Practice Midterm 2- Solutions

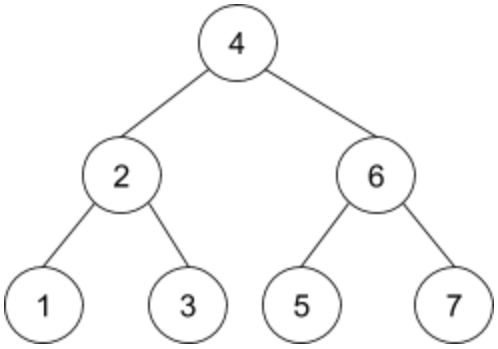
This test has 10 questions worth a total of 120 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write the statement out below in the blank provided and sign. You may do this before the exam begins.

#	Points	#	Points
1	15	6	12
2	9	7	12
3	0	8	9
4	6	9	12
5	20	10	25
		Total	120

1. Dat-uhhh Structures: (15 pts) Provide the solutions in the blanks under the questions. Each question is worth 3 points.

a) How many nodes would be in the left subtree of a complete binary search tree where height (h) > 0 and the root starts at a height of 0. Note, if you want to use height in your calculations, you must use the height of the full binary search tree, not subtrees.

2^h - 1: This can be seen by drawing a basic complete binary search tree as follows:



In the left subtree, there are 3 nodes and the height is 2. You may have needed to draw a larger Binary Search Tree to see the pattern.

b) How many nodes would be in the right subtree of the right subtree in a full binary search tree. The same assumptions should be made as in the previous problem

2^{h+1} - 1: This can be seen by drawing a basic binary search tree such as the one above. It is probably a larger one than this one would have had to been used because there was only 1 node in the right subtree of the right subtree.

c) How many different heaps can be both max heaps and min heaps? If there are none, justify why, tell us the cases that result in heaps that fit this criteria.

Infinitely many. If all the elements are all the same.

d) Can red-black trees have only black links? When is this the case?

Yes, it is possible for red black trees to only have black links. This happens when there is only 1 node or with very careful deletion. A red black tree with only black links would be the equivalent of a 2-3 tree with nodes that either have 0 children or 2 children. Additionally, a red-black tree with only black links would essentially just be a Binary Search Tree.

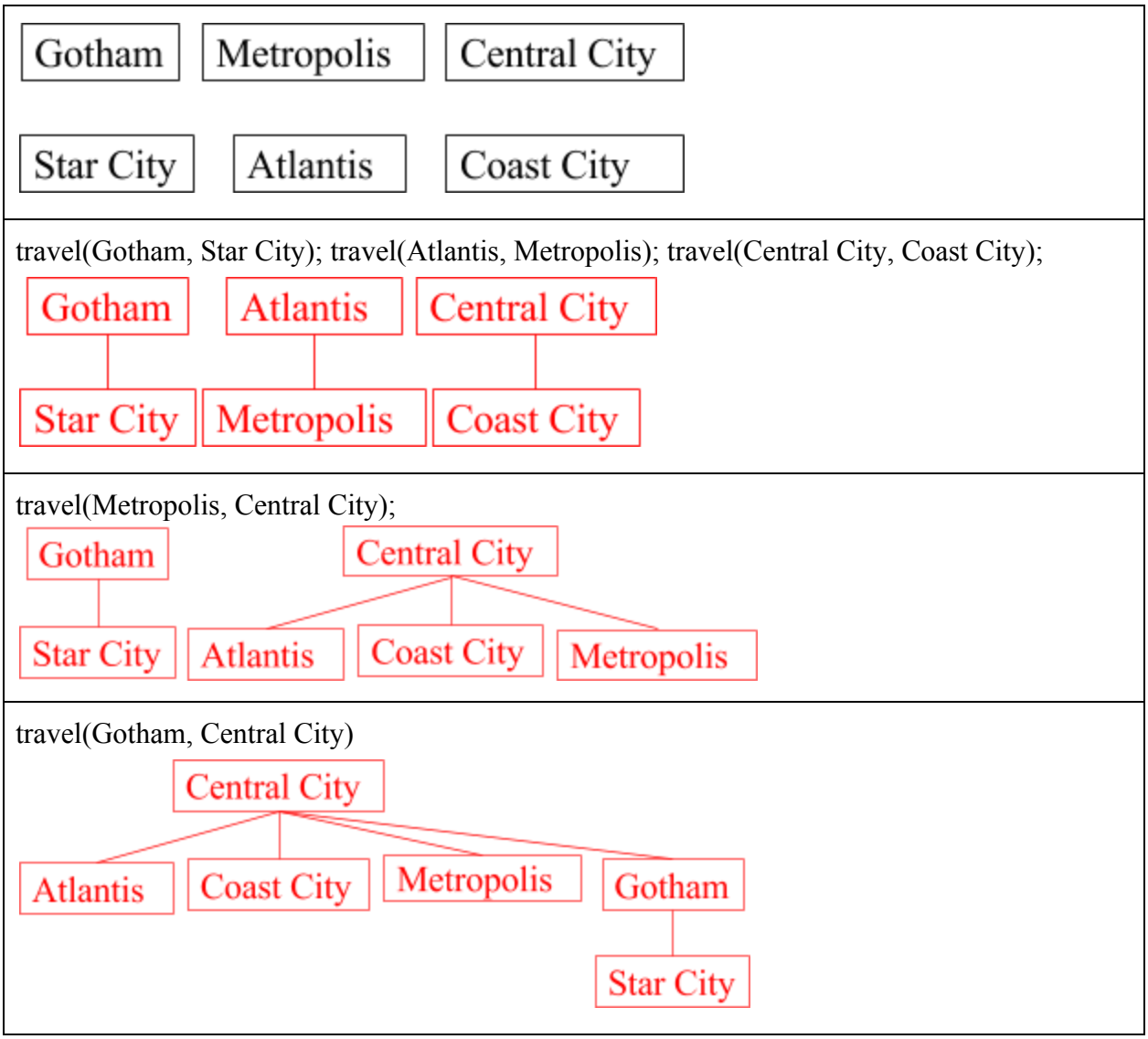
e) What is wrong with the following code snippet? Assume that it compiles properly, there is a constructor, and a hashcode method that has a hashcode function works well (spreads everything well).

```
Public class Human{
int legs;
int arms;
….
@Override
public boolean equals(Human no) {
        return legs == no.legs && arms == no.arms;
}
```

