CSM 61B Fall 2017

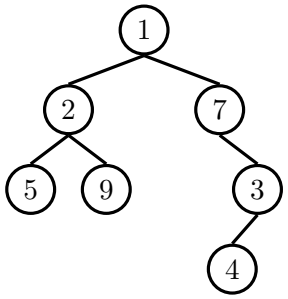
Binary Trees

Mentoring 8: October 16, 2017

1 Binary Trees

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.

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




```
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`?

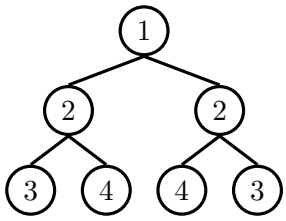
$\Theta(N)$ in the best case, $\Theta(N \log N)$ in the worst case. This can also be read as $\Omega(N), O(N \log N)$ overall.

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 $O(N \log N)$

1.3 Define `isSymmetric` which checks whether the binary tree is a mirror of itself.




```
public boolean isSymmetric() {
    if (root == null) {
        return true;
    }
    return isSymmetric(root.left, root.right); // use helper method
}

private boolean isSymmetric(Node left, Node right) {
    if (left == null) {
        return right == null; // if left is null, right must also be null
    } else if (right == null) {
        return false;         // left is not null but right is null, so not symmetric
    } else if (!left.value.equals(right.value)) {
        return false;         // left value and right value are unequal
    } else {
        return isSymmetric(left.right, right.left) &&
               isSymmetric(left.left, right.right);
    }
}
```

Meta: We can use a helper function here to create a new method that takes in two parameters, the left and the right branch of the current tree we are rooted at. This allows us to more easily compare the content of the two branches to see if they are the same.

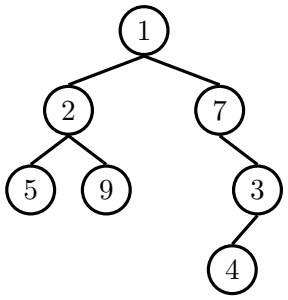
2 Binary Search Trees

2.1 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.

3 Successor *Extra Practice*



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

Depth-First Traversals Visit deep nodes before shallow ones.

3.1 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

$5 - 9 - 2 - 4 - 3 - 7 - 1$

3.2 Give the level-order traversal of the tree.

$1 - 2 - 7 - 5 - 9 - 3 - 4$

3.3 Given a node in a binary search tree (with parent pointers), implement `successor` which returns the next node in the in-order traversal of the BST. If there is no successor, return **`null`**.

```
public class BinarySearchTree<T extends Comparable<T>> {
    protected Node root;
    protected class Node {
        public T value;
        public Node parent, left, right;
    }
    private Node successor(Node node) {
        if (node.right != null) {
            node = node.right;
            while (node.left != null) {
                node = node.left;
            }
            return node;
        } else {
            Node parent = node.parent;
            while (parent != null && parent.right == node) {
                node = parent;
                parent = parent.parent;
            }
            return parent;
        }
```

```
    }
}
```

There are two cases to find the next node in the in-order traversal. Let's think about the case specifically for Binary Search Trees. The next node in the in-order traversal is the same as the next biggest element in the Binary Search Tree.

The first case is when the node passed in has a right branch. If it has a right branch, then the next largest element must lie in the sub-tree whose root is the right-child of the given node. Let's call this sub-tree T. The final thing to realize is that the element we are looking for is the smallest element in this sub-tree T. This is why we traverse left-wards, until we reach the left-bottom of T, which is the node we need to return.

The second case is when the node passed in has no right branch. To reason about this case, first consider a simple case with just 2 nodes.

Say that the node passed in is Node 1. Since the node doesn't have a right child, the next-largest node must be somewhere above it. In this example, we see that the answer is Node 2. As we traverse up from Node 2 to Node 1, we realize that the edge between 2 and 1 is a left-edge (1 is a left-child of 2). The general argument here is: keep traversing up the parents. As soon as we traverse a left-edge, we can stop and return the parent node (in this case, Node 2). Let's traverse up the parents, and name the first left-edge we encounter E, the parent on this edge P, and the child C. Why can we be sure that P is the node to return?

The first thing to note is that as we traverse up the tree, if we don't use a left-edge, then the child is greater than the parent. This is not what we want, since we want the next largest node. The next thing to notice is that going up a left-edge means that the parent is necessarily greater, by the definition of a left-edge in a Binary Search Tree. All that's left to argue now is that if node N is passed in, then $P > N$, and that there is no other node that is smaller than P, but greater than N. Consider the following example:

If the node that we're passed in is Node 2, then the first left-edge we will traverse while going up would have 4 as the parent. This example helps us see that the left-edge's parent will be greater, i.e., $P > N$, since it is guaranteed that N lies in the left-subtree of P. Note that we don't need to consider any of P's ancestors. If P has a parent through a right-edge, then that parent is bigger than P, which is not what we want. If P has a parent through a left-edge, then that parent is smaller than P, but it is also smaller than N, since N would lie in the left-subtree of P's parent.

The final argument is that there is no other node that is smaller than P, but greater than N. Let's look back at the example. First, we look at Node 1. Node 1 is smaller than Node 2, since it is a right-edge. In the example, Node 1 does not have any left-children, but if it did, we can be sure that they would all, too, be smaller than Node 2. So, any time we traverse a right-edge, we can ignore not only the right-edge's parent, but also be sure that everything in the left subtree of the right-edge can be safely ignored. This concludes the argument, since we have shown that right-edges or their subtrees do not contain any nodes that are smaller than P but larger than N.