = \{1, 2, \ldots, nm\}$.

(2.1) (5 points) Prove that, no matter how ingeniously you design your hash function, there *must exist* a subset S of U of size at least n such that every key in S hashes to the same location in the hash table.

(2.2) (3 points) What does (2.1) imply about the *worst-case* running time needed to insert n keys drawn from U into your hash table (again, assuming separate chaining).

Problem 3. (20 points) You are given a 2-dimensional point kd-tree, as described in class, where we assume that the cutting dimension alternates between x and y with each level of the tree. This tree stores a set P of n points in $\mathbb{R}^2$. For each of the following parts, the query object is a square, represented by its center point $q = (q_x, q_y)$ and radius $r > 0$. Let $S(q, r)$ denote a square consisting of the points that lie within a square of side length $2r$ that is centered at q (see Fig. 2(a)). Formally, $S(q, r)$ defined to be the set of points $p \in \mathbb{R}^2$ such that

$$q_x - r \;\leq\; p_x \leq q_x + r \quad \text{and} \quad q_y - r \;\leq\; p_y \leq q_y + r.$$

(Note that if a point is on the boundary of the square, it is considered as lying within the square.) Equivalently, we can define the *max distance* between two points p and q, denoted max-dist(p, q) to be $\max(|p_x - q_x|, |p_y - q_y|)$. The square $S(q, r)$ consist of all p such that max-dist$(p, q) \leq r$.

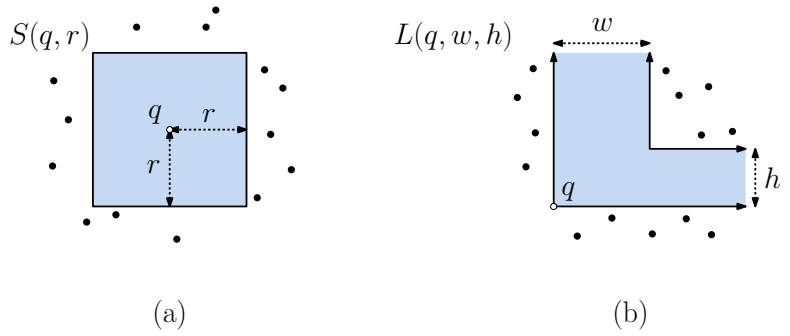


Figure 2: Square and L-shaped emptiness queries in a kd-tree.

(3.1) (6 points) Derive an efficient function

```
boolean emptySquare(Point2D q, float r)
```

which determines whether $P \cap S(q, r) = \emptyset$ (see Fig. 2(a)). That is, it returns `true` if no point of P lies within $S(q, r)$ and false otherwise. By "efficient" we mean that the query time should be $O(\sqrt{n})$, assuming that the tree is balanced, where n is the number of points in the tree. I would suggest using a recursive helper function:

```
boolean emptySquare(Point2D q, float r, KDNode p, Rectangle2D cell)
```

You may assume that any primitive geometric operations involving points, squares, and rectangles can be computed in constant time. For example, if your pseudocode, you can write "if square $S(q, r)$ and rectangle R are disjoint then ..." without explaining how to determine this.

Briefly explain your algorithm and present pseudocode.

(3.2) (4 points) Derive the running time of your function from (3.1), under the assumptions that the tree contains n points, the splitting dimension alternates between x and y, and the tree is balanced.

Hint: It may help to first review the analysis of the orthogonal range-search algorithm from the latex lecture notes. As we did in class, you may make the idealized assumption that the left and right subtrees of each node have equal numbers of points.

(3.3) (6 points) Given a point $q \in \mathbb{R}^2$ and two positive floats w and h, define the *L-shaped region* $L(q, w, h)$ to be the set of points lying within the "L"-shaped region whose lower left corner is q and which extends infinitely upwards and rightwards. The width of the vertical part is w, and the height of the horizontal part is h (see Fig. 2(b)). As above, points lying on the boundary of the region are considered to lie within the region.

Present pseudocode for an efficient function

```
boolean emptyL(Point2D q, float w, float h)
```

which determines whether $P \cap L(q, w, h) = \emptyset$.