The main problem with this code is that the equals does not actually override any methods! The signature is wrong. Normally, the equals takes in an Object not a Human. This will cause errors when a “Human” is inputted.

2) Flash Union: (9 pts)

a) Beary Allen, a fast UC Berkeley student wants to make trips between cities. He wants to see if they are connected in the fastest way possible- as a result, he will use the fastest union data structure that we were taught. His *travel* method acts like *connect* or *union* in a normal union data structure. Draw the Union Structure in each box after all the commands have executed. (5 pts)



In this problem, it was important to understand the qualities of a Weighted Quick Union with Path Compression and that a Weighted Quick Union with Path Compression is the fastest Union Data Structure.

b) Beary Allen wants to determine the longest time that it would take for him to check if 2 cities are connected. Please write the runtime in terms of N, the number of cities. (2 pts)

Almost $\Theta(1)$ but not quite. Amortized $\Theta(1)$ is acceptable.

c) Beary decided that the runtime is not fast enough for the fastest man in the world. Can he go faster by changing the way the Union Data Structure works? If so, briefly describe how you would do so. (1 pt)

Not possible to argue in the scopes of the course. Another possible answer is that Beary can just run back in time and tell the city planners to put the cities closer together.

3) Riddle me this: (0 pts)

Which president wears the biggest hat?

The one with the biggest head.

4) Pitching and Catching: (6 pts) What does the following code display? You need not use all the lines.

```
  public static void triedTooHard(String[] n){
     try{
        for(int i = 0; i < n.length + 1; i++){
           System.out.println(n[i].charAt(0));
        }}
     catch(NullPointerException E){
        System.out.println("Tried" + n.length + "hard");}
     catch(IndexOutOfBoundsException E){
        String[] t = new String[n.length+1];
        System.arraycopy(n, 0, t, 1, n.length);
        triedTooHard(t);
     }
     finally{
        System.out.println(n.length);}}

public static void main(String[] args){
  triedTooHard(new String[] {"my ", "everything ", "my", "eternal"});
}
```

m
e
m
e
Tried5hard
Size is 5
Size is 4

The main tricky point in this problem is that the finally block happens AFTER everything in the normal try catch blocks. Because of this, the finally occurs after the recursive call has run to completion. The NullPointerException occurred in the second call because the item at the 0th index in the input array was null.

