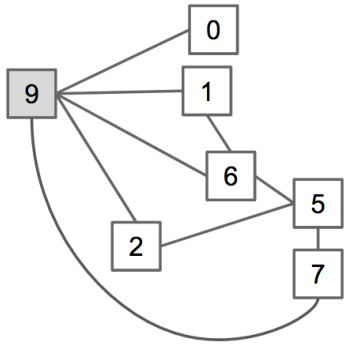
1. Traversin (11.5 points).

a) (6 pts) For the graph below, give the DFS preorder, postorder, and BFS order traversals starting from **vertex 9**. The BFS order is the order in which vertices are enqueued. Assume ties are broken in numerical order (i.e. the edge 15 → 16 would be considered before 15 → 17).



DFS Preorder: 9, 0, 1, 6, 5, 2, 7

DFS Postorder: 0, 2, 7, 5, 6, 1, 9

BFS Order: 9, 0, 1, 2, 6, 7, 5

b) (4 pts) Suppose we have a min heap of 7 **unique** items and we want to print the values of the heap **in increasing order**. For which of our standard tree traversals will we get the values of the heap in increasing order if we print when we visit a node? **Fill in the bubbles completely.**

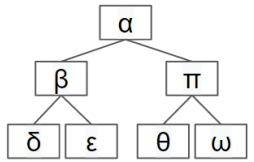
Preorder	○ Never	● Sometimes	○ Always
Inorder	● Never	○ Sometimes	○ Always
Postorder	● Never	○ Sometimes	○ Always
Level order	○ Never	● Sometimes	○ Always

c) (1.5 pts) Draw a tree with 5 nodes for which the preorder traversal is the reverse of the inorder traversal, and for which all values are unique. Or if this is not possible, simply write "Impossible".

Any tree that where all nodes only have left children is fine.

2. An Operational Understanding (15.2 points).

a. Consider the tree on the left where greek letters represent numerical values. In the boxes to the right, shade all values that might match the text. **Assume all values are unique.** For BSTs, assume left items are less than. When treating the tree like a graph, assume nothing about the order of adjacency lists.

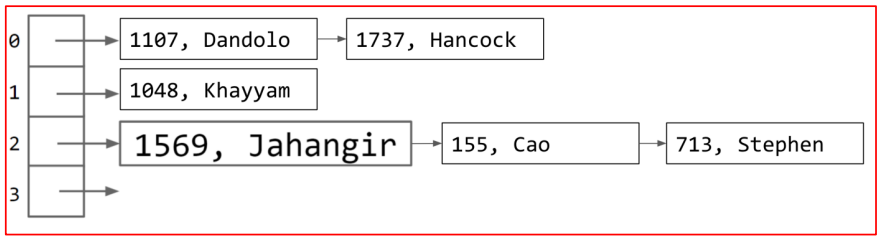


Fill in the boxes completely.

	α	β	π	δ	ε	θ	ω	φ
MinHeap, largest item	□	□	□	■	■	■	■	
MinHeap, smallest item	■	□	□	□	□	□	□	
BST, largest item	□	□	□	□	□	□	■	
BST, smallest item	□	□	□	■	□	□	□	
MinHeap, median item	□	■	■	■	■	■	■	
BST, median item	■	□	□	□	□	□	□	
MinHeap, new root after deleteMin	□	■	■	□	□	□	□	
BST, new root after Hibbard[1] deletion of α	□	□	□	□	■	■	□	
MinHeap, root after inserting new item φ	■	□	□	□	□	□	□	■
BST, root after inserting new item φ	■	□	□	□	□	□	□	□
Last item dequeued running BFS from ω	□	□	□	■	■	□	□	

b. Suppose we have an initially empty hash map (as discussed in lecture) that maps a year to a famous person born that year. Suppose that the hash code of the year is given by the sum of the first and last digits, e.g. hashCode(1569) would be 10. Draw the hash table after calling **put** with the following key/value pairs. Assume that each bucket is a list, and that there is no resizing. The first one has been completed for you. Assume new items are placed at the end of the list.