Briefly explain your algorithm and present pseudocode. I would suggest the recursive helper function:

```
boolean emptyL(Point2D q, float w, float h, KDNode p, Rectangle2D cell)
```

(3.4) (4 points) Derive the running time of your function from (2.3), under the same assumptions as in (2.2). A complete analysis is not needed. You can explain what modifications are needed to the analysis of (2.2).

Note: You may assume the results proven in class, without the need to prove them again. If you need to modify these results, you need only explain the modifications.

Problem 4. (10 points) In this problem, we will consider how to use/modify range trees to answer two related queries. Throughout, the input set P is a set of n points in $\mathbb{R}^2$ (Fig. 3(a)). 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.

(4.1) (5 points) In a *vertical segment sliding*, the query is given a vertical line segment, specified by its lower endpoint $q = (q_x, q_y)$ and its height h (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 is hit, the query returns `null`.

Describe how to preprocess the point set P in a data structure so that given any query (q, h), segment sliding queries can be answered efficiently. Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

(4.2) (5 points) A *min-V query* is defined by $q = (q_x, q_y)$. Consider the V-shape defined between two rays emanating upwards from q, one with slope $+1$ and one with slope -1 (see Fig. 3(c)). Among all the points of P lying within this V-shape, the answer is the one with the smallest y-coordinate. If no point of P lies within the shape, the query returns `null`. Your data structure should use $O(n \log^2 n)$ storage and answer queries in $O(\log^3 n)$ time.

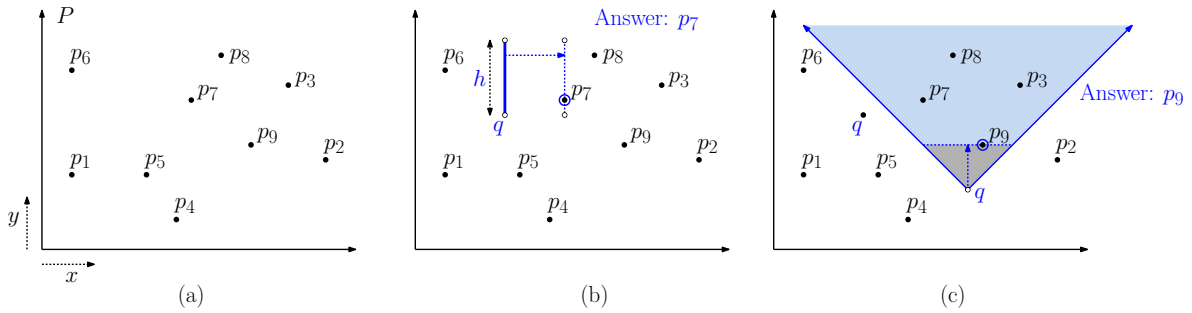


Figure 3: Vertical segment-sliding and min-V queries.

Challenge Problem: You just started work at your new company, "Dilbert's Pretty Good Data Structures". Your team has been tasked to implement a new "automatically initializing" array. The array `table` is specified by its size $m > 0$ and an initialization function f. For simplicity, let's assume that the entries of `table` are integers. The initial value of `table[i]` is defined to be $f(i)$. For example, if $f(i) = i^2$, the array's initial "virtual" contents are as shown in Fig. 4. You may assume that f can be computed in constant time.

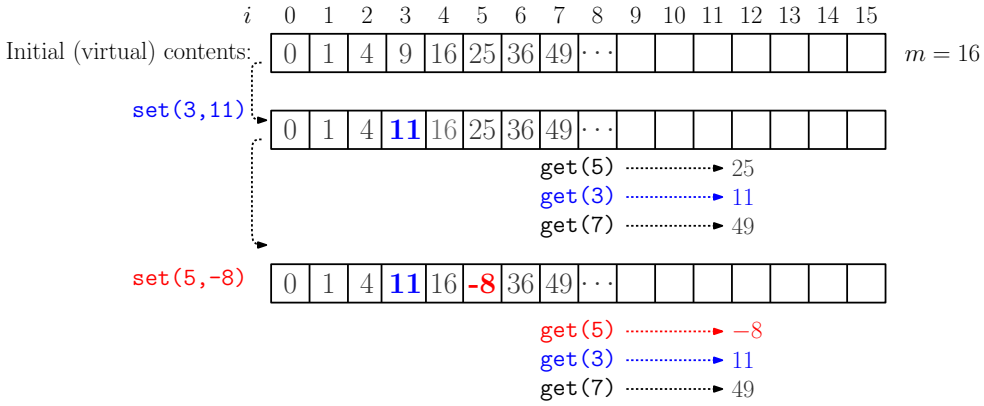


Figure 4: Auto-initializing array for $m = 16$ and $f(i) = i^2$. The initial actual (not virtual) contents are entirely unpredictable.