5) B-arbor Day: (20 pts) Woody TreePecker is a bit sad because he wants a place to perch. The only problem is he doesn't have a nice bushy tree to perch in- all he has is spindly LinkedLists. He has employed you, professional tree barber (or gardner whatever you want to call it), to give him some really nice trees. Now you, being the conservationist you are, want to turn the LinkedLists into Balanced Binary Search Trees.

Write the code that will convert a sorted LinkedList (one from java's documentation) into a Balanced Binary Search Tree (that can contain generics). For consistency, your main method that converts the LinkedList to a Binary Search Tree should be labeled **LinkedListToBST**. You will be given the remainder of this page and one more page to complete your class.

```
import java.util.LinkedList;

public class linkToBST {

  class TreeNode<T> {
     T item;
     TreeNode left, right;

     TreeNode(T item) {
        this.item = item;
        left = null;
        right = null;
     }
  }

  TreeNode LinkedListToBST(LinkedList l){
        return LinkedListToBST(l.size(), l);
  }
  TreeNode LinkedListToBST(int n, LinkedList l){
     if(n<=0){
        return null;
     }
     TreeNode left = LinkedListToBST(n /2 , l);
     TreeNode root = new TreeNode(l.removeFirst());
     TreeNode right = LinkedListToBST(n - n/2 - 1, l);
     root.left = left;
     root.right = right;
     return root;
```

```
    }
}
```

Since we are using the java implementation of a LinkedList, there was no need to create your own LinkedList node. Having a TreeNode class was pretty much a requirement because you needed to be able to store the left and right child. The method is relatively straightforward; however, it is very easy to trip up on the calculations.

We remove first when calculating the root because the first time that we want to make the element at index 0 into the leftmost element. We can take advantage of the fact that it is an ordered list to do this. The division by 2 allows us to “split” the list at that point so that everything remains balanced.

This is a relatively tricky problem, especially since it was hard to use your own LinkedList node (due to no private class allowed and thus no “static” head).

6) Test making you sad? Call that A-simp-totics: (12 pts) Write the Asymptotic runtime of the following function. Use the tightest bounds possible. Each problem is worth 4 points.

1. $\Theta(N*2^N)$

```
public static void louwTheWay(int N){
 for int(i = 0; i < N / 2 ; i ++){
    System.out.println("you");
  }
  louwTheWay(N - 1); louwTheWay(N-2);\\lie
}
```

We must realize that there are N layers. Additionally, the amount of work done per node is N. The nodes per layer is 2^i ,where i is the current layer, because of the 2 recursive calls. The summation would look as follows. $\sum_{i=0}^{N} N * 2^i \rightarrow N*2^N$

2. $\Theta(\log(n))$

```
public static void takeCaretik(int N){
      for(int i = 0; i < (N*1000+30^100)/N ; i ++){
           System.out.println("I'll take care of you.");
           }
   takeCaretik(N/4);
}
```

The amount of work done in the for loop is constant (though a very large constant), so we disregard it and just consider it constant time. The amount of layers would be $\log_4 N$. Since we have constant work on each layer, 1 node per layer, and $\log_4 N$ layers, the runtime is $\Theta(\log(n))$.

3. $\Theta(N^2)$

```
public static void weDontTalk(int N){
    for(int i = 0 ; i < N; i++){
       for (int j = 0 ; j < i; j++){
         System.out.print("Anymore")
         }
     }
     weDontTalk(N/2) \\Like we used to
}
```

The amount of work done in the second for loop is proportional to the work done in the first for loop- for the first iteration, this will be N^2; however, we have a recursive loop that we must take into account. The amount of work done per now is $(\frac{N}{2^i})^2$ or $(\frac{N^2}{4^i})$ where i is the current layer.

The amount of layers is log(N) because we divide by 2 at each recursive call. This makes the summation formula $\sum_{i=0}^{log(N)} (\frac{N^2}{4^i}) \rightarrow (\frac{N^2}{4^{log(N}}) \rightarrow N^2$

7) MashedSet: (12 pts)

a) We have a HashSet, but a goon broke part of our code so it does not handle duplicates properly. This means that duplicates can added without care. Luckily, your hashcode, which is quite nice, is still intact (it was encrypted well), along with this, the spread of the hashcode is pretty good. Determine how long it would take in order to traverse your broken HashSet to find and delete ALL duplicates of ONE key. For example, how long would it take to find the key "macaroni". Provide the runtime (It is your choice what symbol to use) based off our prior assumptions. Give reasons for both runtimes. Correct runtimes with wrong explanations will be given 0 points.

