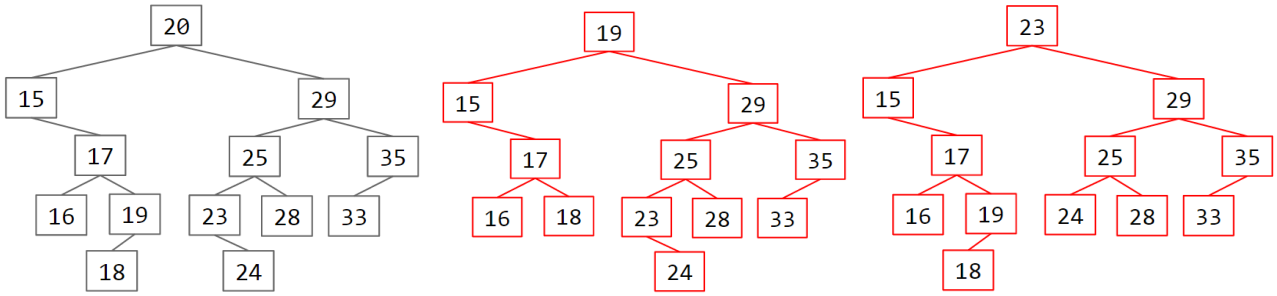
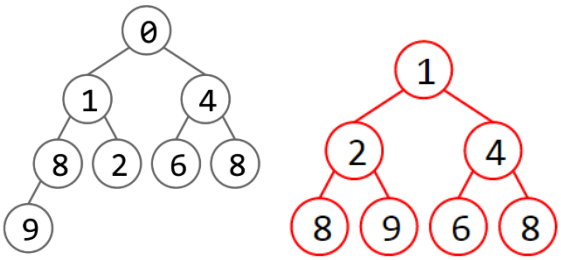
1. **Basic Operations (6 Points).**

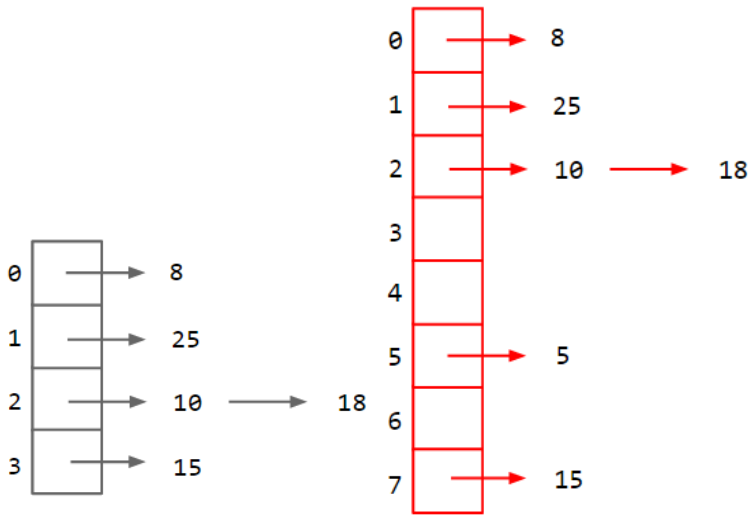
a. **To the right of the BST below**, draw a BST that results if we delete 20 from the BST. You should use the deletion procedure discussed in class (i.e. no more than 4 references should change). Either of the two trees in red below are correct. In the left case, we've chosen the predecessor of 20, i.e. 19, as the new root. In the right, we've chosen the successor.



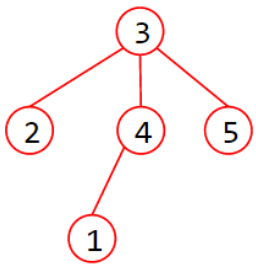
b. **To the right of the minHeap below**, draw the minHeap that results if we delete the smallest item from the minHeap.The only correct answer if we use the procedure from class is the minHeap shown to the right.



c. **To the right of the External Chaining Hash Set below**, draw the External Chaining Hash Set that results if we insert 5. As part of this insertion, you should also resize from 4 buckets to 8 (in other words, the implementer of this data structure seems to be resizing when the load factor reaches 1.5). Assume that we're using the default `hashCode` for integers, which simply returns the integer itself.The correct answer is shown to the right. All items must appear in exactly the bucket shown (though 10 and 18 could be in reverse order).



d. Draw a valid Weighted Quick Union object that results after the following calls to connect: `connect(1, 4)`, `connect(2, 3)`, `connect(1, 3)`, `connect(5, 1)`. Don't worry about the order of the arguments to each connect call, we'll accept any reasonable convention. There are many correct answers depending on your convention for order of arguments. The most important points are that when you connect items, that you're only connecting the roots of each subtree to other roots, and that when you connect 5 to the weight 4 tree, that the 1 points directly at the root of that tree. One example solution is given below. Giving an answer in terms of an id[] array was also acceptable.



2. **Asymptotics (5 Points)**

a. Suppose we run experiments to understand the runtime performance of the `add` method of the `PotatoSack` class. The runtime as a function of N (the number of inserts) is shown below. Using the technique from the asymptotics lab, ***approximate*** the empirical run time in ***tilde notation*** as a function of N. As a reminder, in that lab, we assumed that the runtime is aN^b, and found a and b. Do not leave your answer in terms of logarithms. Your a and b must be within 25% of our answers. Use only the data points that you expect to give the best approximation of the asymptotic behavior of the algorithm. Hint: To double check your answer, plug in N=1000 and see if the runtime prediction seems sensible.

