Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

A Binary Search Type

- Here, we'll use the following simple binary search tree type. Ignore all the style violations, please.

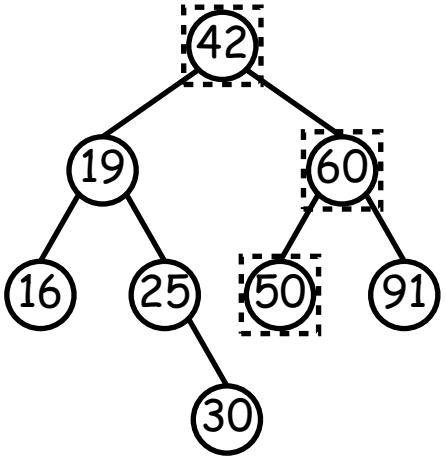
```
/** A node of a binary search tree associating a value of type VALUE
 *  with a key of type KEY.  (Thus, the labels in this tree are
 *  key/value pairs.) */
class BST<Key extends Comparable<Key>, Value> {
    Key key;
    Value value;
    BST<Key, Value> left, right;

    BST(Key key0, Value value0,
        BST<Key, Value> left0, BST<Key, Value> right0) {
        Body left to the reader.
    }
    BST(Key key0, Value value0) {
        this(key0, value0, null, null);
    }
}
```

- (Ignore the `Key extends Comparable<Key>` stuff for now. It just says that keys (of type `Key`) can be compared to each other.)

Finding

- Searching for 50 and 49:

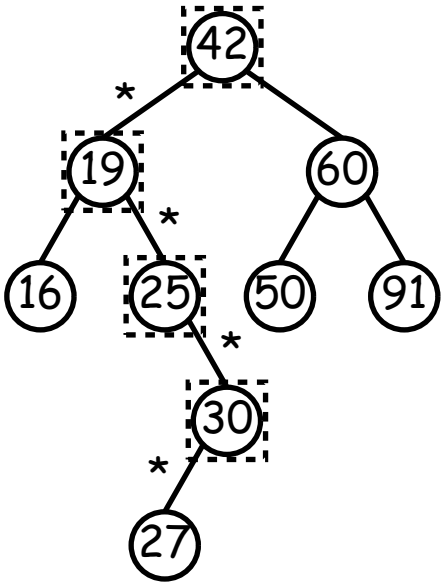



```
/** Return node in T containing L.  Null if none. */
static <Key extends Comparable<Key>, Value>
  BST<Key, Value> find(BST<Key, Value> T, Key L) {
    if (T == null)
        return T;
    if (L.compareTo(T.key) == 0)
        return T;
    else if (L.compareTo(T.key) < 0)
        return find(T.left, L);
    else
        return find(T.right, L);
}
```

- Dashed boxes show which node labels we look at.
- Number of nodes examined is proportional to height of tree.

Inserting

- Inserting 27




```
/** Insert V in T with key K, replacing existing
 *  value if present. Return the modified tree. */
static <Key extends Comparable<Key>, Value>
  BST<Key, Value> insert(BST<Key, Value> T,
                         Key K, Value V) {
    if (T == null)
        return new BST(K, V);
    if (K.compareTo(T.key) == 0)
        T.value = V;
    else if (K.compareTo(T.key) < 0)
        T.left = insert(T.left, K, V);
    else
        T.right = insert(T.right, K, V);
    return T;
}
```

- Starred edges are set (to themselves, unless initially null).
- Again, time proportional to height.

Deletion Algorithm

```
/** Remove K from T, and return the new tree. */
static <Key extends Comparable<Key>, Value>
  BST<Key, Value> remove(BST T, Key K) {

    if (T == null)
        return null;
    if (K.compareTo(T.key) == 0) {
        if (T.left == null)
            return T.right;
        else if (T.right == null)
            return T.left;
        else {
            BST<Key, Value> smallest = minNode(T.right);  // ??
            T.value = smallest.value;
            T.key = smallest.key;
            T.right = remove(T.right, smallest.key);
        }
    }
    else if (K.compareTo(T.key) < 0)
        T.left = remove(T.left, K);
    else
        T.right = remove(T.right, K);
    return T;
}
```

Minimum Node in a Subtree

```
/** Return the node containing the minimal key T.
 *  T must not be null. */
public static <Key extends Comparable<Key>, Value>
    BST<Key, Value> minNode(BST<Key, Value> T) {

    while (T.left != null) {
        T = T.left;
    }

    return T;
}
```

- This is a *generic function*. As before, let's ignore all the "`<Key extends Comparable...>`" stuff for now.