Ω(1)- We have a good hashcode and a good spread, and since the key would be the same, it would always hash to the same bucket, this means that you would have to check very few elements.

O(N)- Technically this can happen if every single element in our HashMap is the same. It is not acceptable to say that the hashcode is bad.

b) You're trying to figure out how the goon managed to break your code so easily. You spent so many hours on it, and he managed to change it very quickly so that duplicates aren't handled. What is the most probable way he did this?

A change in the equals method for the item being hashed would result in the duplicates not being handled. Your equals method normally goes through the bucket to check if the element is not working, but if your equals method does not work, it won't realize there is a duplicate.

A NOT acceptable response would be that the hashcode function was screwed up as in the original problem it was stated that it remained intact.

c) Pretend that you did not remove the duplicates from the faulty HashSet and you wanted to keep track of how many times each item was inside of the HashMap. How would one do this? We would like our solution to be as fast as possible.

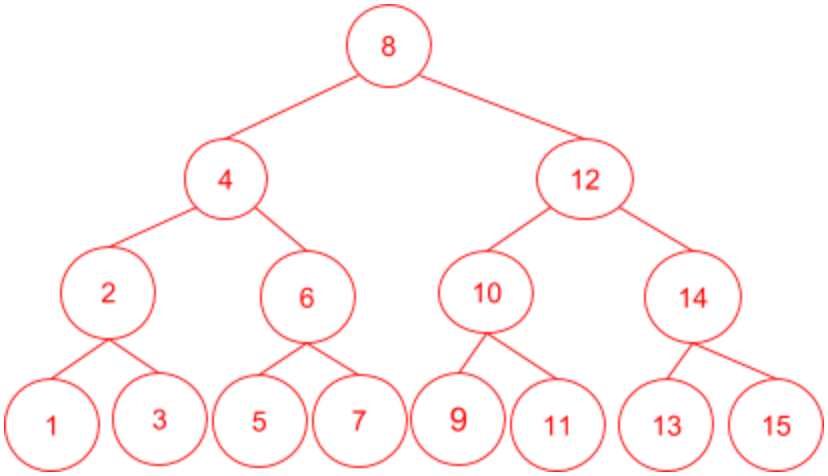
The most intuitive way would be to create a HashMap where you insert elements as you go through the original HashSet. The element would be a node that has a the key and the count. Every time you found a duplicate, you would find the node and change the value. This would take $\boldsymbol{\Theta}(1)$ because we would use the SAME hashcode that was provided to us earlier in the problem. The overall runtime as a result would be $\boldsymbol{\Theta}(N)$

8) Rotations and Flips: (9 pts)

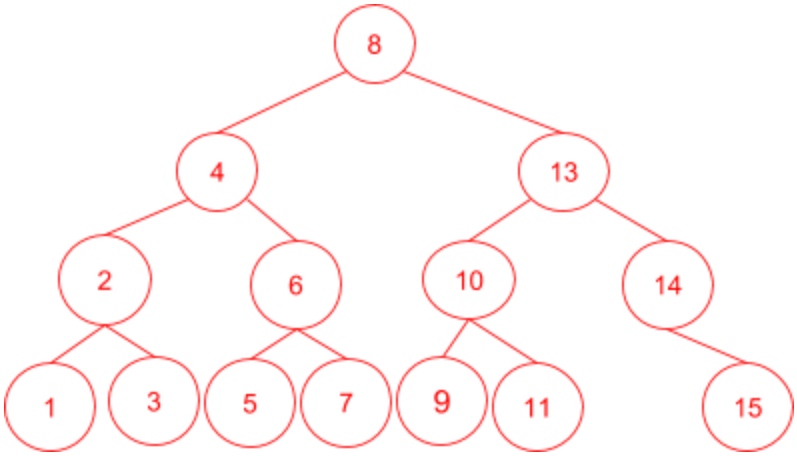
a) What series of inserts would result in a perfectly balanced binary search tree for the following input: 1, 2, 3, 4, 5, 6 , 7 , 8 , 9 , 10, 11, 12, 13, 14, 15. Draw the resulting Binary Search Tree.

8, 12,4, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15

A way that makes this a bit easier to approach is construct a 2-3 tree and see where the nodes get promoted to. It decreases the randomness and allows you to do it relatively quickly.