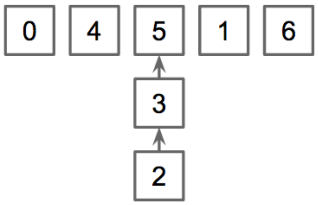
K/V pairs: [1569 / Jahangir], [155 / Cao], [1107 / Dandolo], [1737 / Paine], [713 / Stephen], [1048 / Khayyam], [1737 / Hancock]



[1] Hibbard deletion is the deletion technique from lecture where we arbitrarily take one of two values to be the new root.

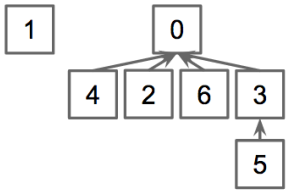
3. Weighted Quick Union (12 points)

a. (2 pts) Suppose we have a **weighted quick union** object. What calls to `connect(a, b)` produce the following trees? Assume that each **WQU** starts with all items disconnected. Or fill in the "Impossible" option if the given tree is impossible. Assume that in case of a tie, the root of the left argument is placed below the root of the right argument.



`connect(_____, _____)`
`connect(_____, _____)`

● Impossible



`connect(2, 0)`
`connect(4, 0)`
`connect(5, 3)`
`connect(3, 0)`
`connect(6, 0)`

○ Impossible

b. (4 pts) Suppose we add a new operation `undo(a, b)` that undoes an earlier Disjoint Sets `connect` operation. If `connect(a, b)` has never been called, then this method has no effect. For each of the implementations of Disjoint Sets, **mark** the corresponding box **if it is impossible** to add the `undo` operation without adding additional data structures (i.e. instance variables) to that implementation.

■Quick Union ■Quick Find ■Weighted Quick Union(WQU) ■WQU with Path Compression

c. (6 pts) Suppose we use a **graph** instead of a quick union tree to solve the disjoint sets problem. Assume we implement the `connect(a, b)` and `isConnected(a, b)` operations using a graph represented as an adjacency list with no duplicates allowed. Assume that we implement `connect` using `addEdge`, and `isConnected` using DFS with a marked array, where DFS terminates early if a connection is detected. Let N be the number of nodes in the graph, and let M be the total number of calls to either the `connect` or `isConnected` methods. For example, if we call `connect` 37 times, then `isConnected` 13 times, then `connect` 20 times, then M=70.

What will be the **worst case** runtime of any single call to `connect(a, b)`?

○Θ(1)	○Θ(log N)	○Θ(log M)	●Θ(N)	●Θ(M)	●Θ(N + M)	○Θ(NM)	○Θ(N + M log* M)

What will be the **worst case** runtime of any single call to `isConnected(a, b)`?

○Θ(1)	○Θ(log N)	○Θ(log M)	●Θ(N)	●Θ(M)	●Θ(N + M)	○Θ(NM)	○Θ(N + M log* M)

Explanation: We accepted all three answers above as reasonable for both problems because there was some ambiguity about how you might interpret "the worst case".

The first question essentially boiled down to "If we add M edges to a graph with N vertices, what is the worst case runtime for addEdge?" and the second question is essentially just "If we have a graph with N vertices and M edges, what is the worst case runtime for DFS?"

While these might seem straightforward, consider the following question: We know that DFS is Θ(V+E) in the worst case, but is it also Θ(V) in the worst case? Certainly as V grows, the size of our marked array will also grow, so a plot of runtime vs. V will indeed be linear. Since we haven't discussed such ambiguities in class and this can be quite confusing, we opted to accept any of these three answers in the spirit of this ambiguity.

4. Asymptotics (20 points).

a) (7 pts) For each code block below, fill in the blank(s) so that the function has the desired runtime. Do not use any commas. If the answer is impossible, just write "impossible" in the blank.

```
public static void f1(int N) {      // desired runtime: Θ(N)
  for (int i = 1; i < N; i += 1) { System.out.println("hi"); }
}

public static void f2(int N) {      // desired runtime Θ(log N)
  for (int i = 1; i < N; i *= 2) { System.out.println("hi"); }
}

public static void f3(int N) {      // desired runtime Θ(1)
  for (int i = 1; i < N; i += N) { System.out.println("hi"); }
}
```

b) (8 pts) Give the runtime of the following functions in Θ or O notation as requested. Your answer should be as simple as possible with no unnecessary leading constants or lower order terms. For f5, your bound should be as tight as possible (so don't just put $O(N^{NM!})$ or similar for the second answer). **Don't spend too much time on these**!

