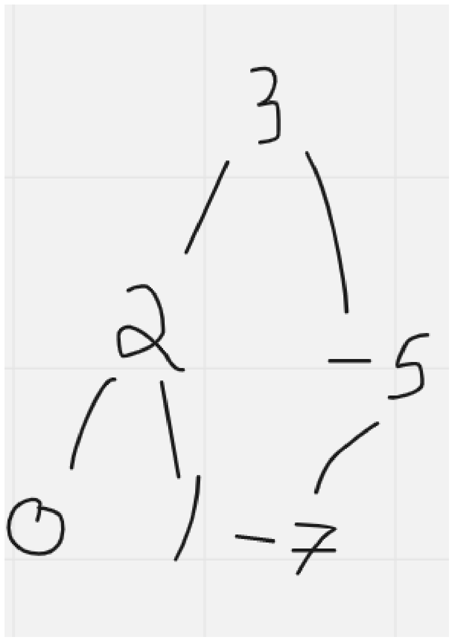
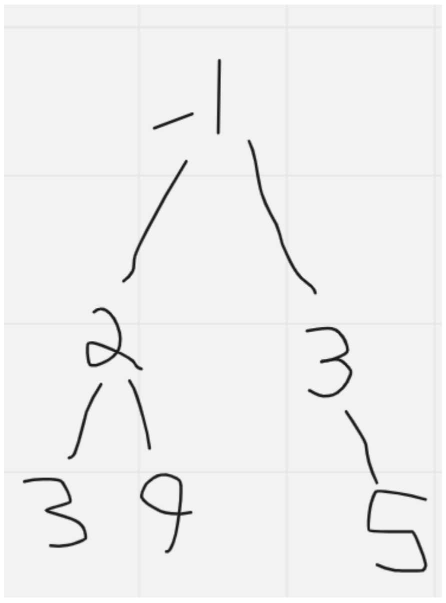
Heaps Introduction

Heaps are representations of complete binary trees with one of the following conditions met:

1. For all nodes *N*, the largest value of the subtree tree is at *N(Max-Heap)*
2. For all nodes *N*, the smallest value in the subtree is at *N*(*Min-Heap*)

 Max-Heap	 *Very close to a min-heap*, but notice it's not complete! The gap is not in the lower right corner.

This can be useful because an extreme value of the tree is easily available.

- In an emergency room, each node in a heap could be used to represent a patient. A patient with the most severe/urgent damage would be at the root of the heap. After that patient is treated, the heap would reconfigure to put the next most urgent patient at the root.

Heaps Stored in Arrays

Since the tree is complete, there are no gaps in the binary tree until the very "end" of the tree.
The nodes are stored into an array through a breadth-first search order (level order).
In the above example of a max heap, the heap would be stored as the array:
[0, 3, 2, -5, 0, 1, -7]
The zero index is *not* used and a placeholder value (like 0 or null) is put.

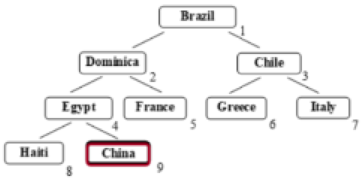
Notice that the children of a node at index *i* are at $2i \text{ and } 2i + 1$
Reversing that logic, the parent of a node at index *i* occurs at $\lfloor \frac{i}{2} \rfloor$.

Adding to a Heap

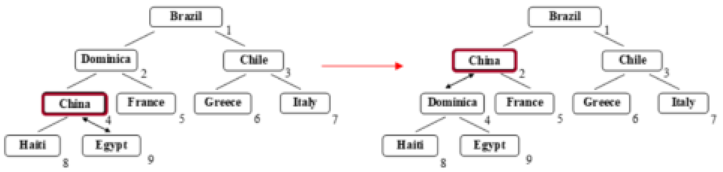
- The algorithm for **add** uses the *reheap-up* procedure.

Add a leaf. Starting at the last leaf, swap the node with its parent as many times as needed to repair the heap.