Your part of the project is to implement two accessor functions `void set(int i, int x)` and `int get(i)`. The first function sets `table[i] = x`. For the second function, if entry `table[i]` has been set by a previous `set` command, then its (latest) set value is returned. If its value has not yet been set, it returns $f(i)$. In both cases, you may assume that $0 \le i < m$.

This would be easy, if you were allowed $O(m)$ time to initialize the array, but you are not. The initial contents of the array are entirely unpredictable (and may have even been set maliciously to trick your algorithm). In spite of this, each function must run in worst-case $O(1)$ time, independent of the value of m or the number n of elements that have already been set.

Your team leader has informed you that you may use additional storage of size up to $O(m)$ in which you may store *any* auxiliary data structure you like (limited to primitive data structures

such as queues and stacks or one of the data structures we have seen this semester). But, as with the table, you cannot assume that this additional storage comes initialized. It too may contain garbage.

Explain how to produce a data structure to solve this problem in the required time and space. Explain your algorithm and derive its running time.

Hint: You will definitely need the additional storage to solve the problem in the stated time bound. For full credit, your answer should be deterministic (not randomized) and the worst-case running time is $O(1)$. For partial credit, show how to perform the operations in amortized $O(1)$ time, or randomized $O(1)$ time (correct with high probability) or alternatively in deterministic $O(\log m)$ time. Note that if you attempt to apply a solution based on hashing, you need to take into consideration the time needed to initialize the hash table.

CMSC 420: Spring 2021

Practice Problems for Midterm 2

This exam will be asynchronous and online. The exam will be made available through Gradescope for a 48-hour period starting at **12:00am the morning of Thu, Mar 29** and running through **11:59pm the evening of Fri, Mar 30** (Eastern Time). The exam is designed to be taken over a 1.5-hour time period, but to allow time for scanning and uploading, you will have **2 hours** to submit the exam through Gradescope once you start it. The exam will be open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. (You may use a simple arithmetic calculator, but I don't expect that you will need one.)

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 a data structure are to be accessed most of the time, but you do not know which these keys are. Among the data structures we have seen this semester, which would be best for this situation? Explain briefly.

(c) 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

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

(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.)

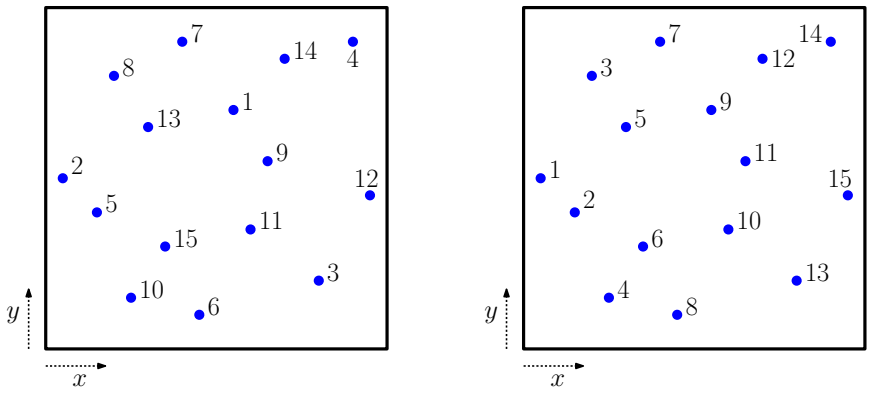


Figure 1: Height of kd-tree.

(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.)

