Trees Worksheet

Consider the following tree:

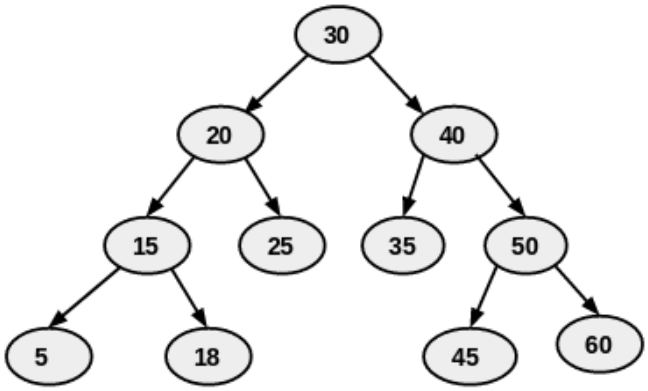


Image source: https://iq.opengenus.org/binary-tree-traversals-inorder-preorder-postorder/

Tree terminology

1. What is the height of this tree?
   Answer: 4

2. **Leaf Nodes**: Identify all the leaf nodes in the tree.
   Answer: 5*, 18, 25, 35 ,45, 60*

3. **Parent Nodes**: For the node with the value 25, who is its parent?
   Answer: 20

4. Give an example of a perfect binary tree

The above tree without the 5,18,45, and 60. More generally, a perfect binary tree is any binary tree with height n that contains exactly 2^n - 1 number of nodes.

5. Give an example of a complete binary tree that is neither perfect nor full

The above tree without 18,25,and 60.

6. Give an example of a full binary tree that is neither perfect nor complete

The above tree without 45 and 60.

A full binary tree: each node has either 0 or 2 children
A complete binary tree: every level is full except possibly the last, which is filled from left to right.

7. True or false:
    a. All perfect trees are full and complete (TRUE)
    b. All complete trees are full, but not necessarily perfect (FALSE)
    c. All full trees are complete, but not necessarily perfect. (FALSE)

Tree traversal

Consider the following binary search tree,

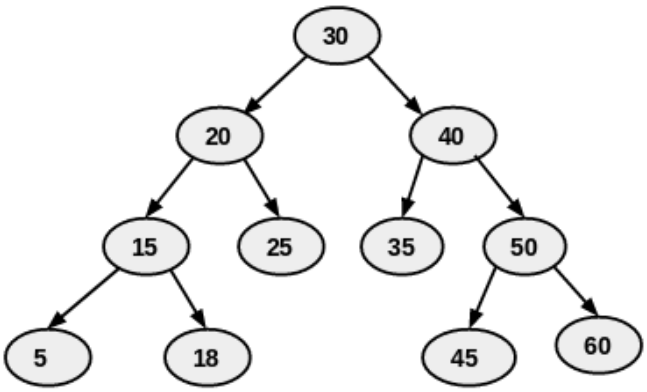


Image source: https://iq.opengenus.org/binary-tree-traversals-inorder-preorder-postorder/

1. Complete the Node class for the binary search tree. Also include a constructor.

```
private class Node {
	private E value;
	private Node left;
	private Node right;
	private Node(V value) {
	this.value = value;
	this.left = null;
	this.right = null;
	}
}
```

2. Assuming we are using pre-order traversal, list out the order the nodes will be visited/printed.
   Answer: 30, 20, 15, 5, 18, 25, 40, 35, 50, 45, 60

3. Assuming we are using in-order traversal, list out the order the nodes will be visited/printed.
   Answer: 5, 15, 18, 20, 25, 30, 35, 40, 45, 50, 60

4. Assuming we are using post-order traversal, list out the order the nodes will be visited/printed.
   Answer: 5, 18, 15, 25, 20, 35, 45, 60, 50, 40, 30

5. Assuming we are using level-order traversal, list out the order the nodes will be visited/printed.
   Answer: 30, 20, 40, 15, 25, 35, 50, 5, 18, 45, 60