Step 1: the new value is added as the rightmost leaf at the bottom level, keeping the tree complete



Step 2: "reheap up": the new value keeps swapping places with its parent until it falls into place



*The numbers are the indices of the array (the tree is for visual purposes)

This heap considers the min to be the String starting with A to be the least element, etc*

Removing from a Heap

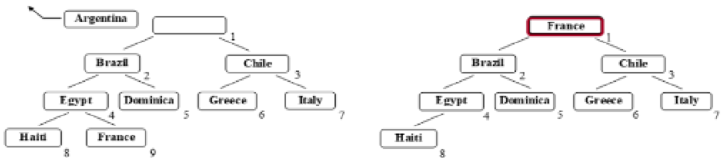
The algorithm for **remove** uses the *reheap-down* procedure.

Remove the root and place the last leaf at the root. Starting at the root, swap the node with its smaller child, as many times as needed to repair the heap.

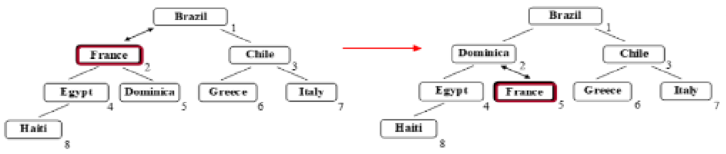
Remove in a heap will remove the root (index 1)

Step 1: the root is removed

Step 2: the rightmost leaf from the bottom level replaces the root



Step 3: "reheap down": the new root value keeps swapping places with its smaller child until it falls into place



* We still have a complete binary tree at the end*

HeapSort

Improved selection sort. Time complexity is O(nlog(n)), which is similar to merge sort - merge sort does take more memory than heap sort, though.

Using a max-heap will sort the objects in ascending order.

Using a min-heap will sort the objets in descending order.

The algorithm is presented for a max-heap sort, which turns an array that represents a max-heap into an array sorted in ascending order:

Original Heap				
5	3	4	1	2
Each Iteration at Step 3				
4	3	2	1	5
3	1	2	4	5
2	1	3	4	5
1	2	3	4	5

It's now all sorted!

Yellow is the section considered a heap.
Green is the sorted section.

1. Swap the first element in the heap with the last element of the heap
2. Decrease the size of the heap by 1
3. Re-heap down (swapDown)
4. Repeat until the heap is of size 1

The algorithm is presented for a min-heap sort, which turns an array that represents a min-heap into an array sorted in ascending order:

Original Heap				
-3	0	9	5	7
Each Iteration at Step 3				
0	5	9	7	-3
5	7	9	0	-3
7	9	5	0	-3
9	7	5	0	-3

It's now all sorted!
Yellow is the section considered a heap.

1. Swap the first element in the heap with the last element of the heap
2. Decrease the size of the heap by 1
3. Re-heap down (swapDown)
4. Repeat until the heap is of size 1

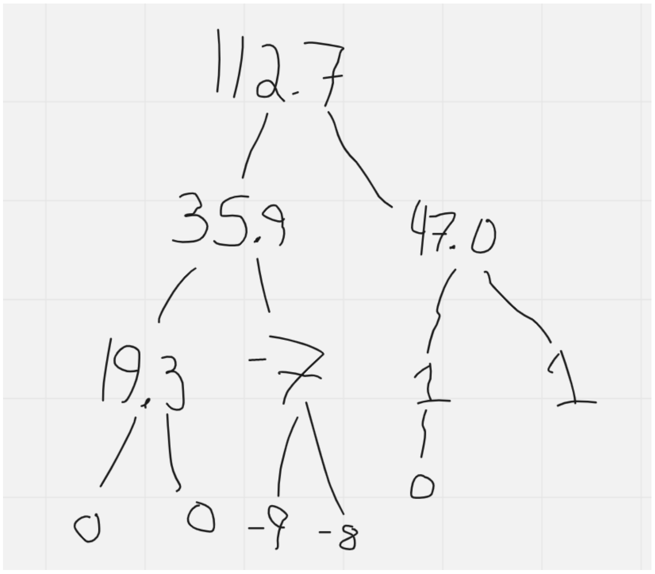
Yes, it's the same algorithm!

Green is the sorted section.	