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
}
```

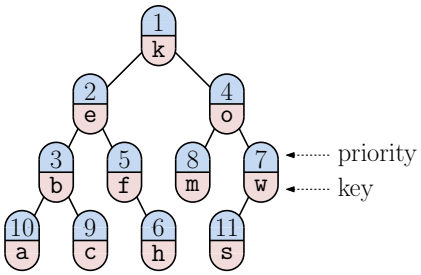


Figure 2: Treap node structure and an example.

(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 $x_0 \le x \le x_1$. If the treap has no keys in this range, the function returns `Integer.MAX_VALUE`. Briefly explain why your function is correct.

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

(b) Assuming that the treap stores n keys and has height $O(\log n)$, what is the 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. In scapegoat trees, we showed that if `size(u.child)/size(u)` $\leq \frac{2}{3}$ for every node of a tree, then the tree's height is at most $\log_{3/2} n$. In this problem, we will generalize this condition to:

$$\frac{\texttt{size(u.child)}}{\texttt{size(u)}} \leq \alpha, \qquad (*)$$

for some constant α.

(a) Why does it **not** make sense to set α larger than 1 or smaller than $\frac{1}{2}$?

(b) If every node of an n-node tree satisfies condition $(*)$ above, what can be said about the height of the tree as a function of n and α? Briefly justify your answer.

Problem 5. 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!)

Problem 6. 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, $x_1 \leq p.x \leq x_2$) and has the maximum y-coordinate (see Fig. 3).

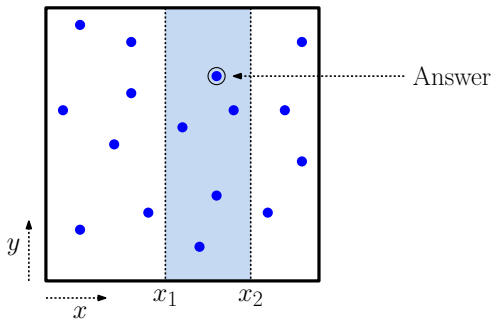


Figure 3: Partial-range max query.

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).

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

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 8. 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. 4). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.

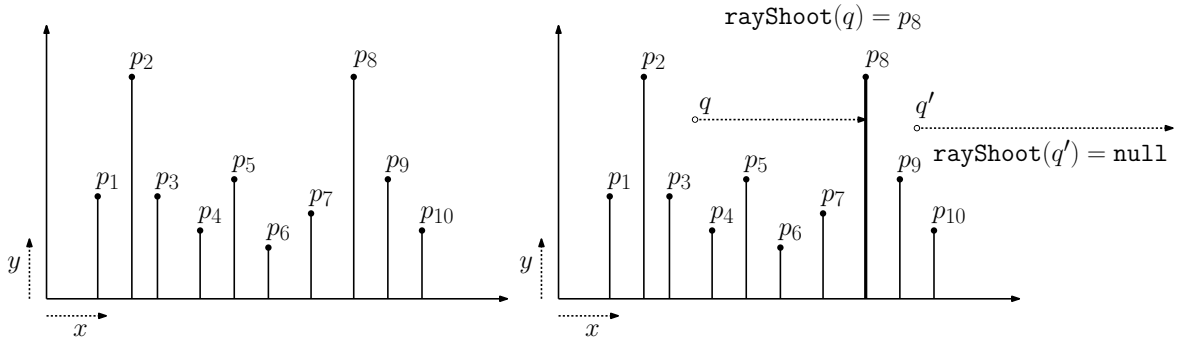


Figure 4: Ray shooting in a kd-tree.

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.

Problem 9. (Expect a problem on range trees. I can't find any good practice problems, but the examples from Homework 2 are pretty good.)

CMSC 420: Spring 2021

Midterm Exam 2

The exam is asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. The total point value is 100 points. Good luck!

Problem 1. (20 points) In this problem we will consider a sequence of operations performed on a hash table. In each case the operations are performed one after the other in sequence. In each case, indicate how many probes are made to the table. (A *probe* is defined to be any access to the table, whether to read or write an entry.) Intermediate results are not required, but may be given to help assigning partial credit.

(a) Double hashing

```
insert("X")  h("X") = 7; g("X") = 2
insert("Y")  h("Y") = 13; g("Y") = 6
insert("Z")  h("Z") = 4; g("Z") = 5
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				M			L		P	K			B	I