$\Theta(N^2 \log N)$

```
public static void f4(int N) {
   if (N == 0) { return; }
   f4(N / 2);
   f4(N / 2);
   f4(N / 2);
   f4(N / 2);
   g(N); // runs in Θ(N^2) time
}
```

$O(N)$

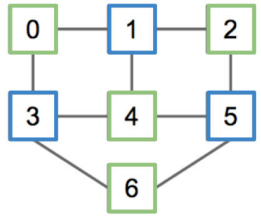
```
public static void f5(int N, int M) {
   if (N < 10) { return; }
   for (int i = 0; i <= N % 10; i++) {
      f5(N / 10, M / 10);
      System.out.println(M);
   }
}
```

c) (0 pts) This mostly subterranean building, designed by I.M. Pei, cost more than $250,000,000 to construct, and was built by a small religious group that believes that by "building architectural masterpieces in remote locations, they are restoring the Earth's balance".

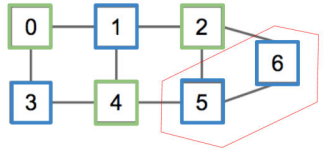
d) (5 pts) Suppose we write a method to assign careers to a list of puppies, defined below:

6. Bipartite Graphs (12.5 points).

a) (3 pts) Suppose we want to color every vertex of a graph either blue or green such that no vertex touches anther vertex of the same color. This is possible for some graphs but not others. A graph where a valid coloring exists is called "bipartite". Which of the graphs below are bipartite?



● Bipartite ○ Not Bipartite



○ Bipartite ● Not Bipartite

Explanation: This problem was to help build an intuition of how the two color algorithm works. The algorithm works as follows. Start with a node and perform a DFS traversal. For each vertex you visit, alternate between two colors. If you never color a vertex such that it shares a color with its neighbor, then the graph is bipartite!

b) (9.5 pts) Suppose we are using the undirected Graph API from the lecture / optional textbook, shown below.

```
public class Graph {
  public Graph(int V):                Create empty graph with v vertices
  public void addEdge(int v, int w): add an edge v-w
  Iterable<Integer> adj(int v):       vertices adjacent to v
  int V():                            number of vertices
  int E():                            number of edges
  ...
}
```

Fill in the method `twocolor` below such that a correct assignment to the blue vertices is printed out when the code runs, or if no such assignment is possible, an exception is thrown. Write only one statement per line (note that the for loop counts as one statement by the rules of the exam on page 1).

Explanation: The trick in this problem was to use recursion that treated one of the arguments (either that in position a or b) to be the specially designated set being assigned to during that recursive call. This means that when you recurse, you have to switch the positions of a and b! Solution code on next page:

```
HashSet<Integer> blue = new HashSet<Integer>();
HashSet<Integer> green = new HashSet<Integer>();
twocolor(G, 0, blue, green);
System.out.println("Blue vertices are: " + blue.toString());

public static void twocolor(Graph G, int v, Set<Integer> a, Set<Integer> b){
    a.add(v);
    for (int u : G.adj(v)) {
        if (a.contains(u)) {
            throw new IllegalArgumentException("graph is not bipartite"); }
        if (!b.contains(u)) {
            twocolor(G, u, b, a);
        }
    }
}
```


7. Trees and Hashing (18 points).

a. (4.5 pts) Suppose we implement a `LLRBBucketHashSet` where the hash table buckets are stored as left leaning red black binary search trees. Assume we resize by doubling the number of buckets whenever the load factor L exceeds 2, and that we never decrease the number of buckets. Assume that hashCode computation is constant time. **Do not assume** that the hash code nicely spreads out items! If there are currently N items and M buckets in the hash table, fill in the runtimes for each operation for a **single call** in the table below (i.e. the first box is the best case for **one `put` call**). Give your answer in Θ notation in terms of N and M. You may not need both N and M. Not all of these facts may be relevant.

Special note: A small number of students appeared to have interpreted this problem as meaning that we had an LLRB of buckets. That is, rather than an array of LLRBs, they had an LLRB of lists (or even an LLRB of LLRBs). If you are one of these students, and you feel like your answers are correct, please do submit a regrade. However, be aware that you interpretation of the problem is a lot harder than the one on the actual exam, so please only submit a regrade if you think you actually got the right answer. Best cases are still Theta(1) and worst cases involve Theta(Log M + …).