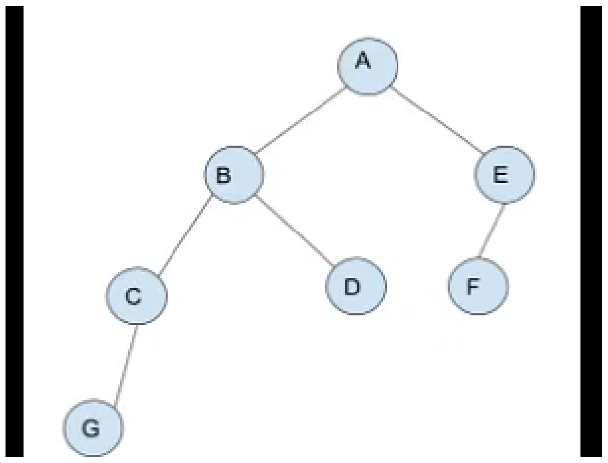
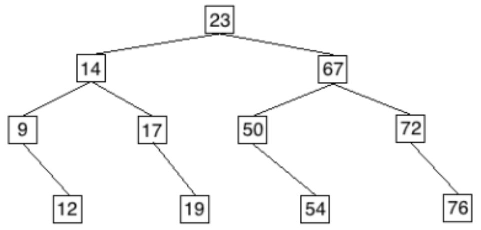
Pratice Problems for Trees

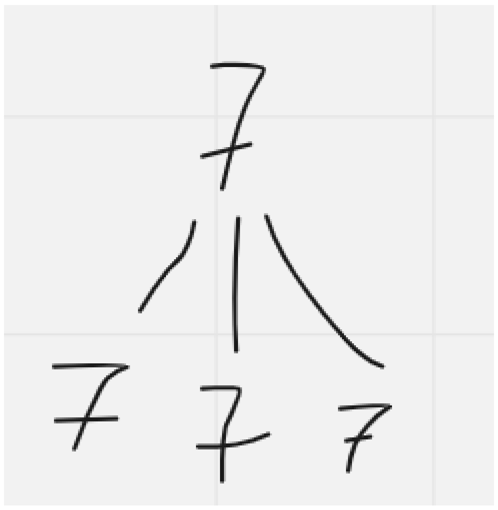
1. Convert the following array into a drawing of a heap, and identify if it is a min heap or max heap, or not a heap at all.
    a. [null, “Alice”, “Bob”, “Billy”, “Ron”, “Carlie”, “iCarly”, “Zumadahl”]

2. Remove from the following heap and redraw it.



3. Of the following three data structures, name them as specifically as possible (using subtypes of heaps and binary trees). If they fit multiple descriptions, name all of them:

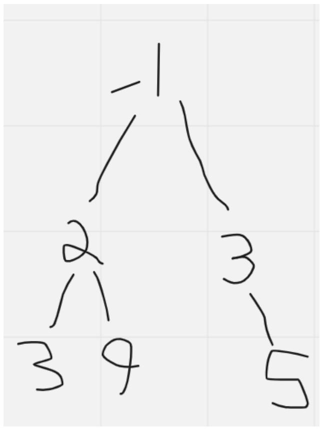


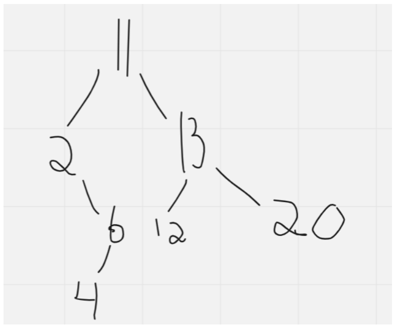
CODE using the following definition for various classes. You may use helper methods in all cases if that will make it easier. Notice it is storing doubles for data:

```
private static class DoubleTreeNode {
    private double data;
    private DoubleTreeNode left;
    private DoubleTreeNode right;
    public DoubleTreeNode(double data) {
      this.data = data;
    }
  }
private class Tree {
      DoubleTreeNode overallRoot;
      int size;
//rest of class not shown
}
```

4. Using post-order tree traversal, make a method that will print a list of all the edges between nodes of a tree. Do not include edges between a non-null and a null tree node. Go with the left edge first, then right.