N	Time (s)
1	0.00
2	0.01
3	0.01
6	0.03
13	0.16
25	0.63
50	2.50
100	9.97

Answer: $0.001N^2$

Since we are interested in asymptotic behavior, we use the largest data points. To calculate b: $\log_2(9.97/2.5)$ is approximately 2. Then we have that $9.97 = a100^2$, so $a = 0.001$. We can double check by plugging in N=1000, getting 1000 seconds, which 100x the time as we'd expect from 10xing the input to a quadratic algorithm.

b. Suppose we measure the performance of a collection X, and find that inserting N items takes $\Theta(N^2)$ time. For each of the following**, circle the collection type if it is possible** for that collection to take $\Theta(N^2)$ time to insert N items on a worst-case input, and **cross out the collection type if it is impossible**. Assume that each is correctly implemented. Either circle or cross out every answer.

LinkedList | ~~2-3 Tree Set~~ | ~~HeapMinPQ~~ | ~~LLRBST Set~~ | Your BSTMap (from HW6) | External Chaining Hash Map | ArrayList

NOTE: You will lose points if an item is neither circled nor crossed out.

c. If we have two correct algorithms for solving the same problem that use the exact same amount of memory, but have worst-case runtimes that are $\Theta(N)$ and $\Theta(N^2)$, is it always better to use the algorithm that is $\Theta(N)$? If so, why? If not, why not?

Best Answers:

- For small N, $\Theta(N^2)$ could be faster than $\Theta(N)$ if there is a smaller coefficient for $\Theta(N^2)$. This comes up, for example, in matrix multiplication algorithms (more during the last week of class).
- The worst-case may only occur for specific inputs that we don't care about. This is, for example, what we observe with Quicksort, which is worst case , $\Theta(N^2)$, but often used in lieu of Mergesort which is worst case$\Theta(\mathrm{N}\log N)$.

3. **Exceptions (2 Points).**

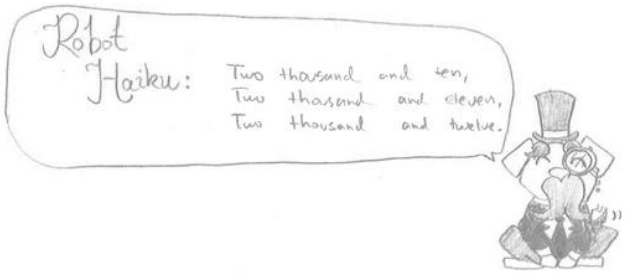
One common software engineering strategy is to create log files that can be manually examined if something goes wrong. In the code below, the writeToLog method writes the given argument to some log. What are the contents of the log file after the code below is executed? You may not need all of the lines provided.

```
_10_____________________________
_No tenth item available!_______
_____________________________
_____________________________
_____________________________
```

```
public class QuestionThree {
    public static void printTenth(int[] a) {
        try {
            writeToLog(a[10]);
        } catch(IndexOutOfBoundsException e) {
            writeToLog("No tenth item available!");
            throw(e);
        }
    }

    public static void main(String[] args) {
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
        printTenth(new int[]{0, 1, 2, 3});
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
    }
}
```

This area is a designated fun zone. Perhaps you would like to compose a poem in your native language about some topic of interest? Or perhaps you'd like to draw a dog with an overly ornate moustache?



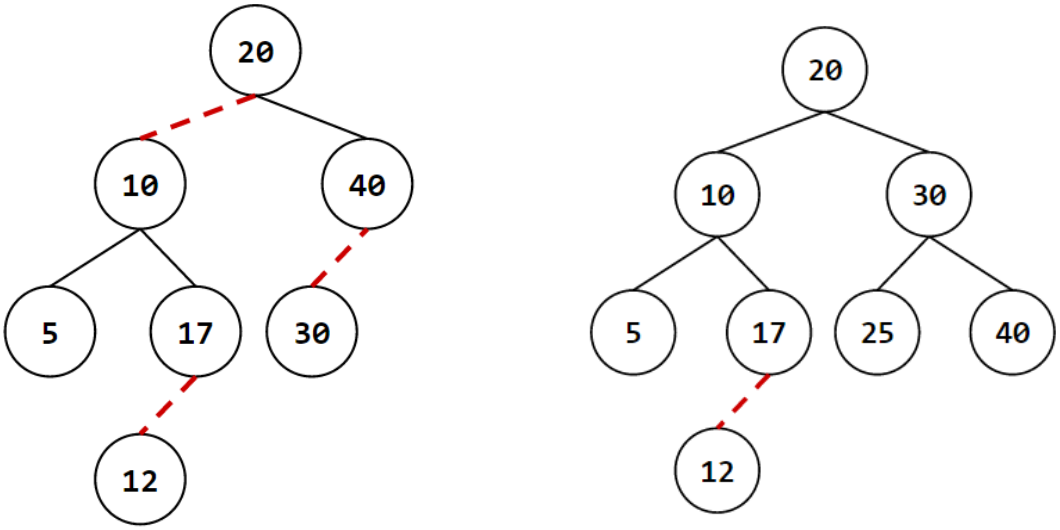
4. **TreeTime (5 Points).**

a. True or false: If A and B are 2-3-4 trees with the same exact elements, they must be identical. If true, justify with a short **(less than 20 word)** explanation. If false, provide a counter-example.

