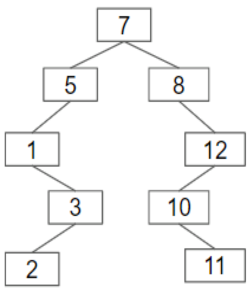
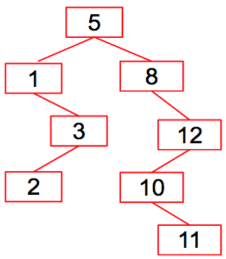
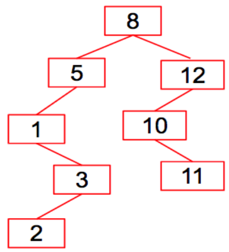
0. So it begins (1 point). Write your name and ID on the front page. Write the exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your login in the corner of every page. Enjoy your free point ☺.

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. Two possible answers; either is correct.





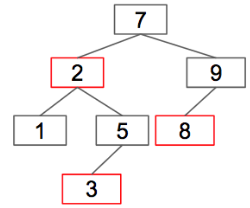


b) **(4 points).** Give an example of a rotation operation on the original tree from 1a (on the left) that would increase the height. **<u>You do not need to draw the tree</u>**, just write the operation, e.g. "rotateRight(11)".

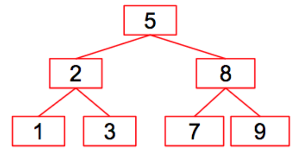
rotateLeft(7) or rotateRight(7)

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

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.



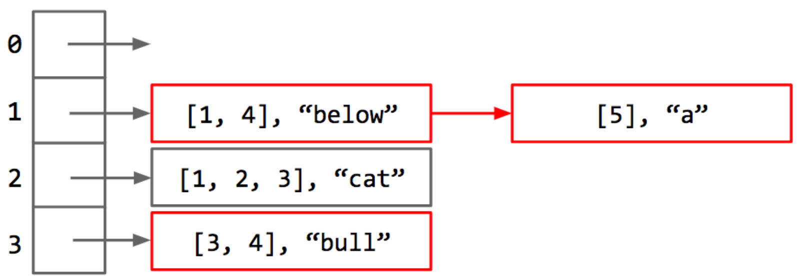
f) (6 points). Under what conditions is a **<u>complete</u>** BST containing N items **<u>unique</u>**? By "unique" we mean the BST is the only complete BST that contains exactly those N items. By "complete" we mean the same concept that was required for a tree to be considered a heap (precise definition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

N can be any value (that's non-negative of course).
We know that there is at least one complete BST containing a specific set of N items, let's call it T. We also know that there is only one way to arrange the N nodes to form a complete BST (e.g. a linear chain of N nodes is not a complete BST).
Now we show that the arrangement of values in T is unique. Suppose we take two different values x and y in T; without loss of generality, assume x < y (the same argument applies for y > x, and we know x != y because all items are unique). If we try to swap the places of x and y, we would obtain a tree where y is to the absolute left of x, which would violate the property of BST's where all items to the absolute left of a node are less than or equal to the item in that node.

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");
```



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);              __cat_______________
fbhm.get(XList.of(1, 2, 3));      __null______________
```


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.

`.get` best case: $\Theta(1)$
`.get` worst case: $\Theta(N)$
`.put` best case: $\Theta(1)$
`.put` worst case $\Theta(N)$

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: $\Theta(1)$
`.get` worst case: $\Theta(N)$
`.put` best case: $\Theta(1)$
`.put` worst case $\Theta(N)$

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 **6 items**. 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. **Give your answer as an exact value**. Assume Heighted Quick Union is like Weighted Quick Union, except uses height instead of weight to determine which subtree is the new root.

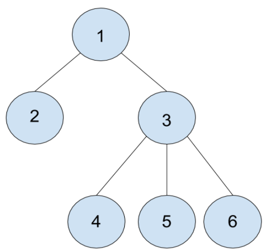
	Best Case Height	Worst Case Height
Weighted Quick Union	1	2
Heighted Quick Union	1	2

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 that would be **possible for Heighted Quick Union**, but **impossible** for Weighted Quick Union. If no such tree exists, simply write "none exists."

