UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #2, Spring 2018

This test has 9 questions worth a total of 240 points and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to two double sided written cheat sheets (can use front and back on both sheets). 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.**

"I have neither given nor received any assistance in the taking of this exam."

Signature: ___________________________

#	Points	#	Points
0	1	6	14
1	24	7	46
2	28	8	46
3	32	9	30
4	0		
5	20		
		TOTAL	240

Name: ___________________________

SID: ___________________________

Three-letter Login ID: __________

Login of Person to Left: ________

Login of Person to Right: ______

Exam Room: ______________________

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- Not all information provided in a problem may be useful, and **<u>you may not need all lines</u>.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs in the exam, we'll announce a fix. **<u>Unless we specifically give you the option, the correct answer is not 'does not compile.'</u>**
- ○ indicates that only one circle should be filled in.
- ☐ indicates that more than one box may be filled in.
- For answers which involve filling in a ○ or ☐, please fill in the shape completely.
- **Throughout the exam, assume that hash table resizing takes linear time.**

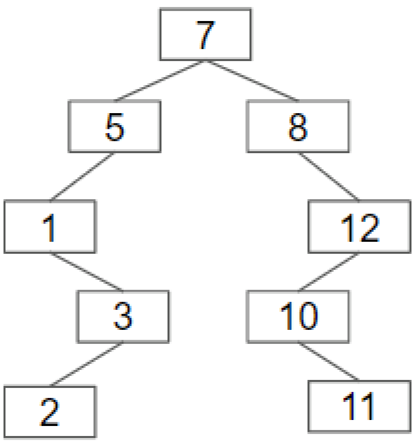
Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

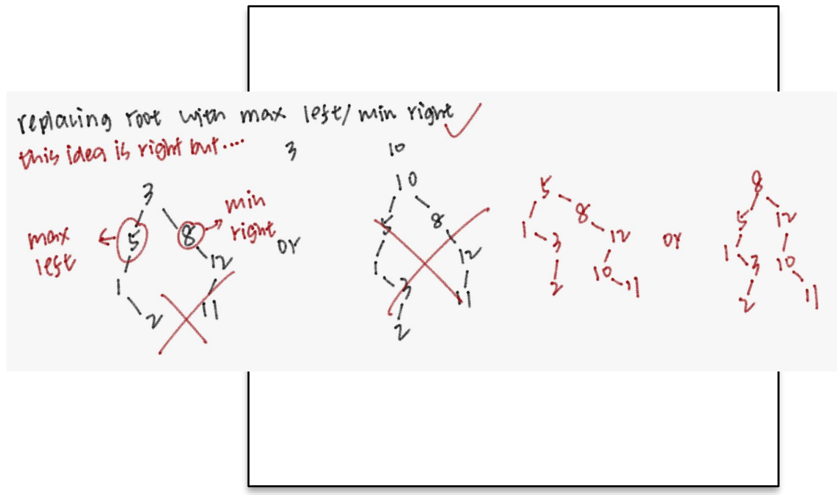
Before exam: [☹________________☺].
After exam: [☹________________☺].

0. So it begins (1 point). Write your name and ID on the front page. Write the exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your login in the corner of every page. Enjoy your free point ☺.

1. Tree time.

a) **(4 points).** Suppose we have the BST shown below. Give a valid tree that results from deleting “7” using the procedure from class (a.k.a. Hibbard deletion). Draw your answer to the right of the given tree in the box.



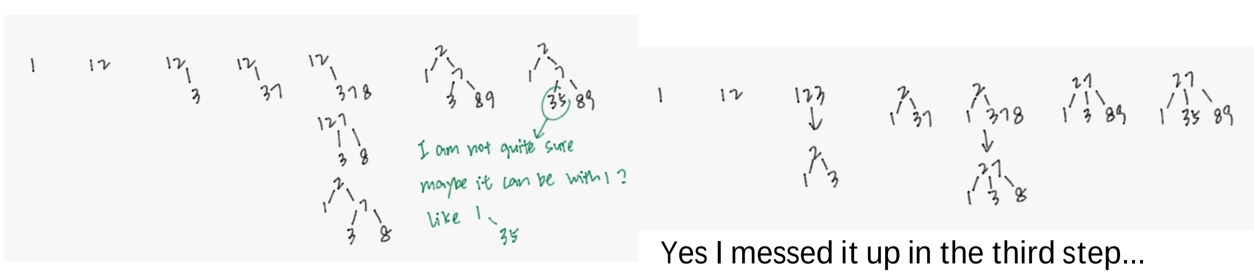


b) **(4 points).** Give an example of a rotation operation on the original tree from 1a (on the left) that would increase the height. **<u>You do not need to draw the tree</u>**, just write the operation, e.g. “rotateRight(11)”.