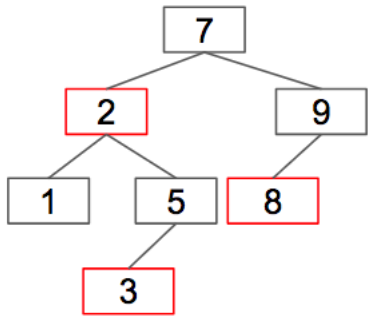
Some Pseudocode for Searching (Maximizing Player)

```
/** The estimated minimax value of position POSN, searching up to
 *  DEPTH moves ahead, assuming it is the maximizing player's move.
 *  If the value is determined to be <=ALPHA, then the function
 *  may return any value <=ALPHA, even if inaccurate.  Likewise if the
 *  value is >=BETA, it may return any value >=BETA.  Assumes ALPHA<BETA. */
int maxPlayerValue(Position posn, int depth, int alpha, int beta)
{
    if (posn is a final position of the game || depth == 0)
        return staticGuess(posn);
    int bestSoFar = -∞;
    for (each legal move, M, in position posn) {
        Position next = makeMove(posn, M);
        int response = minPlayerValue(next, depth-1, alpha, beta);
        if (response > bestSoFar) {
            bestSoFar = response;
            alpha = max(alpha, bestSoFar);
            if (alpha >= beta)
                return bestSoFar;
        }
    }
    return bestSoFar;
}
```

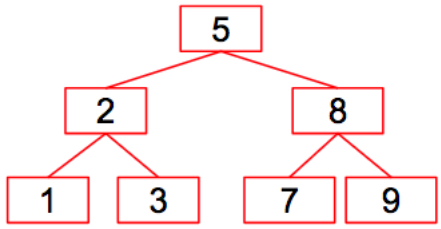
Some Pseudocode for Searching (Minimizing Player)

```
/** The estimated minimax value of position POSN, searching up to
 *  DEPTH moves ahead, assuming it is the minimizing player's move. */
int minPlayerValue(Position posn, int depth, int alpha, int beta)
{
    if (posn is a final position of the game || depth == 0)
        return staticGuess(posn);
    int bestSoFar = +∞;
    for (each legal move, M, in position posn) {
        Position next = makeMove(posn, M);
        int response = maxPlayerValue(next, depth-1, alpha, beta);
        if (response < bestSoFar) {
            bestSoFar = response;
            beta = min(beta, bestSoFar);
            if (alpha >= beta)
                return bestSoFar;
        }
    }
    return bestSoFar;
}
```

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.



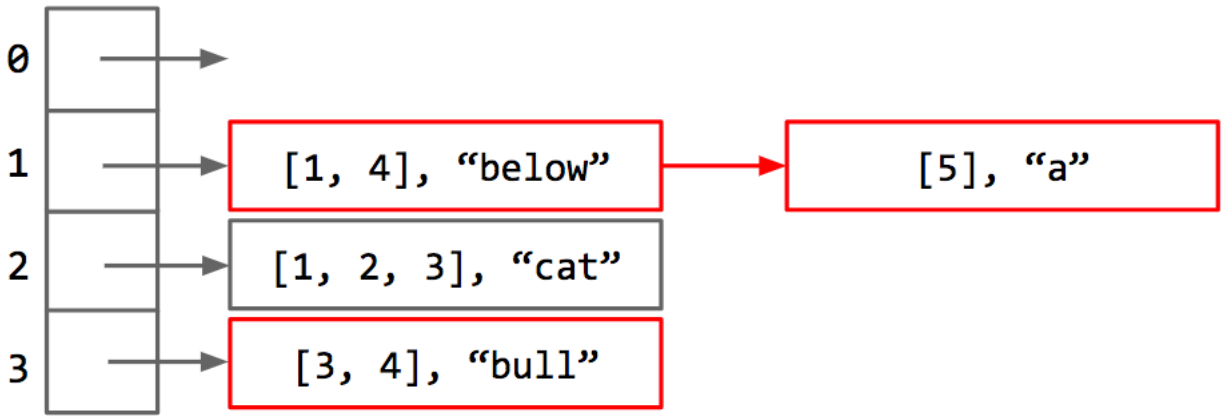
f) (6 points). Under what conditions is a **complete** BST containing N items **unique**? By "unique" we mean the BST is the only complete BST that contains exactly those N items. By "complete" we mean the same concept that was required for a tree to be considered a heap (precise definition not repeated 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: 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 **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.