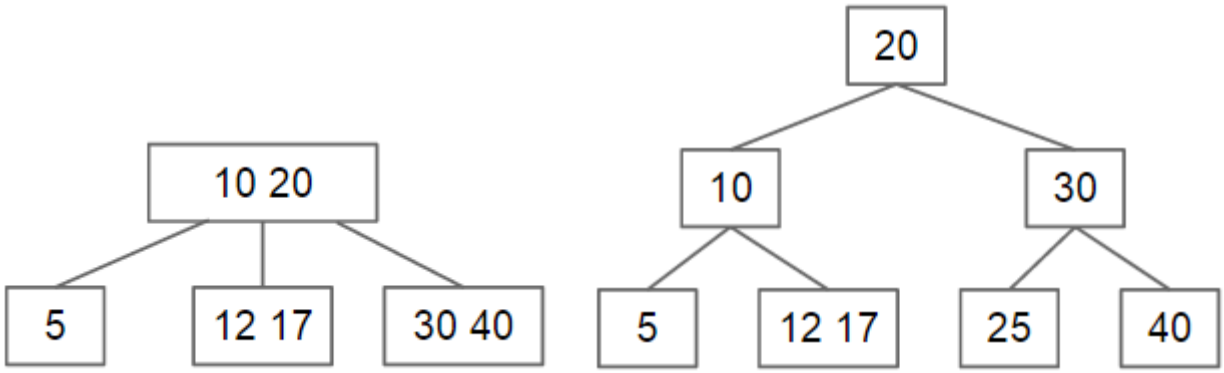
Insert 1,2,3,4,5 (pushing up the 2 upon inserting the 4).
Insert 2,3,4,5,1 (pushing up the 3 upon inserting the 5).

These 2 sequences of inserts produce different trees.

b. Draw the red-black tree which results after calling `insert(25)` on the red-black tree shown below. We denote red links with dashed lines and black links with normal lines. Please use the same notation in your answer.**You should draw your tree in the empty space to the right of the given tree. Do not modify the given figure.**Hint: Every 2-3 tree corresponds to exactly one LLRB, and every LLRB corresponds to exactly one 2-3 tree.Answer to the right.



To achieve this answer, we can start by converting the LLRB into a 2-3 tree (show below, to the left), then inserting 15 (result below, to the right). Then converting back into an LLRB.



c. Suppose that we want to write a method `sumDescendants`, which replaces the value of each node in a tree with the sum of all of its descendants' values (not including itself), and then returns the sum of its original value (before being changed) plus all of its descendants' values.

For example, given the tree on the left, `sumDescendants` on node 6 would return 42 and change the tree to look like the one on the right (since 36 + 6 = 42).

Fill in the `sumDescendants` method. You may not need all lines. Do not use more lines.

```
public class TreeNode {
    public TreeNode left, right;
    public int value;

    public TreeNode(int n) {
        value = n;
    }

    /* Replaces value with sum of all of its descendants' values. */
    public int sumDescendants() {
        if (left == null && right == null) {
            int oldVal = value;
            value = 0;
            return oldVal;
        }
        else {
       int leftSum = 0;int rightSum = 0;
       if (left != null) {
           leftSum = left.sumDescendants();
       }
       if (right != null) {
           rightSum = right.sumDescendants();
       }
       int oldVal = value;
       value = leftSum + rightSum;
            return oldVal + value;
        }
    }
}
```

Left null check(s) & right null check(s)

- Can never call a method on or check an instance variable of **left** or **right** without first doing a null check
- Null check must check **left** and call **left.sumDescendents()**, check **right** and call **right.sumDescendents()**

Call to **left.sumDescendents()** & call to **right.sumDescendents()**

- Things that are okay:
  - Missing parentheses
  - Spelling the method name incorrectly
- Things that are NOT okay:
  - Passing in left or right as a parameter e.g. **sumDescendents(left)**
  - Not calling **sumDescendents()** on an object

Correct base case value

- **value** = 0
- Things that are okay:
  - **val** = 0

Correct non-base case value (sum of **left** and **right** descendents)

- Must sum recursive calls of **sumDescendents** for **left** and **right** children
- Things that are NOT okay:
  - Declaring variables inside if/else statements and referring to them outside
  - Returning early without changing **oldValue**
  - Changing **value** before changing **oldValue**
  - Modifying **value** incorrectly, or modifying **left.value** or **right.value** incorrectly
  - Forgetting some cases (only **left** is null, only **right** is null, neither are null)

5. Code Analysis (2.5 points).

For each of the pieces of code below, give the runtime in $\Theta(\cdot)$ notation as a function of the given parameters. Your answer should be simple, with no unnecessary leading constants or unnecessary summations.

$\Theta(n)$

```
public static void f1(int n) {
        for (int i = 0; i < 2*n; i += 1) {
            System.out.println("hello");
        }
    }
```

$\Theta(n \log n)$

```
public static void f2(int n) {
        if (n == 0) { return; }
        f2(n/2);
        f1(n);
        f2(n/2);
    }
```

Note: The problem above is the same pattern as mergesort.

$\Theta(n \log n)$

```
public static void f3(int n) {
        if (n == 0) { return; }
        f3(n/3);
        f1(n);
        f3(n/3);
        f1(n);
        f3(n/3);
    }
```

Note: The problem above is the same pattern as mergesort, but now with 3 subproblems,all of size N/3.