rotateRight(7)

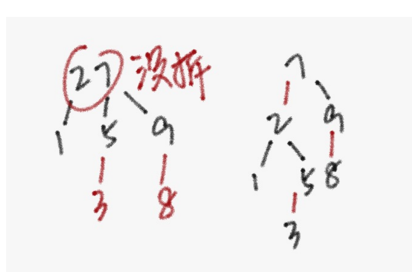
I know the operation of rotating trees, but I am not sure as to how to take the parameters
I think I should take the node that is going to be the left/right child of another node

c) **(4 points).** Draw the 2-3 tree that results from inserting 1, 2, 3, 7, 8, 9, 5 in that order.

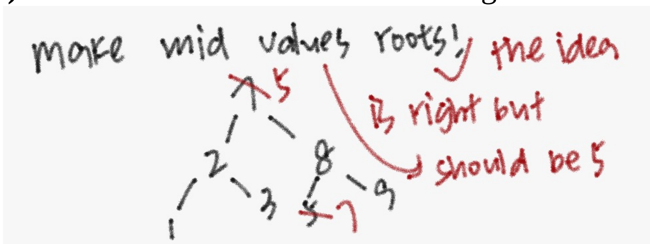


Yes I messed it up in the third step...

d) **(3 points).** Draw the LLRB that results from inserting 1, 2, 3, 7, 8 9, 5 in that order. Write the word “red” next to any red link.



e) **(3 points).** Draw a valid BST of minimum height containing the keys 1, 2, 3, 7, 8 9, 5.



f) (6 points). Under what conditions is a **complete** BST containing N items **unique**? By unique we mean the BST is the only complete BST that contains exactly those N items. By complete we mean the same idea that was required for a tree to be considered a heap (not repeated here). Reminder: We never allow duplicates in a BST.

-6

N = 2^s - 1, s = 1, 2, 3, ...

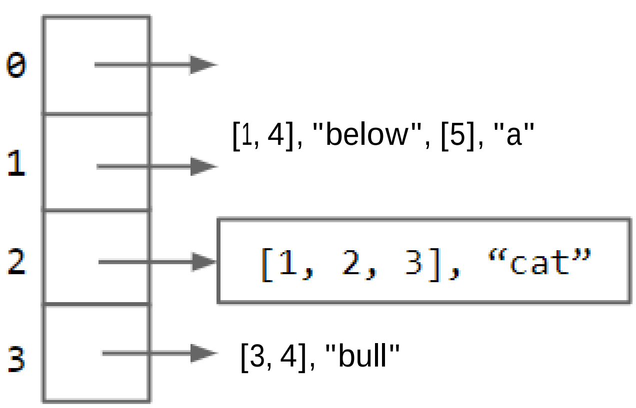
no, n can be any non-negative value; recall the definition of a complete tree, which can still have empty spots

2. Hash Tables.

a) **(5 points).** Draw the hash table that is created by the following code. Assume that `XList` is a list of integers, and the hash code of an `XList` is the sum of the digits in the list. Assume that `XLists` are considered equal only if they have the same length and the same values in the same order. Assume that `FourBucketHashMaps` use external chaining and that new items are added to the end of each bucket. Assume `FourBucketHashMaps` always have four buckets and never resize. The result of the first `put` is provided for you. Represent lists with square bracket notation as in the example given.

```
FourBucketHashMap<XList, String> fbhm = new FourBucketHashMap<>();
fbhm.put(XList.of(1, 2, 3), "cat");
fbhm.put(XList.of(1, 4), "riding");
fbhm.put(XList.of(5), "a");
fbhm.put(XList.of(3, 4), "bull");
fbhm.put(XList.of(1, 4), "below");
```

hashmap contains both keys and values



-1.5

b) **(4.5 points).** Next to the calls to `get`, write the return value of the `get` call. Assume that `get` returns `null` if the item cannot be found.

```
FourBucketHashMap<XList, String> fbhm = new FourBucketHashMap<>();
XList firstList = XList.of(1, 2, 3);
fbhm.put(firstList, "cat");
fbhm.get(XList.of(1, 2, 3));                    "cat"
firstList.addLast(0); // list is now [1, 2, 3, 0]
fbhm.get(firstList);                            null  "cat"
fbhm.get(XList.of(1, 2, 3));                    null
```