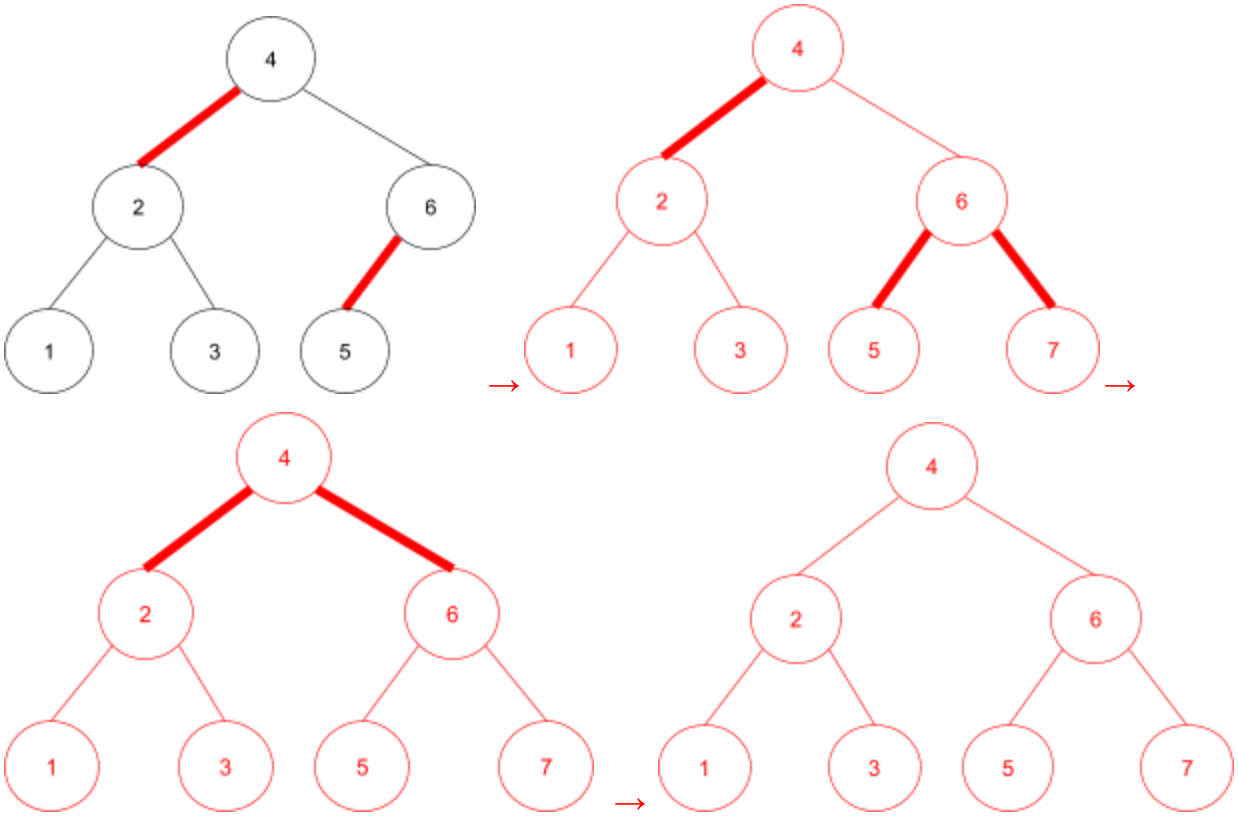


b) Delete the item 12 from the above Binary Search Tree and draw the results.



To do this, you want to do the “hibbard” deletion where you find the smallest item on the right side of the element and then swap it with the item

c) Insert 7 into the following red black tree. Show your steps using the red-black tree method for full credit. ½ credit will be given for converting into a 2-3 tree and back.



Problem is straightforward if you know the rules well

9) Runtimes not finished call that Not dun-times: (12 pts) Write all runtimes in Theta or Omega and O. Ignore constants but do not ignore lower order terms.

a) Runtime of putting N presorted keys into a binary search tree.
$\boldsymbol{\Theta}(N^2)$ The items are presorted (either least to greatest or greatest to least) and this means that the Binary Search Tree will end up looking like a LinkedList

b)You have a minheap and you will perform deleteMin operations until the min-heap is empty. Whenever an element is deleted, it is added to a Binary Search Tree.
$\boldsymbol{\Omega}(NlogN)$- Deleting the min takes log(N) time and you perform that operation N times. Then, in best case, inserting into a Binary Search Tree takes $\boldsymbol{\Theta}(log(N))$ time, it is done N times. Accounting for both operations makes the value $\Omega(NlogN + NlogN) = \Omega(2NlogN)$