(b) Linear probing/Deletion

```
insert("R")  h("R") = 2
delete("B")  h("B") = 13
insert("D")  h("D") = 1
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E	A	F	M	B	J	L			P		K		I	W

Figure 1: Hasing with open addressing

(1.1) (10 points) Show the results of performing the insertion operations given in Fig. 1(a) assuming *double hashing*. (The second hash function g is shown in the figure.)

(1.2) (10 points) Show the result of performing the sequence of insert and delete operations shown in Fig. 1(b) assuming *linear probing*. The shaded entries are "empty".

Problem 2. (25 points) Short answer questions. Unless requested, explanations are not required, but may always be given to help with partial credit.

(2.1) (3 points) In the skip list data structure we assumed that the coin we tossed was even, coming up heads or tails with equal probability. If we were instead to use a biased coin that comes up heads more often than tails (but both still with constant probability), what would the effect be on the *expected storage* needed by the data structure? Select one and explain briefly:

(a) The expected storage requirements would go up (by at least a constant factor).

(b) The expected storage requirements would go down (by at least a constant factor).

(c) The expected storage requirements would remain unchanged.

(2.2) (3 points) Same as (2.1), but what would the effect be on the *expected query time*? Select one and explain briefly:

(a) The expected query time would go up (by at least a constant factor).

(b) The expected query time would go down (by at least a constant factor).

(c) The expected query time would remain unchanged.

(2.3) (3 points) What properties of B-trees make them particularly attractive for use in external (disk) memory?

(2.4) (4 points) 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?

(2.5) (4 points) A treap and an standard (unbalanced) binary search tree both have the property that they support dictionary operations in $O(\log n)$ expected time. How is the notion of "expected" different in each case, and which structure would be preferred? Explain briefly.

(2.6) (4 points) Give one example of an operation that can be performed more efficiently (on average) with hashing compared to AVL trees. Explain briefly.

(2.7) (4 points) Give one example of an operation that can be performed more efficiently (on average) on AVL trees compared to hashing. Explain briefly.

Problem 3. (20 points) This problem involves the question of how the depths of nodes change when splay operations are performed. Consider the splay tree shown in Fig. 2(a) with each node's depth indicated in blue. After performing `splay(a)` observe that some node depths have increased, some have decreased, and some of remain unchanged. The greatest increase is node g, whose depth increases by $+2$.

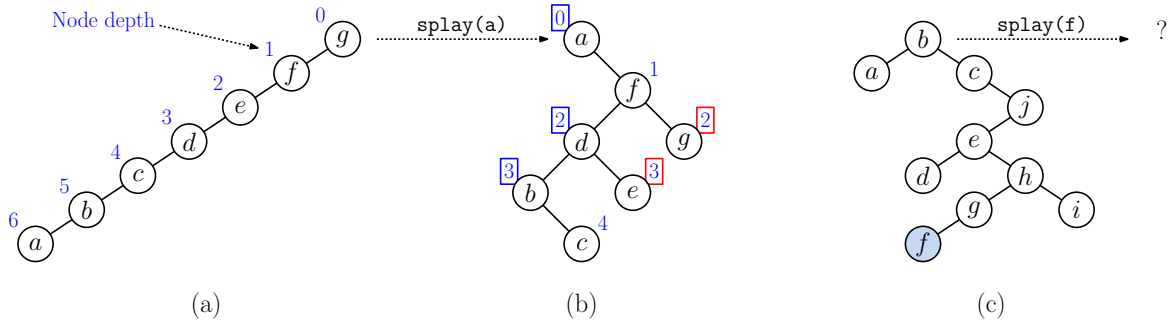


Figure 2: How depths change in a splay tree.

(3.1) (8 points) Consider the tree shown in Fig. 2(c). Show the result of performing the operation `splay(f)` on this tree. (We only need the final tree, but intermediate results can be shown to help with partial credit.)

(3.2) (2 points) Indicate in your drawing which nodes have increased in depth, and in each case by how much the depth has increased.

(3.3) (10 points) Select one of the two options below, and justify your choice.

- Given any splay tree T and any key x in this tree, after performing `splay(x)` if a node's depth increases, *this increase is at most 2.*