the hashmap stores the key list as an reference, not a copy since the new list has the same hashcode as before, we will go into the correct bucket, and find the reference and its value

c) **(10.5 points).** Next to the calls to `get`, write the return value(s) of the `get` call. Assume that `get` returns `null` if the item cannot be found.

```
FourBucketHashMap<XList, String> fbhm = new FourBucketHashMap<>();
XList firstList = XList.of(1, 2, 3);
fbhm.put(firstList, "cat");
firstList.addLast(1); // list is now [1, 2, 3, 1]
fbhm.get(firstList);                            null
fbhm.get(XList.of(1, 2, 3));                    null
fbhm.get(XList.of(1, 2, 3, 1));                 null
fbhm.get(XList.of(3, 4));                       null
fbhm.put(firstList, "dog");
fbhm.get(firstList);                            "dog"
fbhm.get(XList.of(1, 2, 3));                    null
fbhm.get(XList.of(1, 2, 3, 1));                 "dog"
```

d) **(4 points).** What are the best and worst case `get` and `put` runtimes for `FourBucketHashMap` as a function of N, the number of items in the HashMap? Don't assume anything about the distribution of keys.

what do you mean?

- `.get` best case: _Θ__1______
- `.get` worst case: _Θ__N______
- `.put` best case: _Θ__1______
- `.put` worst case _Θ__N______

e) **(4 points).** If we modify `FourBucketHashMap` so that it triples the number of buckets when the load factor exceeds 0.7 instead of always having four buckets, what are the best and worst case runtimes in terms of N? Don't assume anything about the distribution of keys.

- `.get` best case: _Θ__1______
- `.get` worst case: _Θ__N______
- `.put` best case: _Θ__1______
- `.put` worst case _Θ__N______

As noted on the front page, throughout the exam you should assume that a single resize operation on any hash map takes linear time.

3. Weighted Quick Union.

a) **(10 points).** Define a "fully connected" `DisjointSets` object as one in which `connected` returns true for any arguments, due to prior calls to `union`. Suppose we have a fully connected `DisjointSets` object with **<u>6 items</u>**. Give the best and worst case height for the two implementations below. The height is the number of links from the root to the deepest leaf, i.e. a tree with 1 element has a height of 0. **<u>Give your answer as an exact value</u>**. Assume Heighted Quick Union is like Weighted Quick Union, except uses height instead of weight to determine which subtree is the new root.

	Best Case Height	Worst Case Height
Weighted Quick Union	1	2
Heighted Quick Union	1	2

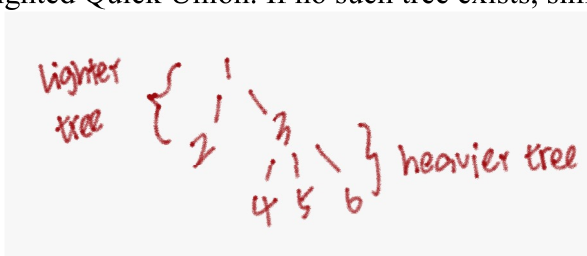
b) **(8 points).** Suppose we have a Weighted Quick Union object of height H. Give a general formula for the minimum number of objects in a tree of height H as a function of H. Your answer must be exact (e.g. not big theta).

2^H

c) **(6 points).** Draw a Quick Union tree of 6 objects or fewer that would be **<u>possible</u>** for Heighted Quick Union, but **<u>impossible</u>** for Weighted Quick Union. If no such tree exists, simply write "none exists."

-6

none exists



d) **(8 points).** Create a set for storing `SimpleOomage` objects Assume that `hashCode` for `SimpleOomage` is the perfect hashcode you were expected to write in HW3, where hash code values are unique and always between 0 and 140,607, inclusive.

```
public class SimpleOomageSet {
  private WeightedQuickUnionUF wq = new WeightedQuickUnionUF(140609);
  public void add(T item) {
      wq.union(140608, item.hashCode())
  }
  public boolean contains(T item) {
      return wq.connected(140608, item.hashCode())
  }
} // reminder: WeightedQuickUnionUF methods are union() and connected()
```

4. PNH (0 points). This 1996 simulation video game by Maxis had a hidden feature introduced secretly by a programmer, where on certain dates of the year, "muscleboys in swim trunks" would appear by the hundreds and hug and kiss each other.