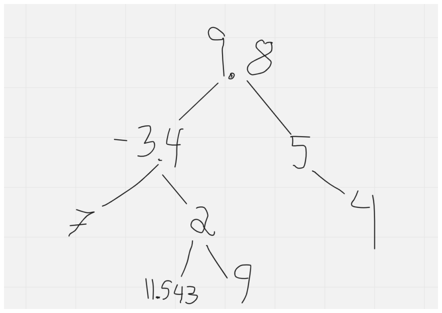
Tree	Method Call	Printed in the console

 The overall root node is called *a*	`printEdges(a);`	`2.0 ->  3.0`
		`2.0 -> 9.0`
		`3.0 -> 5.0`
		`-1.0 ->  2.0`
		`-1.0 -> 3.0`

 The overall root node is called *a*	`printEdges(a);`	`6.0 ->  4.0`
		`2.0 -> 6.0`
		`13.0 ->  12.0`
		`13.0 -> 20.0`
		`11.0 ->  2.0`
		`11.0 -> 13.0`

```
public void printEdges() {

}
```

5. Write a method that will return `true` if the given tree or subtree has a number whose *tenths* place is non-zero, and `false` otherwise.



Tree.
The overall root is called *a.*
The method calls given will never be to null nodes and will never cause nullpointer exception.

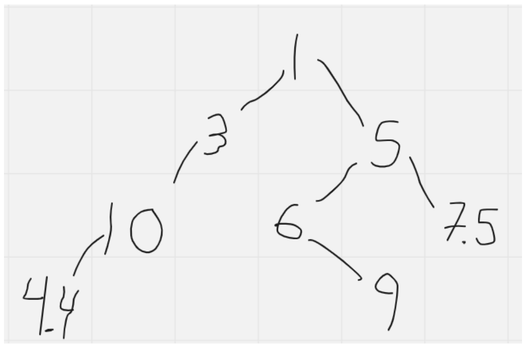
Method Call	Return Value	Notes

tenthsCheck(a)	`true`	-3.4, 9.8, and 11.543
tenthsCheck(a.left)	`true`	-3.4 and 11.543
tenthsCheck(a.right)	`false`	None
tenthsCheck(a.left.right.right)	`false`	None

```
public boolean tenthsCheck(DoubleTreeNode root) {

}
```

6.

a. Write the *depth method.* It returns the depth of a node in terms of nodes passed from the root to the node, inclusive of both. The value passed into the method is always in the binary tree, at exactly one node (no duplicate nodes allowed).



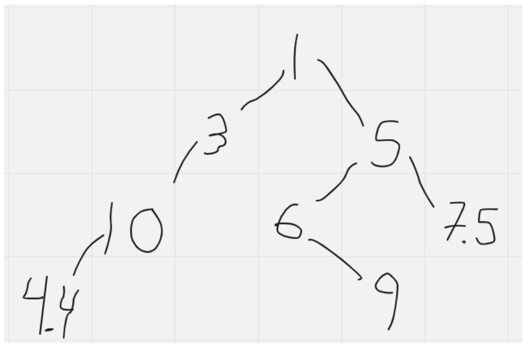
The tree is not necessarily a binary search tree.

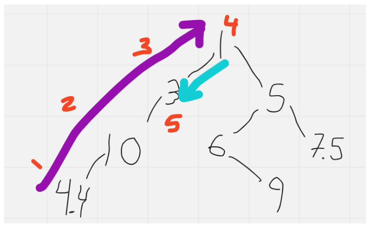
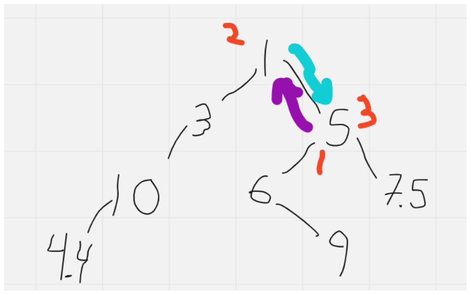
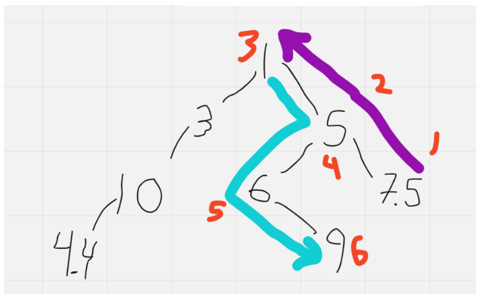
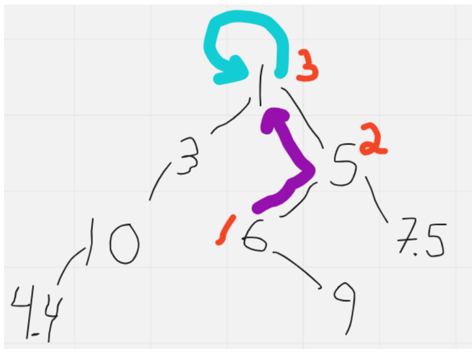
Method Call	Return Value
depth(1)	1
depth(4.4)	4
depth(6)	3