- There exists a splay tree T and a key x such that, after performing `splay(x)` there is a node in this tree such that *its depth increases by 3 or more.*

If you believe that first option is true, provide a short sketch of a proof. If you believe that the second option is true, provide an example of a tree T and x where this happens.

Problem 4. (20 points) Throughout this problem, assume that you are given a standard (not wrapped) kd-tree storing a set P of n points in $\mathbb{R}^2$ (see Fig. 3(a)). Assume that the cutting dimension alternates between x and y. You may also assume that the tree stores a root cell `rootCell`, 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.

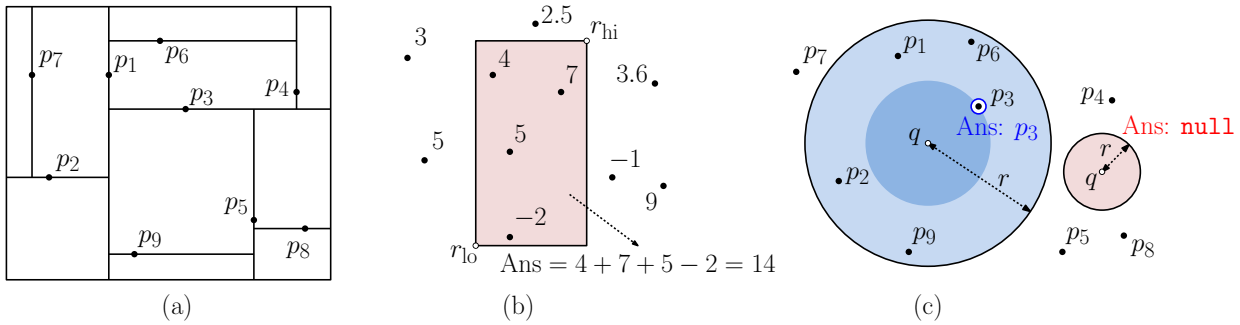


Figure 3: Queries on kd-trees.

(4.1) (7 points) In a standard range-counting query, we want to count the number of points in the range. Suppose that each point $p_i \in P$ has an associated real-valued weight w_i. In a *weighted orthogonal range query*, we are given a query rectangle R, given by its lower-left corner r_{lo} and upper-right corner r_{hi}, and the answer is the sum of the weights of the points that lie within R (see Fig. 3(b)). If there are no points in the range, the answer is 0.

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

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

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

where `p` is the current node in the kd-tree, `cell` is the associated cell. The initial call is `weightedRange(R, root, rootCell)`.

(4.2) (3 points) Briefly analyze the running time of your algorithm, assuming that the tree is balanced. (You may apply/modify results proved in class.)

(4.3) (10 points) 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, float 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. The initial call is `frnn(q, r, root, rootCell, null)`.

To receive full credit, your algorithm should not recurse on the children of any node p that (based on p's cell) cannot contribute a (better) answer to the query.

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

Problem 5. (15 points) 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 pseudo-code) is sufficient. Justify your algorithm's correctness and derive its running time.

(5.1) (5 points) Assume you are given an n-element point set P in $\mathbb{R}^2$ (see Fig. 4(a)). In an *empty square annulus 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 `true` if $A(q, r_1, r_2)$ contains no points of P, and `false` otherwise (see Fig. 4(b)).

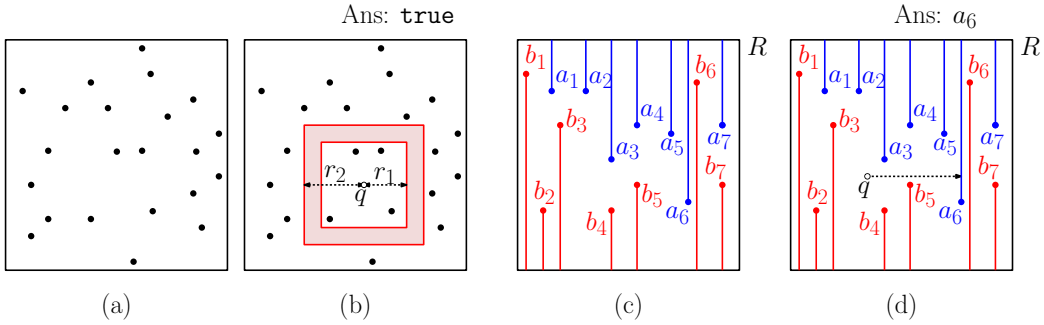


Figure 4: Queries on range trees.

Describe how to preprocess the point set P in a data structure that can efficiently answer any empty annulus query (q, r_1, r_2). Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time. (I don't care how you handle edge cases, such as points lying on the boundary of the annulus as being inside or outside.)

(5.2) (10 points) This problem takes place inside a large axis-aligned rectangle R. You are given two sets of points $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, all lying within R. At each point a_i there is a vertical line segment from a_i to the top edge of R, and at each point b_j there is a vertical line segment from b_j down to the bottom edge of R (see Fig. 4(d)). A *horizontal ray-shooting query* is presented by a point $q \in R$. The answer

to the query is the first line segment (either from the the A set or B set) that is hit by a horizontal ray shot to the right from q (see Fig. 4(d)).