5. Multiset. The `Multiset` interface is a generalization of the idea of a set, where items can occur multiple times.

```
public interface Multiset<T> {
    public void add(T item);          // adds item.
    public boolean contains(T item); // true if item occurs at least once.
    public int multiplicity(T item); // number of times item occurs.
}
```

For example, if we call `add(5)`, `add(5)`, `add(10)`, `add(15)`, `add(5)`, then the resulting Multiset contains {5, 5, 10, 15, 5}. In this case, `multiplicity(5)` will return `3`.

a) **(12 points).** A 61B student suggests that **one way to implement Multiset is to <u>modify a BST</u>** so that it is instead a "Trinary Search Tree", where the left branch is all items less than the current item, the middle branch is all items equal to the current item, and the right branch is all items greater than the current item. The multiplicity is then simply the number of times that an item appears in the tree. Implement the `add` method below.

```
public class TriSTMultiset<T extends Comparable<T>> implements Multiset<T> {
    private class Node {
        private T item;
        private Node left, middle, right;
        public Node(T i) { item = i; }
    }
    Node root = null;
    public void add(T item) {
        root = add(item, root);
    }
    private Node add(T item, Node p) {
        if (p == null) { return new Node(item) }
        int cmp = item.compareTo(p.item);
        if (cmp < 0) {
            p.left = add(item, p.left);
        } else if (cmp > 0) {
            p.right = add(item, p.right);
        } else {
            p.middle = add(item, p.middle)
        }
        return p;
    } ...
```

-4 b) **(4 points).** Let X be an item with multiplicity M, and let N be the number of nodes in the tree. Give an Omega bound for the best case runtime of any possible implementation of `multiplicity(X)` for a `TriSTMultiset`. Give the tightest possible bound you can.

Ω(log N   M   )

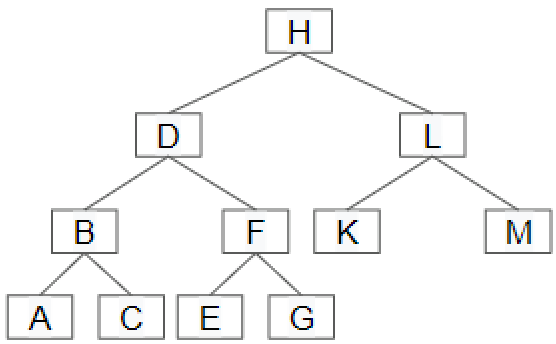
I don't get it, cuz finding x takes log N time; unless x is
always at the root, or at least very close to the root?

c) **(6 points).** Rather than implementing an entirely new data structure from scratch, we might consider using delegation or extension to implement `Multiset`. Which is better, delegation or extension? If delegation, what class should you delegate to? If extension, what class should you extend? **<u>If applicable, provide generic types.</u>** Fill in one of the bubbles and the corresponding blank below.

java.util.TreeMap<T, Integer>

◯ Delegation to an instance of the hashMap<T, Integer>______________ class is better.
◯ Extending the ________________________________ class is better.

6. Min Heaps (14 points). Consider the min heap below, where each lette ***represents*** some value in the tree. For each question, indicate which letter(s) correspond to the specified value. **Assume each value in the tree is <u>unique</u>.**



a) Assuming values are inserted into the heap in **increasing order**, indicate all letters which could represent the following values:

Smallest value: ☐A ☐B ☐C ☐D ☐E ☐F ☐G ☐H ☐K ☐L ☐M

Median value: ☐A ☐B ☐C ☐D ☐E ☐F ☐G ☐H ☐K ☐L ☐M

Largest value: ☐A ☐B ☐C ☐D ☐E ☐F ☐G ☐H ☐K ☐L ☐M

b) Assuming values are inserted into the heap in **decreasing order**, indicate all letters which could represent the following value:

Smallest value: ☐A ☐B ☐C ☐D ☐E ☐F ☐G ☐H ☐K ☐L ☐M

c) Assuming values are inserted into the heap in an **unknown order**, indicate all letters which could represent the following values:

only D and H can be eliminated, because they cannot have five smaller counterparts

Median value: ☐A ☐B ☐C ☐D ☐E ☐F ☐G ☐H ☐K ☐L ☐M

Largest value: ☐A ☐B ☐C ☐D ☐E ☐F ☐G ☐H ☐K ☐L ☐M

-4

all leaf-nodes

7. Iteration.

a) **(12 points).** Fill in the `toList` method. It takes as input an `Iterable<T>`, where `T` is a generic type argument, and returns a `List<T>`. If any items in the `iterable` are null, it should throw an `IllegalArgumentException`. You should use the for each notation. Do not use `.next` and `.hasNext` explicitly.