$\Theta(2^n)$

```
public static void f4(int n) {
        if (n == 0) { return; }
        f4(n-1);
        f1(17);
        f4(n-1);
    }
```

$\Theta(n^m)$

```
public static void f5(int n, int m) {
        if (m <= 0) {
            return;
        } else {
            for (int i = 0; i < n; i += 1) {
                f5(n, m-1);
            }
        }
    }
```

6. **The Right Tool for the Job (6 points).**

For each of the five tasks below, pick <u>one or two</u> data structures and describe very briefly (in 20 words or less) how you'd use those data structures to solve the problem. You should select from the following Java Collections: TreeMap, TreeSet, HashMap, HashSet, LinkedList, ArrayList, HeapMinPQ, HeapMaxPQ, WeightedQuickUnion. TreeMap and TreeSet utilize red-black trees. HashMap and HashSet utilize external chaining.

You should pick the data structure (or structures) that are best suited to the task in terms of performance and ease-of-use, taking into account the specific types of inputs listed in each problem. For most problems, you should need only one data structure. If some part of the problem seems ambiguous, state the assumptions that you're making.

For each task, also give the runtime in $O(\cdot)$ notation. Give the tightest bound you can (e.g. don't just write $O(n^{n^{nn}})$, which while technically correct, isn't very informative).

Task 1) Read in a text file and print all of the unique words that appear. The words should be printed in alphabetical order. Give the O(·) runtime in terms of N, the number of words in the file. Remember, you must provide either a choice of either <u>one or two</u> data structures, as well as a short (< 20 word) description of how you'd use those data structures to solve the problem.

The simplest and most natural choice is a TreeSet. Inserting all items takes O(N log N) time. Iteration (which performs an in-order traversal) takes O(N) time. TreeMap also works, but it is awkward to use a Map instead of a Set since you're not really using the key-value mapping feature (and instead are just using containsKey). However, performance and code complexity for a TreeMap would be identical, despite the aesthetic grossness.

Alternately, one could use a HashSet, with insertion taking O(N) time (assuming we don't have pathologically colliding Strings). However, we'd then need to somehow get the items in the HashSet in sorted order. The simplest way would be to stick them in an array and sort it, in O(N log N) time, but other possibilities work as well. This solution is inferior to TreeSet in both performance and complexity for the programmer. It was not correct to create a HashMap that mapped from hashCode to String – what you were thinking of was a HashSet (a map from hashCode to String doesn't detect duplicates).

Task 2) Given two Collections of very long Strings (e.g. DNA Sequences of tens of thousands of characters or more), `observed` and `known`, check each String in `observed` to see if it exactly matches any String in `known`. `observed` is an `ArrayList<String>` of size N_O. For this task, choose a data structure for `known`. Give the O(·) runtime in terms of N_O and N_K, where N_K is the size of `known`. Assume that the Strings in `known` are highly dissimilar from each other. Assume also that known has already been constructed.

For this particular case, a TreeSet will vastly outperform a HashSet on any real dataset, due to the fact that we can avoid computing the hash value of any particular string. Runtime will be $O(N_O \log N_K)$.

A HashSet will have runtime $O(N_O)$, which while asymptotically faster will be inferior in practice since $\log N_K$ is much smaller than the length of a String. This is a very subtle but interesting technical point

that was allocated little credit, but which it is nonetheless good to be aware of. I suspect very few students made this realization.

More verbosely, we can compare TreeSet and HashSet performance would be to consider the runtimes as a function that also include the string length M. For a HashSet, the runtime will be $O(N_O M)$, due to the cost of computing a String .hashCode. By contrast, a TreeSet will only compare the first few characters (since each string is highly dissimilar from every other string) at each level of the tree. For simplicity, let's call represent this number of characters by a function $c(M)$. This means that the runtime for a TreeSet will be $O(N_O \log N_K c(M))$. Given that M is in the tens of thousands, we can reasonably assert that $M > \log N_k c(M)$ for any real dataset.

One workaround for this would be to create a custom hash function which only uses only some characters of a String.

Task 3) Read in a large number of grayscale images, and display all of the unique images. The images may be displayed in any order. Each image is stored as a Picture object that contains image data stored as an `int[][]`variable, all of which are 512 x 512 (i.e. have a width and height of 512). Give the O(·) runtime in terms of N, the number of images.

The most natural approach is a HashSet. In this case, the runtime would be simply O(N), assuming that our hashCode evenly distributes all items. A TreeSet is also a reasonable choice, and may actually be faster for the same reasons in task 2; however, in this case, the hashCode was not specified, so we cannot say for certain which data structure will do better. If we use a TreeSet, the runtime is $O(N \log N)$, and the Picture class would need to implement the Comparable interface (which is a bit awkward for a Picture class).

Note: The original version of the midterm simply stated that the picture was stored as an int[][]. The intention was that this int[][] was wrapped in some sort of object, otherwise usage of a TreeSet or HashSet would fail. TreeSet would fail since int[][] are not Comparable, and HashSet would fail because Java would use the default equals method for arrays, which only compares using == instead of actually comparing elements. An efficient solution to this interpretation of the problem would require creation of a container class like Picture to allow usage of a TreeSet or HashSet.

Multiple different runtimes were accepted because of different possible interpretations of the runtime (runtime of just a single image or runtime of all of the N images?)