Describe how to preprocess the point sets A and B into a data structure that can efficiently answer any horizontal ray-shooting query. Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time. (I don't care how you handle edge cases, such as if the ray passes through a point from A or B.)

CMSC 420: Spring 2021

Final Exam

The exam is asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. The total point value is 120 points. Good luck!

Problem 1. (35 points) Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(1.1) (2 pts) What was play on words with the data structure named "deque"?

(1.2) (6 pts) Recall that a *free tree* is an unrooted tree. A *leaf* in a free tree is a node that has exactly one incident edge. Consider a free tree with $n \geq 200$ nodes. As a function of n, what is the *minimum* number of leaves the tree can have? What is the *maximum* number of leaves the tree can have? (Draw two pictures to illustrate your answers.)

(1.3) (3 pts) You just inserted a key into an AVL tree, where the newly inserted key is larger than any other key in the tree. **True or false**: If a rotation was needed to balance the tree after the insertion, it was a *single rotation* (not a double rotation). Briefly explain.

(1.4) (3 pts) It is a fact that any 2-3 tree storing a set of n keys has height less than or equal to the height of an AVL tree storing the same set of keys. **True or false:** Given a 2-3 tree and AVL tree, both storing the same set of keys, the number of comparisons needed to perform `find(x)` in the 2-3 tree is less than or equal to the number of comparisons needed to perform the same operation in the AVL tree. Briefly explain.

(1.5) (3 pts) The splay operation promotes a key to the root of the splay tree through rotations. While we could have achieved this through simple Zig rotations, we instead preferred Zig-Zag and Zig-Zig rotations. Why does using Zig rotations alone fail to produce a data structure that guarantees efficient amortized performance? (You may want to explain with the help of a drawing.)

(1.6) (6 pts) Given a B-tree of order $m = 15$, what is the minimum and maximum number of children that any internal node might have, assuming this node is not the root of the tree. What if the node is the root?

(1.7) (6 pts) We showed in class that if we alternate splitting between x and y in a height-balanced kd-tree storing n points in $\mathbb{R}^2$, the number of nodes whose associated cell is stabbed by any axis parallel line is $O(\sqrt{n})$.

- Show (by drawing a picture) that this may fail if the tree is *not height balanced*.
- Show (by drawing a picture) that this may fail if the splitting directions *do not alternate*.

(1.8) (3 pts) The fastest known algorithm for building a binary search tree from a set of n keys, runs in time $O(n \log n)$. But, with scapegoat trees, we claimed that we could rebuild a subtree containing m keys in time $O(m)$. How is this possible?

(1.9) (3 pts) The *Cartesian tree* by Jean Vuillemin and the *priority search tree* of Edward McCreight are both closely related to what other data structure that we have studied this semester?

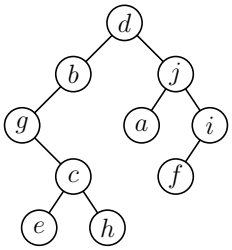


Figure 1: Tree traversals.

Problem 2. (15 points) This problem involves the binary tree shown in Fig. 1.

(2.1) (3 points) List the nodes according to a *preorder traversal*

(2.2) (3 points) List the nodes according to an *inorder traversal*

(2.3) (4 points) List the nodes according to the order given in the pseudocode block below. The initial call is `crazyTraverse(root)`, where `root` is the root of the tree.