Example tree shown below. In general, the heights of the two input trees should be the same, but the heavier tree should be attached to the lighter one.



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) {
      union(item.hashCode(), 140608);
  }
  public boolean contains(T item) {
      return connected(item.hashCode(), 140608);
  }
} // reminder: WeightedQuickUnionUF methods are union() and connected()
```


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) **(10 points).** A 61B student suggests that **one way to implement `Multiset` is to modify a BST** 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;
    } ...
```

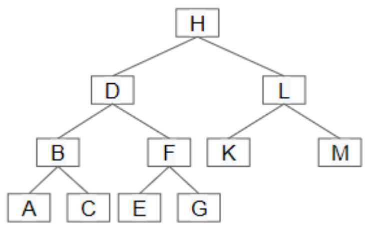
b) **(6 points).** Let X be an item with multiplicity M, and let N be the number of nodes in the tree. Give an Omega bound for the best case runtime of any possible implementation of `multiplicity(X)` for a `TriSTMultiset`. Give the best possible bound you can.

$\Omega(M)$

c) **(6 points).** Rather than building an entirely new data structure from scratch, we might consider implementing `Multiset` using delegation or extension with an existing data structure from the `java.util` library. Which is the better choice: delegation or extension? If delegation, what class should you delegate to? If extension, what class should you extend? **If applicable, provide generic types.** Fill in **one of the bubbles** and the corresponding blank below. There may be multiple reasonable answers.

○ Delegation to an instance of the `java.util.`TreeMap<T, Integer> class is better.
○ Extending the `java.util.`________________________________ class is better.

6. Min Heaps (14 points). Consider the min heap below, where each letter ***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 unique.**



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) **(2 points).** 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) **(6 points).** Assuming values are inserted into the heap in an **unknown order**, indicate all letters which could represent the following values:

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

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 for-each notation (e.g. `for (x : blah)`). Do not use `.next` and `.hasNext` explicitly.

```
public class IterableUtils {
    public static <T> List<T> toList(Iterable<T> iterable) {
        List<T> r = new ArrayList<T>();

        for (T t: iterable) {
            if (t == null) {
                throw new IllegalArgumentException();
            }
            r.add(t);
        }
        return r;
    }
} // 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 part a. If you did not complete part a, you can still do this problem. 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 this;
    }
} // assume any classes you need from java.util have been imported
```

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 one if its dependencies failing to compile.

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

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

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

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

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>**!

_$\Theta(N^6)$____
```
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");
        }
    }
}
```

_$\Theta(2^N)$_____
```
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");
        }
    }
}
```

_ $\Theta(N)$_____
```
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) **(5 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. **Don't spend too much time on these**! 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: _$\Theta(N)$_____
Worst case: _$\Theta(2^N)$______

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

Best case: _$\Theta(\log N)$_______
Worst case: _$\Theta(N)$_______

e) **(6 points).** Give the best and worst case runtime of the code below in terms of N, the length of x. Assume `HashSet61Bs` are implemented exactly like hash tables from class, where we used external chaining to resolve collisions, and resize when the load factor becomes too large. Assume `resize()` is implemented without any sort of traversal of the linked lists that make up the external chains.

```
public Set<Planet> uniques(ArrayList<Planet> x) {
    HashSet61B<Planet> items = new HashSet61B<>();
    for (int i = 0; i < x.size(); i += 1) {
        items.add(x.get(i));
    }
    return items;
}
```

Best case runtime for `uniques`: _ $\Theta(N)$__ Worst case runtime for `uniques`: _ $\Theta(N^2)$ _

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 of length N. Give the best and worst case runtime.

Best case runtime for `uniques`: _ $\Theta(N)$___ Worst case runtime for `uniques`: _ $\Theta(N^3)$__