Common answers:

- HashMap or TreeMap. It's unnecessary to use a mapping because you can store the image directly. There's no need to count instances, hence a Set structure is better. A mapping works if you properly store the image as the key. Using the .hashCode value as the key only works if the hash code is a perfect hash, or else non-duplicates might be considered duplicate.
- ArrayList or LinkedList. A less efficient solution is to use a list. For every image, check to see if it's equal to all previously added lists, and if it's unique, then add it to the list. You must have specified the check for uniqueness in order to receive any points at all (or else you aren't satisfying the job).

Task 4) Store the email address for each username on our website, which is devoted to publishing articles about computer hackers being terrible people. Usernames and email addresses are both Strings (and thus use the default `.hashCode` and `.compareTo` method). Our website allows anybody to register any number of accounts. Give the $O(\cdot)$ runtime needed to add each user in terms of N, the current number of users.

A quick glance indicates that we'd want to use either a TreeMap or a HashMap. We did not specify whether you'd want to map from username to email address or vice versa, but this detail is irrelevant.

Here, we're forcing you to use the (well known) .hashCode for Strings. This means that a community of hackers trying to ruin the performance of our website could create a bunch of usernames that all hash to the same hashCode. This would eventually cripple our site. To safely use a hash table, we would have to either use a fancier hashing strategy (i.e. not HashMap which is just external chaining), some sort of secure hash function.

An even better answer would be a TreeSet, which is not vulnerable to such attacks, though the performance in the absence of such attacks would be somewhat slower.

For a HashMap, the runtime would be $O(1)$ assuming no collisions, $O(N)$ assuming many collisions, and for a TreeSet, it would be $O(\log N)$.

Task 5)*Erweitetern Netzwerk*is a new German minimalist social network that provides three functions. Its user base is capped at a maximum of K users, where K is some large constant known at runtime:

- Neu: Enter a username and click the Neu button. Create a new user with this username if space is still available.
- Befreunden: Enter two usernames and click the befreunden button. The users are now friends. If one of the users does not exist, ignore the command.
- Erweiterten Netzwerk: Enter two usernames and click the Erweitertern Netzwerk button. The website prints true if there is a chain of user friendships that connect the users.

Your Neu, Befreunden, and Erweiterten Netzwerk commands must run in O(log N) time in the worst case. Give the $O(\cdot)$ runtime in terms of N, the number of users at the time the command was executed. Assume that K > N.

To track connectedness, we use a WeightedQuickUnion. Since the connect operation of WeightedQuickUnion requires integer arguments, we also need a map from username to id (for example a HashMap, though a TreeMap would be fine as well if we're worried about collisions). In this case, we have the following runtimes:

- Neu: O(1) for adding to a HashMap.
- Befruenden: O(1) to do lookups in a HashMap. O(log N) to do a connect call. Overall then O(log N).
- Erweiterten Netzwerk: O(1) to do lookups, O(log N) to do isConnected call. Overall then O(log N).

7. GorpyCorp (2 points).

Gorpy McGorpGorp is the founder of GorpyCorp. GorpyCorp is organized into several independent teams known as ”Circles”, where every Circle has a leader known as a “lead link”. Gorpy uses the following data structure to record the `members` and `teamName` of each Circle:

```
public class Circle {
      HashSet<Member> members;
      String teamName;

      public int hashCode() {
            int hashCode = 0;
            for (Member m : members) {
                  hashCode = hashCode * 31 + m.hashCode();
            }
            hashCode = hashCode + teamName.hashCode();
            return hashCode;
      }

      public int compareTo(Circle other) {
            if (this.members.size() == other.members.size())
                  return this.teamName.compareTo(other.teamName);
            return this.members.size() – other.members.size();
      }

      public void addMember(Member newMember) {
            members.add(newMember);
      }

      ...
}
```

Rather than storing the leader of each Circle inside of the Circle object, Gorpy instead decides to create a separate HashMap defined below, which allows a programmer to look up the lead link of each circle.

```
HashMap<Circle, Member>leadLinks;
```

What is the most significant problem with Gorpy’s usage of a HashMap? If he uses a TreeMap instead of a HashMap, will this problem be fixed? Why or why not?

The problem is that Circle objects are mutable. If we insert them into a HashMap, and then later change the Circle, then we will no longer be able to find the Circle when we try to look it up in the HashMap. For example:

```
Circle x = someMethodThatCreatesACircle();
leadLinks.put(x, Alice);
x.addMember(Doug);
leadLinks.get(x);
```

The problem here is that the last line will not find x, since the hashCode will (with very high probability) calculate a different bucket than the one that x is actually in.

If we use a TreeMap, the code breaks, because the Circle class does not implement Comparable. However, even if it did, a very similar problem occurs due to mutability: If a circle adds a member, it would no longer be in the correct location in the TreeMap and we would be unable to find it when we do the usual BST search.

Less correct answer: The Circle class does not appear to implement equals (though it may be in the … section). This means that Java will use the default .equals method for determining whether an item is in the HashMap. Thus two identical Circles (same name, same members) could coexist in the HashMap, leading to confusion. In this case, TreeMap would resolve the issue since it can use the compareTo method to resolve equality.

Even less credit answer: A HashMap may have too many collisions leading to long chains. The given hashCode should be fairly robust, and there are more significant problems with the implementation. In this case, a TreeMap would resolve this issue, since TreeMaps are always balanced.

8. By the Numbers (3.5 Points).

For each of the scenarios below, give the correct numbers. On each line**, in the first blank write the minimum, and in the second blank write the maximum**. We define the height as the maximum number of links from the root to a leaf (so the tree in problem 1b has a height of 3, not 4). Each blank is worth 0.25 points (so don't burn all your time trying out bajillions of examples).