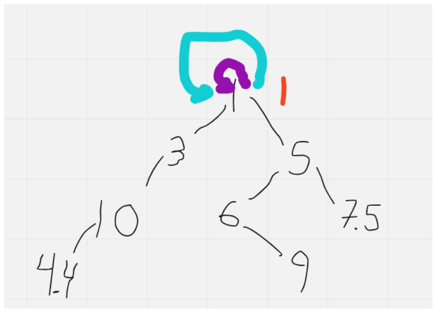
```
public int depth(double target) {

}
```

b. Write a method that will return the length of the path of traveling from node with value *A* to the root, and then to node with value *B*. Count the length of the path by the number of nodes passed, inclusive of A and B. You can use the depth method from a) and assume it will work perfectly as described regardless of your response in a).

The nodes A and B given will always be in the tree.
For clarity, the values A and B will show up exactly once in the tree.

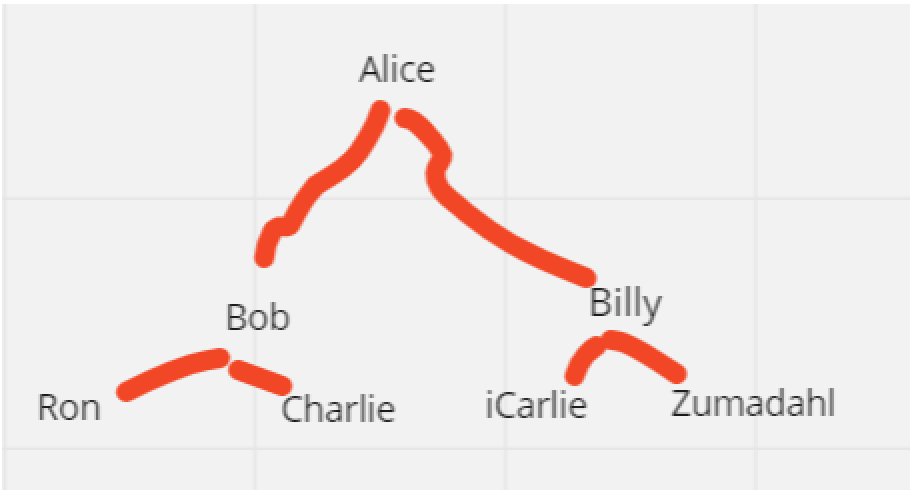


Method Call	Return Value	Notes
pathLen(4.4, 3)	5	
pathLen(5, 5)	3	
pathLen(7.5, 9)	6	
pathLen(6, 1)	3	

pathLen(1,1)	1	

```
public int pathLen(double A, double B) {


}
```

Pratice Problems Answer Key

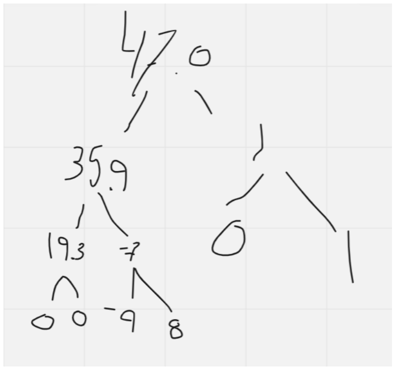
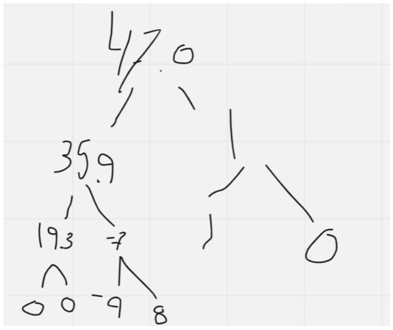


1.