```
void crazyTraverse(Node p) { // a crazy way to traverse a tree
    if (p != null) {
        print(p.key)
        crazyTraverse(p.right)
        if (p.left != null) {
            print(p.key)
            crazyTraverse(p.left)
        }
    }
}
```

(2.4) (5 points) Present pseudocode for a tree traversal procedure that produces the output shown below[1] when run on the tree of Fig. 1. Notice that each key is printed twice.

```
d j i f f i a a j b g c h h e e c g b d
```

Problem 3. (15 points) In this problem we will build a suffix tree for the string $S =$ `baabaababababaa$`.

(3.1) (5 points) 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.

(3.2) (2 points) List the substring identifiers again together with their indices (0 through 13), but this time in alphabetical order (where `"a"` $<$ `"b"` $<$ `"$"`).

(3.3) (8 points) 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.

[1] An early version of the exam contained an extra character "g". If you had this version of the exam and lost points for this reason, please file a regrade request.

Problem 4. (25 points) 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
}
```

(4.1) (10 points) 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. 2.)

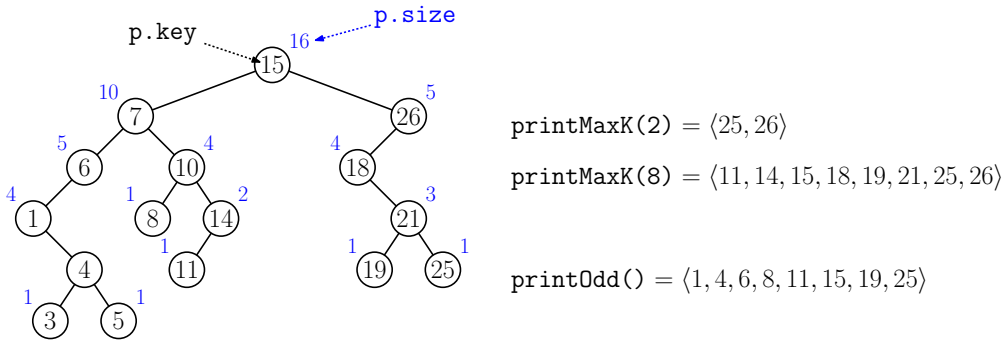


Figure 2: The functions printMaxK and printOdd.

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. (The running time will be derived in (4.2).)

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`.

(4.2) (5 points) Derive the running time of your algorithm in (4.1).

(4.3) (10 points) Present 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. 2.)

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 5. (15 points) 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.

(5.1) (2 points) 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.

(5.2) (8 points) 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.

(5.3) (5 points) 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 6. (15 points) 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.

(6.1) (10 points) 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.)

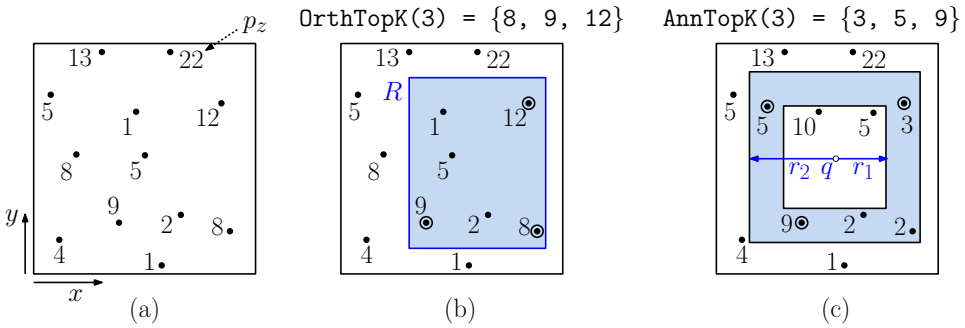


Figure 3: Orthogonal top-k queries and annulus top-k queries.

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.

(6.2) (5 points) 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)).

In Problem 4, we saw how to print the k-largest entries from a balanced binary tree. You may use that result here if you like.