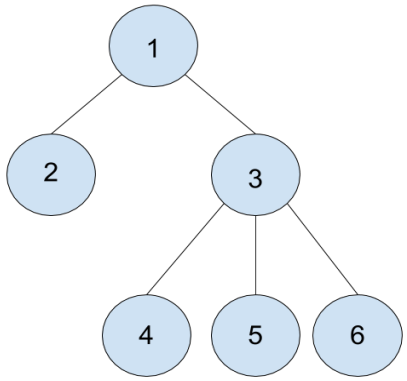
	Best Case Height	Worst Case Height
Weighted Quick Union	1	2
Heighted Quick Union	1	2

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 <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;
    } ...
```

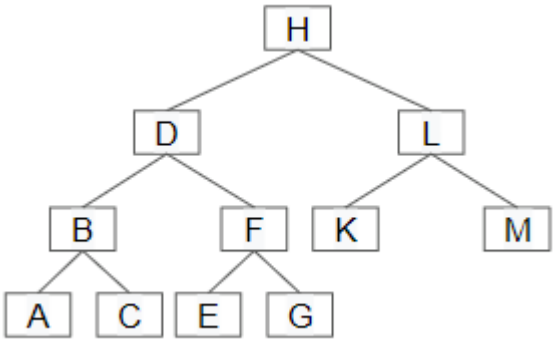
b) **(6 points).** Let X be an item with multiplicity M, and let N be the number of nodes in the tree. Give an Omega bound for the best case runtime of any possible implementation of `multiplicity(X)` for a `TriSTMultiset`. Give the best possible bound you can.

Ω(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. **Don't spend too much time on these**!

_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 Θ(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
Ω(N^2 log N)	○ True	○ False	○ NEI
O(N^3)	○ True	○ False	○ NEI
Θ(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 Θ(N^2)	○ True	○ False	○ NEI
There exists some inputs for which the runtime is Θ(N^3)	○ True	○ False	○ NEI
There exists some inputs for which the runtime is Θ(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: _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 `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`: _ 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 of 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`, `QuadTree`, 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.

Canonically, use a HeapMinPQ<Flight> sorted by start times, and HeapMinPQ<Flight> sorted by end times. Any data structure that can provide an N log N ordered retrieval will work.

Alternatively, any unordered data structure that can be sorted in N log N and then retrieved in O(N log N) time will work. This includes sorting the input ArrayList<Flight> (in which case only one additional data structure was needed).

A note on BSTs and BST-variants: due to the clarification that flights can have duplicate start/end times, any implementation using data structures that did not support duplicates required additional handling of duplicate cases (partial credit was given if not done).

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

First, insert all flights into the two PQs. Set the current tally to zero. Peek at the top of both PQs. Remove the smaller one. If it came from the start PQ, then add the number of passengers to the tally. If the tally is larger than it has ever been, record that as best. If it came from the end PQ, subtract from the tally. Return best.

Here's an example of a sorted input and PQ alternate solution. Sort the input ArrayList by start time using an N log N sort. Initialize a PQ sorted by end time, and leave it empty. Then, iterate through the sorted input list, adding flights into the PQ when seen by the input list iterator, and removing from the PQ in a similar manner to above. Again, track current tally and best.