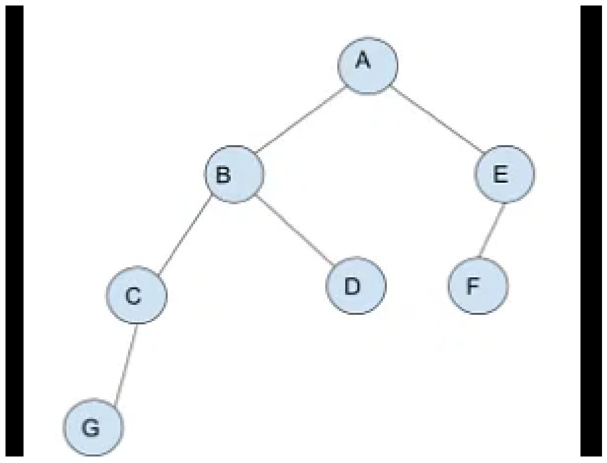
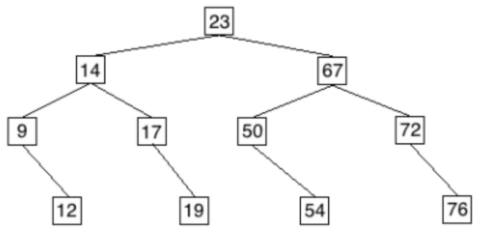
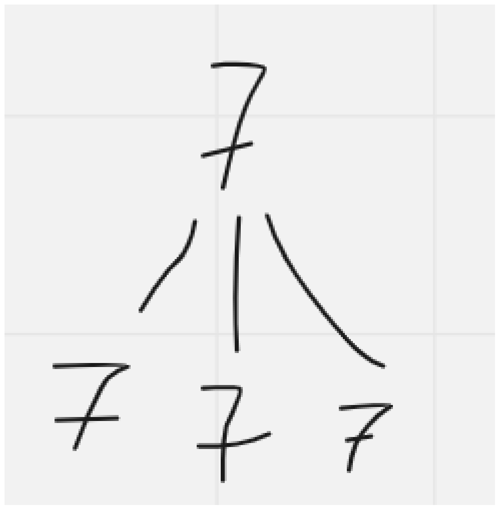
This is a **min heap**

2. Remove the heap

This is a max heap.

	OR	

3. Classifications

	Binary Search Tree that is balanced Min Heap
 23 14 67 9 17 50 72 12 19 54 76	Binary Search Tree that is balanced
	Tree

4. Code printEdges with postorder

```
public void printEdges() {
    printEdges(overallRoot);
  }
```

```
 public static void printEdges(DoubleTreeNode root) {
      if(root == null) {
        return;
      }
      printEdges(root.left);
      printEdges(root.right);
      if(root.left != null) {
        System.out.println(root.data +  " ->  " + root.left.data);
      }
      if(root.right != null) {
        System.out.println(root.data + " -> " + root.right.data);
      }
  }
```

5. tenthsCheck

```
public boolean tenthsCheck(DoubleTreeNode root) {
      if(root == null) {
      return false;
      }
      if(root.data % 1.0 != 0) {
            return true;
      }

      if(tenthsCheck(root.left)) {
            return true;
      }

      if(tenthsCheck(root.right)) {
            return true;
      }
      return false;
}
```

6. Multi-part question
   a. Depth

```
public int depth(double target) {
      return depth(overallRoot, target, 1);
}
public int depth(DoubleTreeNode root, double target, double curdepth) {
      if(root == null) {
            return -1;
      }
      if(Math.abs(root.data - target) <= .001)  {//TOLERANCE
```

```
            return curdepth;
      }
      int a = depth(root.left, target, curdepth+1);
      if(a != -1) {
            return a;
      }
      //if not in left subtree, it must be in right
      //(we assume value always in tree)
      a = depth(root.right, target, curdepth+1);
      return a;

}
```

b. pathLen

```
public int pathLen(double A, double B) {
      return depth(A) + depth(B) - 1;
}
```