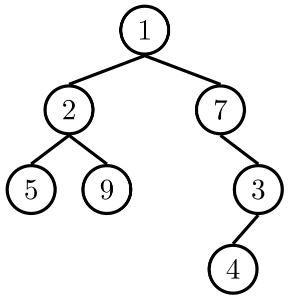
CSM 61B Fall 2020 — So Many Trees

Mentoring 8: October 12 - October 16, 2020

1 Binary Trees

```
public class BinaryTree<T> {
    protected Node root;
    protected class Node {
        public T value;
        public Node left;
        public Node right;
    }
}
```



1.1 Define a procedure, `height`, which takes in a `Node` and outputs the height of the tree. Recall that the height of a leaf node is 0.

```
private int height(Node node) {
    if (node == null) {
        return -1;
    }
    return 1 + Math.max(height(node.left), height(node.right));
}
```

Meta: That this Binary Tree class is an encapsulated binary tree. This means that in a recursive function, we want to recurse on that particular node.

What is the runtime of `height`?

$\Theta(N)$, where N is the number of nodes in the tree. We visit every node once and at each node perform a constant amount of work (**`null`** check). The actual "work" that contributes to the order of growth is done in the recursion, where we repeatedly step down through every node in the tree.

1.2 Define a procedure, `isBalanced`, which takes a `Node` and outputs whether or not the tree is balanced. A tree is **balanced** if the left and right branches differ in height by at most one and are themselves balanced.

```
private boolean isBalanced(Node node) {
    if (node == null) {
        return true;
    } else if (Math.abs(height(node.left) - height(node.right)) <= 1) {
        return isBalanced(node.left) && isBalanced(node.right);
    }
    return false;
}
```

What is the runtime of `isBalanced`?

$\Theta(N)$ in the best case, $\Theta(N \log N)$ in the worst case. This can also be read as $\Omega(N), O(N \log N)$ overall.

The best case is if the tree is unbalanced at the root, meaning that the difference in the height of the root's left branch and the root's right branch is greater than one. In this case, we just call `height` twice, once on the left branch and once on the right subtree. After these two calls, we can immediately see that the tree is unbalanced, so we return false. This leads to a runtime of $\Theta(N)$ in the best case.

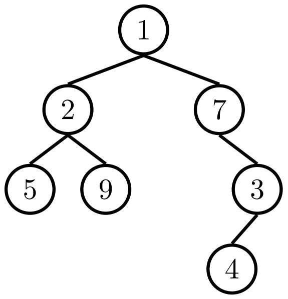
The worst case is if the tree is perfectly balanced. In this case, first, we will call `height` on `node.left` and `node.right`. Each of these nodes has a sub-tree of roughly $N/2$ nodes, and so at this level, 2 height calls are made, each of which costs $N/2$. The total work done on this level is $\Theta(N)$. Next, `node.left` will call `height` on its left and right children, and `node.right` will do the same. These children are now on the third level of the tree (the root node being the first level). Note that these third-level children now have a subtree of roughly $N/4$ nodes each. 4 `height` calls are made at this level, for a total cost of $\Theta(N)$ at this level too. As we keep going, each level will do $\Theta(N)$ work. Note that the bottom-most level (leaf-level) of such a perfectly balanced tree would have roughly $N/2$ nodes, and each height call would take constant time, for a total of $\Theta(N/2) = \Theta(N)$ work at the leaf-level too.

Now that we have established that each level does $\Theta(N)$ work, all that we need to figure out is how many levels there are in our worst case situation. This tree has $\log n$ levels, since a perfectly balanced tree has $\log n$ levels. Therefore, the total runtime cost for the worst case is $\Theta(N \log N)$, which can also be read as $O(N \log N)$

2 Traversals

Level-Order Traversals Nodes are visited top-to-bottom, left-to-right.

Depth-First Traversals Visit deep nodes before shallow ones.



2.1 Give the ordering for each depth-first traversal of the tree.

(a) Pre-order

1 – 2 – 5 – 9 – 7 – 3 – 4

(b) In-order

5 – 2 – 9 – 1 – 7 – 4 – 3

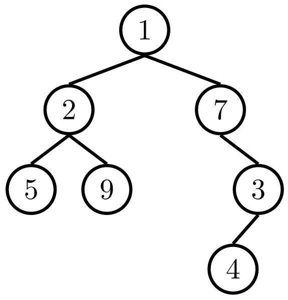
(c) Post-order

5 – 9 – 2 – 4 – 3 – 7 – 1

2.2 Give the level-order traversal of the tree.

1 – 2 – 7 – 5 – 9 – 3 – 4

2.3
```
public void treeTraversal(Fringe<Node> fringe) {
    fringe.add(root);
    while (!fringe.isEmpty()) {
        Node node = fringe.remove();
        System.out.print(node.value);
        if (node.left != null) {
            fringe.add(node.left);
        }
        if (node.right != null) {
            fringe.add(node.right);
        }
    }
}
```



What would Java display?