what does this mean?

```
public class IterableUtils {
    public static <T> List<T> toList(Iterable<T> iterable) {
        List<T> ans = new ArrayList<>();
        for (T item : iterable) {
            if (item == null) {
                throw new IllegalArgumentException();
            }
            ans.add(item);
        }
        return ans;
    }
} // assume any classes you need from java.util have been imported
```

b) **(8 points).** The `ReverseOddDigitIterator` implements `Iterable<Integer>`, and its job is to iterate through the odd digits of an integer in reverse order. **For example, the code below will print out `77531`.**

```
ReverseOddDigitIterator rodi = new ReverseOddDigitIterator(12345770);
for (int i : rodi) {
    System.out.print(i);
}
```

Write a JUnit test that verifies that `ReverseOddDigitIterator` works correctly using your `toList` method from before. Use the `List.of` method, e.g. `List.of(3, 4, 5)` returns a list containing 3 then 4 then 5.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestRODI {
  @Test
  public void testRODI() {
      ReverseOddDigitIterator odi = new ReverseOddDigitIterator(12345770);
      List<Integer> expected = List.of(7, 7, 5, 3,1);
      List<Integer> actual = IterableUtils.toList(odi);
      assertEquals(expected, actual);
  }
} // assume any classes you need from java.util have been imported
```

c) **(18 points).** Fill in the implementation of the `ReverseOddDigitIterator` class below.

```
public class ReverseOddDigitIterator implements Iterable<Integer>,
                                                 Iterator<Integer> {
    private int value;
    public ReverseOddDigitIterator(int v) {
        value = v;
    }
    public boolean hasNext() {
        if (value == 0) {
            return false_________________;       // hint: this class should
        }                                        // be implemented
        if (value % 2 == 1_______________) {     // so that the example
            return true______________;           // code that prints
        } else {                                 // 77531 on the previous
            value = value / 10;                  // page works.
            return hasNext()_________________;
        }
    }

    public Integer next() {
        int d = value % 10_____________________;
        value = value / 10_____________________;
        return d;
    }
    public Iterator<Integer> iterator() {
        return new ReverseOddDigitIterator();
    }
} // assume any classes you need from java.util have been imported
```

the new instance will not have the same "value"

this -2

-2

d) **(8 points).** If you didn't complete part c, assume it is completed and compiles. **For each of the following, which file (if any) will fail to compile as a <u>direct result</u> of the removal?** By "direct result", we mean the compilation failure is not caused by a dependency failing to compile.

Suppose we remove "`implements Iterable<Integer>`", which file will **fail** to compile?
○ `IterableUtils` ● `TestRODI` ○ `ReverseOddDigitIterator` ○ None

Suppose we instead remove `implements Iterator<Integer>`, which file will **fail** to compile?
○ `IterableUtils` ○ `TestRODI` ● `ReverseOddDigitIterator` ○ None

Suppose we instead remove the `hasNext` method, which file will **fail** to compile?
○ `IterableUtils` ○ `TestRODI` ● `ReverseOddDigitIterator` ● None

Suppose we instead remove the `iterator` method, which file will **fail** to compile?
○ `IterableUtils` ○ `TestRODI` ● `ReverseOddDigitIterator` ○ None

if the current class implements the Iterator interface, then it must implement hasNext and Next methods

8. Asymptotics

a) **(12 points).** Give the runtime of the following functions in Θ notation. Your answer should be a function of N that is as simple as possible with no unnecessary leading constants or lower order terms. **<u>Don't spend too much time on these</u>**!

_Θ_N^6_

```
public static void g1(int N) {
        for (int i = 0; i < N*N*N; i += 1) {
            for (int j = 0; j < N*N*N; j += 1) {
                System.out.print("fyhe");
            }
        }
    }
```

_Θ_2^N_

```
public static void g2(int N) {
        for (int i = 0; i < N; i += 1) {
            int numJ = Math.pow(2, i + 1) - 1; // <-- constant time!
            for (int j = 0; j < numJ; j += 1) {
                System.out.print("fhet");
            }
        }
    }
```

_ Θ_N___