`Put`		`containsKey`		`remove`	
Best case	Worst case	Best case	Worst case	Best case	Worst case
Θ(1)	Θ(N log N)	Θ(1)	Θ(log N)	Θ(1)	Θ(log N)

b. (4.5 pts) Fill in the runtimes below for a **single operation** for an `LLRBBucketNoResizeHashSet`, which is the same as in part a, except that the number of buckets is never increased. Give your answer in terms of N and M. You may not need to use both of these parameters. Not all facts may be relevant.

`put`		`containsKey`		`remove`	
Best case	Worst case	Best case	Worst case	Best case	Worst case
Θ(1)	Θ(log N)	Θ(1)	Θ(log N)	Θ(1)	Θ(log N)

c. (2 pts) If we have an `LLRBBucketHashSet` that is **initially empty** and we perform Q insertions, what is the amortized (i.e. average) runtime for a **single call** to `put` **assuming that our hash code spreads items nicely across the buckets**? Give your answer in the blank below in terms of Q and M. You may not need to use both of these parameters. Not all facts may be relevant.

Amortized time per `put` call after Q calls: O(1)

d. (2 pts) Same question as part c, but for an `LLRBBucketNoResizeHashSet`.

Amortized time per `put` call after Q calls: O(log (Q / M))

e. (3 pts) Is an `LLRBBucketHashSet` significantly worse, about the same, or significantly better than a standard HashSet that uses a linked list for buckets? Explain your answer.

LLRBBucketHashSets are: ● Significantly Worse ● About the same ● Significantly Better

Explanation: All of the answers were correct, what matters for this problem is your explanation and assumptions.

Significantly Better:
In the case that the hashcode was bad, leading to worst case performance in an ordinary HashSet (many collisions), the logN performance boost given by the LLRB tree would significantly improve over the linear performance of a linked list.

About the same:
This case assumed that the hashcode was good. Since with a good hashcode, the number of items in a bucket is about L (the load factor), which is constant. A linked list would yield O(L) performance while an LLRB tree would yield O(log(L)) performance. But since L is constant, both of these would have O(1) asymptotic performance.

Significantly Worse:
The case also assumed that the hashcode was good, so the runtime of both data structures was O(1). But it recognized that the constant factor needed to keep an LLRB tree in order is much higher than the simple operations being performed on a linked list, and therefore the overhead of performing LLRB balancing would hurt performance.

f. (2 pts) Suppose we wanted to implement `LLRBBucketHashSet` by using the `RedBlackBST.java` file provided in our optional textbook. Would it be more appropriate to use an extension based approach (i.e. extending `RedBlackBST`), a delegation approach (having instance variables that include a `RedBlackBST`), or either one?

It'd be better to use: ○ Extension ● Delegation ○ Either is appropriate

8. Xelha (25 points).

Consider the method defined below which generates a `XelhaTree` from a list of numbers.

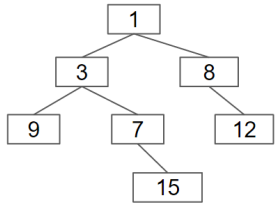
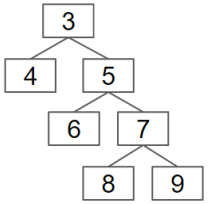
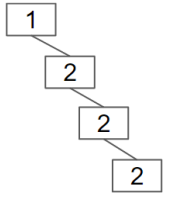
```
public IntTree generateXelhaTree(List<Integer> X)
```

Given a list of numbers X, a `XelhaTree` for that list obeys the following:

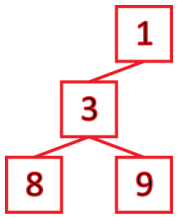
1. The `XelhaTree` has the min-heap property (i.e. every value is less than or equal to its children).
2. An inorder traversal of the `XelhaTree` visits the nodes in the same order as the list.