__3__ __14__ The minimum and maximum height of a BST with 15 nodes.

__1__ __14__ The minimum and maximum height of a Quick Union object with 15 elements where `isConnected(a, b)` returns true for every pair of items.

__1__ __3__ The minimum and maximum height of a Weighted Quick Union object with 15 elements where `isConnected(a, b)` returns true for every pair of items.

__1__ __3__ The minimum and maximum height of a 2-3-4 Tree containing 15 items. Recall that a node in a 2-3-4 tree may have 1, 2, or 3 items inside.

__3__ __5__ The minimum and maximum height of an LLRB set containing 15 items.

__3__ __3__ The minimum and maximum height of a binary heap containing 15 items.

__0__ __30__ The minimum and maximum number of items in a single bucket for a chaining hash table with 30 items and 15 buckets (i.e. with a load factor of 2).

Grading Scheme:

Each blank was worth 0.25 points, as specified in the problem.

We realized that some students may have accidentally counted height such that a tree with only the root had height of 1 instead of 0, despite our warning. Thus, we computed your grade as follows:

a) We scored this question with a height indexed from 0.
b) We scored this question again with a height indexed from 1. To get these answers, add one to the answer specified in the blank. **Under this scheme, the hash table question will still be indexed from 0 (since there is no ambiguity here).**

Your final grade for this question was $\max\{a, b - 0.25\}$. The 0.25 points that were subtracted from scheme (b) was the penalty for not reading the directions and indexing from height of 1.

Justification:

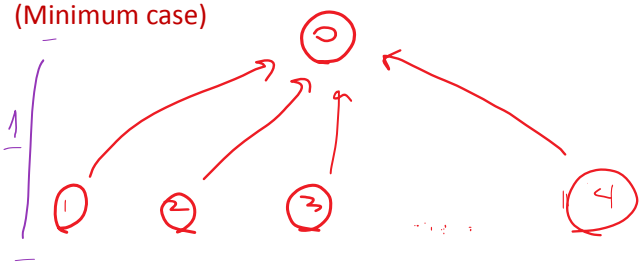
The minimum and maximum height of a BST with 15 nodes.

Minimum is a perfectly balanced binary search tree ("bushy" tree). Thus, the height is equivalent to $\lfloor \log_2 15 \rfloor = \mathbf{3}$. Maximum is a perfectly spindly tree, so the height is **14**.

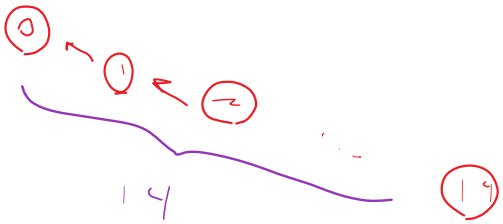
The minimum and maximum height of a Quick Union object with 15 elements where isConnected(a, b) returns true for every pair of items.

Minimum case is when you connect every node a to some node b to yield a height of **1**. Maximum case is when you effectively form a linked list from connecting each node, i.e. you connect with your "previous" node in some ordering, yielding a height of **14**.

(Minimum case)



(Maximum case)



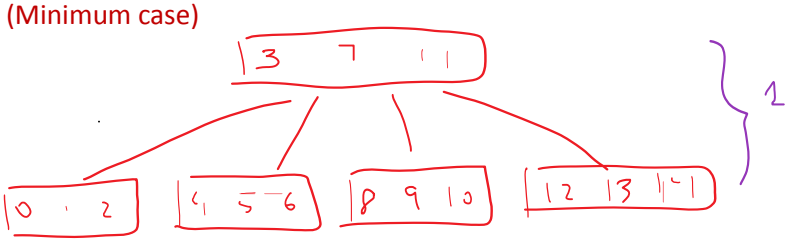
The minimum and maximum height of a Weighted Quick Union object with 15 elements where isConnected(a, b) returns true for every pair of items.

Minimum case is as before: you connect every node a to some node b to yield a height of $\mathbf{1}$. Maximum case, however, will link things together such that the shorter tree is joined with the larger one to minimize the height. Thus, the maximum case is now $\lfloor \log_2 15 \rfloor = \mathbf{3}$.

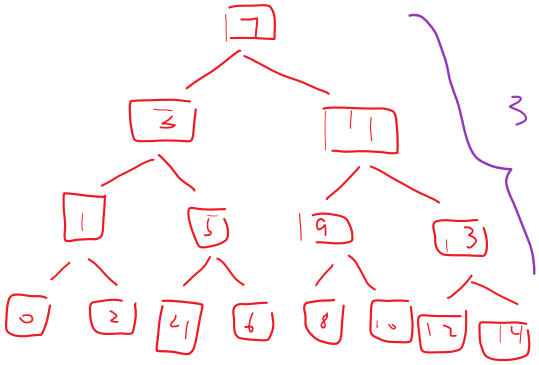
The minimum and maximum height of a 2-3-4 tree containing 15 items. Recall that a node in a 2-3-4 tree may have 1, 2, or 3 items inside.

The minimum height possible is if each node is as full as possible: if they each contain 3 items. This means that you will have up $\frac{15}{3} = 5$ nodes. This corresponds with a height of $\lfloor \log_4 15 \rfloor = \mathbf{1}$. The maximum height possible is if each node is as empty as possible: if they each contain 1 item. This means there are 15 nodes and thus $\lfloor \log_2 15 \rfloor = \mathbf{3}$.