```
public static void g3(int N) {
        for (int i = 2; i < N; i *= i) {}
        for (int i = 2; i < N; i++) {}
    }
```

b) **(4 points).** Suppose we have an algorithm with a runtime that is $\Theta(N^2 \log N)$ in all cases. Which of these statements are definitely true about the runtime, definitely false, or there is not enough information (NEI)?

$O(N^2 \log N)$	● True	○ False	○ NEI
$\Omega(N^2 \log N)$	● True	○ False	○ NEI
$O(N^3)$	● True	○ False	○ NEI
$\Theta(N^2 \log_4 N)$	● True	○ False	○ NEI

c) **(6 points).** Suppose we have an algorithm with a runtime that is $O(N^3)$ in all cases.

There exists some inputs for which the runtime is $\Theta(N^2)$	○ True	○ False	● NEI
There exists some inputs for which the runtime is $\Theta(N^3)$	○ True	○ False	● NEI
There exists some inputs for which the runtime is $\Theta(N^4)$	○ True	● False	○ NEI
The worst case runtime is $O(N^3)$	● True	○ False	○ NEI
The worst case runtime has order of growth N^3	○ True	○ False	● NEI

d) **(12 points).** Give the best and worst case runtime of the following functions in Θ notation. Your answer should be as simple as possible with no unnecessary leading constants or lower order terms. **<u>Don't spend too much time on these</u>**! Assume K(N) runs in constant time and returns a boolean.

```
public static void g4(int N) {
    if (N == 0) { return; }
    g4(N - 1);
    if (k(N)) { g4(N - 1); }
}
```

Best case: _Θ__N____
Worst case: _Θ__2^N___

```
public static void g5(int N) {
    if (N == 0) { return; }
    g5(N / 2);
    if (k(N)) { g5(N / 2); }
}
```

Best case: _Θ__log N___
Worst case: _Θ__N____

e) **(6 points).** Give the best and worst case runtime of the code below in terms of N, the length of x. Assume `HashSets` use the idea of external chaining with resizing used in class, and that resize is linear.

```
public Set<Planet> uniques(ArrayList<Planet> x) {
    HashSet<Planet> items = new HashSet<>();
    for (int i = 0; i < x.size(); i += 1) {
        items.add(x.get(i));
    }
    return items;
}
```

Best case runtime for `uniques`: _ Θ_N___ Worst case runtime for `uniques`: __Θ_N^2__

f) **(6 points).** Consider the same code from part b, but suppose that instead of `Planets`, `x` is a list of `Strings`. Suppose that the list contains N strings, each of which is length N. Give the best and worst case runtime.

Best case runtime for `uniques`: _ Θ_N___ Worst case runtime for `uniques`: __Θ_N^3__

9. **(30 points).** Imagine that we have a list of every commercial airline flight that has ever been taken, stored as an `ArrayList<Flight>`. Each `Flight` object stores a flight start time, a flight ending time, and a number of passengers. These values are all stored as `int`s.

The trick we use to store a flight start time (or end time) as an `int`, rather than as some sort of `Time` object, is to store the number of minutes that had elapsed in the Pacific Time Zone since midnight on January 1st, 1914, which was the first day of commercial air travel.

For example, a flight taking off at 2:02 PM on March 6th, 1917 and landing at 3:03 PM the same day carrying 30 passengers would have takeoff time 1,671,243, landing time 1,671,304, and number of passengers 30.

Give an algorithm for finding the largest number of people that have ever been in flight at once.

Your algorithm must run in N log N time, where N is the number of total commercial flights ever taken. Your algorithm must not have a runtime that is explicitly dependent on the number of minutes since January 1st, 1914, i.e. you can't just consider each minute since that day and count the number of passengers from each minute and return the max.

Your algorithm may use any data structures discussed in the course (e.g. arrays, `ArrayDeque`, `LinkedListDeque`, `ArrayList`, `LinkedList`, `WeightedQuickUnion`, `TreeMap`, `HashMap`, `TreeSet`, `HashSet`, `HeapMinPQ`, etc.)

a. List any data structures needed by your algorithm, including the type stored in the data structure (if applicable). If you use a data structure that requires a `compareTo` or `compare` method, describe **briefly** how the objects are compared. Do not include the provided `ArrayList<Flight>` in your list of data structures. Please list concrete implementations, not abstract data types.

I will use HeapMinPQ<Flight>, with the root being the max value
Therefore, I will modify the compareTo method, to let it compare the number of passengers; if a's is smaller than b's it will return a positive number, which is opposite to the normal comparison

No, I misunderstood the problem. But I am too lazy to figure out ...

b. Briefly describe your algorithm in plain English. Be as concise and clear as possible.

So I will traverse the ArrayList and insert all items into the HeapMinPQ, using the corresponding insertion operation. But I am not sure why the runtime complexity is Nlog N.