For example, given the list [9, 3, 7, 15, 1, 8, 12], the corresponding XelhaTree is as shown below in part a. This tree has the min-heap property, and an in-order traversal of this tree visits the vertices in the order 9, 3, 7, 15, 1, 8, 12. A XelhaTree does not need to be complete. XelhaTrees allow duplicate items.

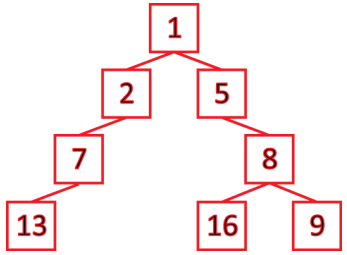
a. (2 pts) Which of the following are valid XelhaTrees for the given sequences? The first is done for you.

		
[9, 3, 7, 15, 1, 8, 12] Valid: ●, Invalid: ○	[4, 3, 6, 5, 8, 7, 9] Valid: ●, Invalid: ○	[1, 2, 2, 2] Valid: ●, Invalid: ○

b. (3 pts) Draw a valid XelhaTree corresponding to the sequence [8, 3, 9, 1].



c. (5 pts) Draw a valid XelhaTree corresponding to the sequence [13, 7, 2, 1, 5, 16, 8, 9]. **Don't spend too much time on this if you're stuck!** Go back to an earlier problem and come back later.



d. (15 pts) Describe an algorithm in English for building a XelhaTree (i.e. `createXelhaTree`). **Your answer will be graded on correctness, efficiency, and clarity.** To keep things organized, you might consider using a numbered list of steps as all or part of your answer. If you didn't figure out c on the previous page, there's probably no point in working on this one.

For **full credit**, your algorithm must take **less than $\Theta(N^2)$** time in the worst case, though **partial credit** will be given for algorithms that complete in **$\Theta(N^2)$ time** in the worst case.

We will provide two common $\Theta(N^2)$ solutions and the non-trivial $\Theta(N)$ solution. The solution we saw most often was this recursive $\Theta(N^2)$ algorithm:

```
createXelhaTree(List L):
    Let index i be the index of the smallest element in L
    Create a TreeNode with value L[i]
    Set the left child of this node to createXelhaTree(L[0:i])
    Set the right child of this node to createXelhaTree(L[i+1:end])
    Return this node.
```

An alternative O(N^2) solution we saw fairly frequently was the following:

```
createXelhaTree(List L):
    Sort L, remembering the original index of each value.
    Make a new tree T, initially empty
    For item i in L:
        Insert i into L.
    Return T
```

Here Insertion is always done at a leaf node and choosing to go left or right at a node when inserting is done by comparing the original indices of the two values.
Though $\Theta(N\log N)$ solutions exist, in most cases a $\Theta(N\log N)$ solution was the $\Theta(N)$ solution made slightly more complicated. We present only the following $\Theta(N)$ solution.

First define a helper function `buildSubtree(List L, int min)` which takes in an inorder list of items L (imagine they've already been placed into TreeNodes) and min the smallest value allowed from L in the subtree constructed in the current call frame.

```
buildSubtree(List L, int min):
    Let current and root be two TreeNode pointers initialized to null.
    While L is not empty:
        if (L[0] is smaller than min):
            return root
        else if (root is null or L[0] is smaller than the root):
            Remove L[0] and assign it to current.
            Assign left child of current to the old root
            Assign root to current
        else if (L[0] is larger than the root):
            Set right child of root to result of buildSubtree(L, root)
    return the root
```

The entire tree can be created in $\Theta(N)$ time by calling `buildSubtree(L, -infinity)`. There is an iterative variant of this method that works in the following way (plus some base cases and null checks). It is worth noting that this requires the tree to have parent pointers, whereas the recursive solution does not.

```
createXelhaTree(List L):
   Define two TreeNode pointers: root, last_inserted
   For i=0 to L.length - 1:
       if L[i] is smaller than root:
           Assign root to L[i]
           Assign L[i]'s left child point to the old root
       Else:
           While L[i] is smaller than last_inserted:
               Reassign last_inserted to parent of old last_inserted
           Assign left child of L[i] to be right child of last_inserted
           Assign the right child of last_inserted to be L[i]
       Assign last_inserted to L[i]
   Return root
```

Note that after the first assign in the else block L[i] will be greater than last_inserted.

End of Exam Decompression Space