(Minimum case)



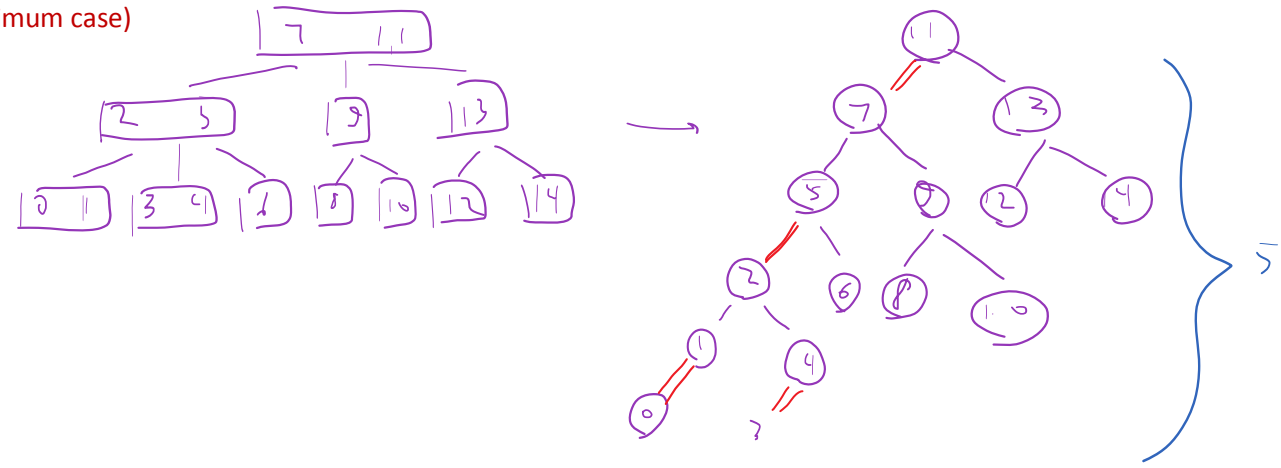
(Maximum case)



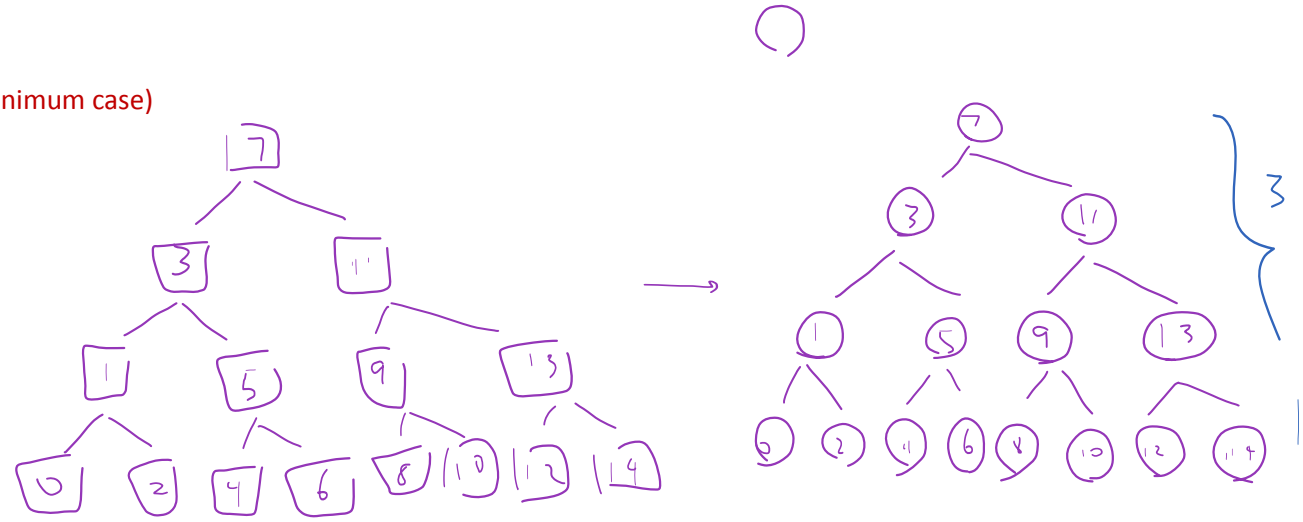
The minimum and maximum height of an LLRB set containing 15 items.

The minimum case is when we just have a tree full of 2-nodes. This gives us a height of $\lfloor \log_2 15 \rfloor = \mathbf{3}$. The maximum case is when we try to stuff as many 3-nodes as possible onto the left of the corresponding 2-3 tree, because we know that each 3-node will add a level into the tree.

(Maximum case)



(Minimum case)



The minimum and maximum height of a binary heap containing 15 items.

A binary heap is perfectly balanced, so the minimum and maximum height are both $\lfloor \log_2 15 \rfloor = \mathbf{3}$.

The minimum and maximum number of items in a single bucket for a chaining hash table with 30 items and 15 buckets. (i.e. with a load factor of 2)

Consider the case where we have a lousy hash function that hashes everything to the same bucket. Thus, we have a minimum of **0** items in one bucket and a maximum of **30** items in a bucket.

9. PNH (0 Points). Who was the *agoyatis* of Mr. Conchis?

Hermes

10. Quartiler (2.5 points)

Warning: This problem is particularly challenging. **Do not start until you feel like you've done everything else you can.** We will be award very little partial credit for this problem. Solutions which are correct but do not meet our time and space requirements (below) will be not be awarded credit.