$\mathbf{O}(N^2 + NlogN)$ - Deleting the min takes log(N) time and you perform that operation N times. The N^2 comes from the worst case runtime of inserting into a Binary Search Tree being $\boldsymbol{\Theta}(N)$ time and that operation is performed N times ($N*N = N^2$)

c) Inserting N elements into a HashMap that all have the same key and different values.
$\boldsymbol{\Theta}(N)$- You insert N elements and since HashMaps allow for no duplicates, you will never have more than 1 element in your HashMap.

d) Given an array sorted from least to greatest put all the elements into a stack (lowest index to highest index)then pop off all the elements. Each time an element is popped off put it into a maxheap.
$\boldsymbol{\Theta}(N)$- The items are presorted and since it is put into the stack, the greatest elements are always above the smaller elements. As a result, no swimming will ever need to be done. We put N elements into a stack, pop off all N elements and insert them into a heap with no swimming so N insert operations. This results in 3N or $\boldsymbol{\Theta}(N)$

10) NBA: National Bongoola Association: (25 pts)
You are the coach of a prestigious Bongoola team. Your job as coach is to make sure that, at any given time, the best players on your team for each positions are on the field. Bongoola, being a great community team sport, invites all people of the community to join. Each player will be assigned a position that they can play- there are a total of **M** positions- and a unique team number. For this problem, we will use **N** to be the total amount of people on your team and **P** to be the people signed up for a certain position.

You have your starting **M** players, but as the game progresses, you will need to replace them. You will base your replacement of people using the *bodacious* factor. During the game, only 1 player's bodacious factor goes down, and once they are benched, their bodacious factor doesn't change. Every **Z** minutes, 1 player on the field will have their bodacious factor decrease and you will be able to replace the player if needed. Once the next eligible player's (a player who plays the same position) bodacious factor exceeds the current player, they are considered a better choice. You are guaranteed that only 1 of the players at a time will need to be replaced. Changing a player's bodacious factor should take $\boldsymbol{\Theta}$(l) time, replacing a player should take $\boldsymbol{\Theta}$(log(P)) time, and putting a new player in the current lineup should take $\boldsymbol{\Theta}$(1) time.

On the next page, write a detailed description of what data structures you will use to make the operations stated above run in the provided time. State any assumptions you need to regarding anything (no you cannot assume that everything is done by some magical warlock).

First thing is first, you will need a **Player** class with a *bodacious* instance variable. You will then create a HashMap which will hash based off the unique player id's. Because of the HashMap, we can change the *bodacious* factor of a player in $\boldsymbol{\Theta}$(1) time. For this part of the problem, you need to state that you assume that there is a good hashcode and good spread- if this was not the case, it would be possible to have $\boldsymbol{\Theta}$(N) runtime.

To represent the players who can play in a certain position, we will use a priority queue. The top player in the priority queue will be in the current lineup. Once their *bodacious* factor is less that one of their children, we will sink the player. This results in $\boldsymbol{\Theta}$(log(P)) time to replace a player (if they truly need to be replaced).

Now it gets a little tricky. Since there are M positions, we cannot disregard the amount of positions as a constant. Two get around this, there are two possible (intuitive) solutions:

The first solution involves realizing that there are as many priority queues as there are positions. You can create 2 M sized arrays- one array will be the current lineup and the other array will be composed of priority queues in the same order. You would modify the **Player** class so that it has

an instance variable that stores the index that it would go to when replacing or being replaced. Accessing an array would occur a maximum of 2 times which would make the runtime $\Theta(1)$ time.

The second solution would be slightly more complicated. You can create 2 HashMaps, again, one of the current lineup and one of the priority queues. However, since the hashing of priority queues is not straightforward, we would create a wrapper class for priority queues and create some instance variable that was unique to each priority queue. For example, we could have a string that would be the name of the position it represents. We then hash the M wrapper classes. Accessing a priority queue would take $\Theta(1)$ time if we assume a good hashcode. If good hashcode was not assumed, then it would take $\Theta(M)$ time

The complexity in this solution (specifically when you attempt to hash the lineup) is that players who replace others would need to have the same hashcode. To do this, the key in this HashMap should be the value of its corresponding wrapper classes priority queue (in our example the name of the position) and the value should be the player. Replacing the item in the HashMap would take $\Theta(1)$ time since we know where it hashes to.

Since the *bodacious* factor only decreases at the end of the Z minutes, we can just make it so that when a sinking occurs, the player is changed.