(a) `tree.traversal(new Queue<Node>());`

```
1275934
```

A Queue has a FIFO (First in First Out) nature that is used to perform the traversal.
To be more specific, you add node 1 to the queue. You then pop it and print its value, and add its children, 2 and then 7 to the queue. Since it is FIFO, you would pop 2 out first and add all its children, 5 and then 9 to the queue. Then you would pop 7, and add its child 3 to the queue. Then you would pop 5 and 9, who have no children to add. Next, you pop 3, add 4 to queue. Lastly you pop 4 and you are done!

Make sure you explain the FIFO nature of the Queue and visually represent how the solution 1275934 is arrived at using a Queue.

(b) `tree.traversal(new Stack<Node>());`

```
1734295
```

A Stack has a LIFO (Last in First Out) nature that is used to perform the traversal.
To be more specific, you add node 1 to the queue. Pop it, add its children, 2 and then 7 to the queue. (Every time you pop a node it gets printed). Since it is LIFO, you would pop 7 out first and add all its children, 3 to the queue. Then you would pop 3, and add its child 4, to the queue. Then you would pop 4, but since it has no children, you would not add anything to the Stack. Once you've popped 4, you would pop 2, and add its children to the Stack - 5 and then 9. Then you would pop 9, but because it has no children, nothing is added to the Stack. Lastly, pop 5!

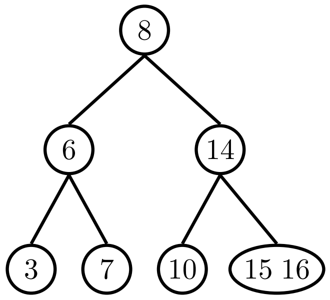
3 Binary Search Trees

3.1 Implement `fromSortedArray` for binary search trees. Given a sorted `int[] array`, efficiently construct a balanced binary search tree containing every element of the array.

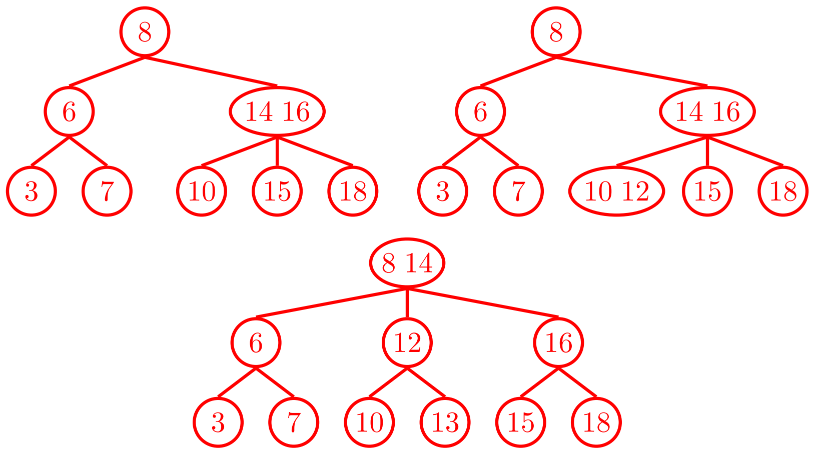
```
public class BinarySearchTree<T extends Comparable<T>> {
    protected Node root;
    protected class Node {
        public T value;
        public Node left;
        public Node right;
    }
    public static BinarySearchTree<Integer> fromSortedArray(int[] values) {
        BinarySearchTree<Integer> bst = new BinarySearchTree<>();
        bst.root = bst.fromSortedArray(values, 0, values.length - 1); // setting a new root
        return bst;
    }
    private Node fromSortedArray(int[] values, int lower, int upper) {
       if (lower > upper) {
           return null;
       }
       int middle = lower + ((upper - lower) / 2); // middle index of the array
       Node mid = new Node();
       mid.value = values[middle];
       mid.left = fromSortedArray(values, lower, middle - 1);  // recurse on the left half
       mid.right = fromSortedArray(values, middle + 1, upper); // recurse on the right half
       return mid;
    }
}
```

Meta: Be sure to explain the difference between a binary tree and a binary search tree.

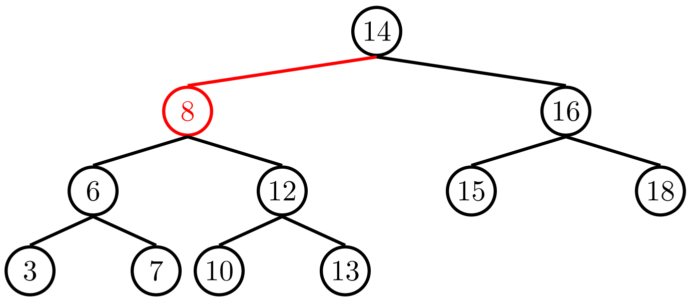
4 2-3 Forever



4.1 Draw what the 2-3 tree would look like after inserting 18, 12, and 13.



4.2 Now, convert the resulting 2-3 tree to a left-leaning red-black tree.


14
8
16
6
12
15
18
3
7
10
13