The interface for the `Quartiler` interface is shown below.

```
public interface Quartiler<Item> {
      /* Adds an item to the Quartiler. */
      public add(Item x);
      /* Gets the item that is closest to the 75th percentile. */
      public getTopQuartile();
      /* Deletes the item that is closest to the 75th percentile. */
      public deleteTopQuartile();
}
```

For example, if we add the integers 1 through 100, then `getTopQuartile` will return 75. If we instead add the integers 1 through 4 to an empty `Quartiler`, then `getTopQuartile` will return 3. If there is a tie (e.g. if the `Quartlier` contains the integers 1 through 6), then ties may be broken arbitrarily. Design a data structure that supports these operations in amortized $O(\log N)$ time and $O(N)$ space.

a. Describe your data structure as concisely as possible while still including all relevant details. If you use existing data structure (for example, those listed in problem 6) as components, give them by name.

Note, we did not fully define Quartile and our definition was non-standard. As it happens, there are multiple standards, not in agreement, about what precisely a quartile is: http://en.wikipedia.org/wiki/Quartile. Consequently, you may have found our implicit definition a bit odd when it came to handling things like tie-breakers. It's not particularly important for this problem, so we did not include your tie-breaking as part of your score.

One approach is to create two heaps. One heap (lowerHeap) will track all values that are in the bottom 75% (not including the topQuartile value), and the top heap (upperHeap) will track all values in the top 25% (not including the topQuartile value). Finally, we will have a separate variable that tracks the actual topQuartile value.

When we insert a value, if it is larger than the topQuartile, we insert it into the upperHeap, and if it is smaller than the topQuartiler, we insert it into the lowerHeap. We enforce that the lowerHeap is always roughly 3 times the upperHeap as follows: If the lowerHeap is too large, we insert the topQuartile value into the upperHeap, delete the largest value from the lowerHeap and then move this value into the topQuartile variable. We do the same if the upperHeap becomes too large.

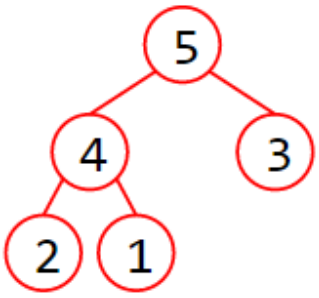
When we deleteTopQuartile, we take an item from the heap such that the same relative size invariant is maintained. For full credit on this problem, you were not required to derive the exact size invariant, particularly as quartile was not well defined. Simply stating that the lower heap should be roughly 3 times the lower heap was fine. We evaluated understanding of the invariant based on your drawing.

An alternate and more straightforward answer is to create a balanced binary search tree where each item tracks its subtree size (for example, the red black tree implementation from the book). In this case, we'd simply request the item of rank(3*N/4) from the tree. To find the item of rank(3*N/4), traverse the tree based on the subtree sizes instead of by key comparison. Stop when you find the tree with size closest to 3*N/4. However, such an approach also requires that we modify our LLRB so that it can somehow accommodate duplicates. Answers that simple mention using a tree were not awarded credit (as you have to handle both duplicate finding and selecting items of a particular rank, both non-trivial problems).

One common incorrect answer is to createa RedBlack tree and search for the topQuartile, which is $\Theta(N)$ in the worst case. Unless we store the subtree sizes inside each node of the tree, we cannot find the top quartile value at any greater speed.

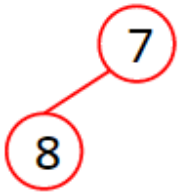
b. Draw your data structure after the following numbers have been added (in this order): 5, 1, 3, 2, 4, 6, 7, 8. You only need to draw the data structure after all 8 insertions have been completed.

To the left is our bottomHeap, and to the right our topHeap. We also have that topQuartile = 6. Note that under standard definitions of Quartile, we could have also had the topQuartile = 7 (leaving topHeap with only one value).



HeapMaxPQ lowerHeap

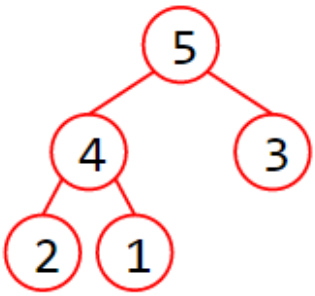
int TopQuartile



HeapMinPQ upperHeap

c. Draw your data structure after a subsequent call to `deleteTopQuartile` (which will remove the 6).

To the letter of our strange (and implicit) definition of topQuartile, our data structure would be as below:



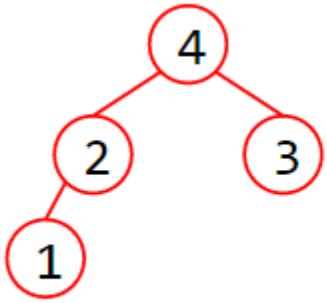
HeapMaxPQ lowerHeap

7

int TopQuartile

HeapMinPQ upperHeap

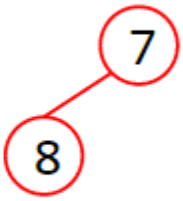
But for a more standard definition of topQuartile, it might be instead (which has the added benefit of having a nicer ratio of 2 as opposed to 5 in the relative sizes of the heaps):



HeapMaxPQ lowerHeap

5

int TopQuartile



HeapMinPQ upperHeap