6. Write a method `public void inOrder()` that prints each value in the tree using in-order traversal. You must use a helper function that is recursive.

```
public void inOrder() {
	inOrder(root);
	System.out.println("");
}
private void inOrder(Node n) {
	if (n == null) {
		return;
	}
	inOrder(n.left);
	System.out.print(" "+n.key);
	inOrder(n.right);
}
```

7. Write a method `public void preOrder()` that prints each value in the tree using pre-order traversal. You must use a helper function that is recursive.

```
public void preOrder() {
	preOrder(root);
	System.out.println("");
}
private void preOrder(Node n) {
	if (n == null) {
		return;
	}
	System.out.print(" "+n.key);
	preOrder(n.left);
	preOrder(n.right);
}
```

8. Write a method `public void postOrder()` that prints each value in the tree using post-order traversal. You must use a helper function that is recursive.

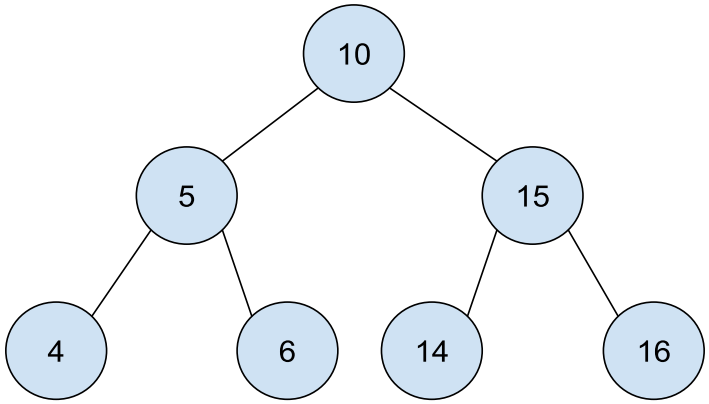
```
public void postOrder() {
	postOrder(root);
	System.out.println("");
}
private void postOrder(Node n) {
	if (n == null) {
		return;
	}
	postOrder(n.left);
	postOrder(n.right);
	System.out.print(" "+ n.key);
}
```

9. Write a method `public int height()` that returns the height of the tree. You must use a helper function that is recursive.

```
public int height() {
	return height(root);
}

private int height(Node n) {
	if (n == null) {
		return 0;
	}
	else {
		int heightLeft = height(n.left);
		int heightRight = height(n.right);
		if (heightLeft >= heightRight) {
			return 1 + heightLeft;
		}
		else {
			return 1 + heightRight;
		}
	 }
}
```

Insertion and Deletion



1. Here is a function that inserts a new key and value into a tree, but a few lines are missing. Complete the function and add comments to explain what's going on.

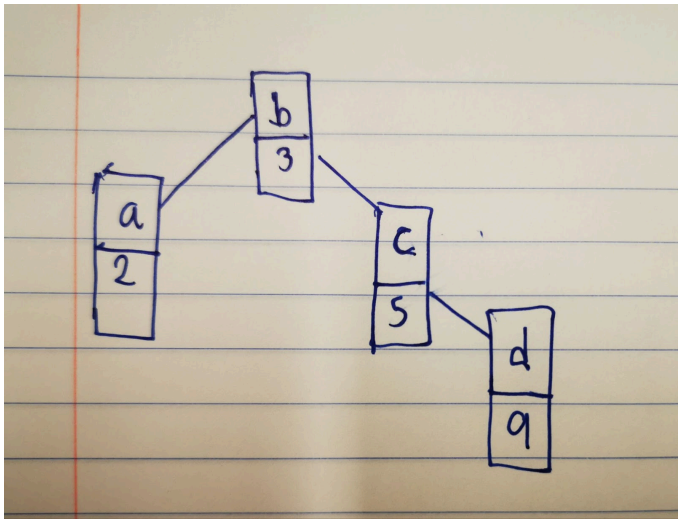
```
public void insert(K key, V value) {
	root = insert(key, value, root);
}
private Node insert(K key, V value, Node n) {
	if (n == null) { // node should go here!
		Node newNode = new Node(key, value);
		return newNode;

	} else if (key.compareTo(n.key) < 0) { // node should go in left
		n.left = insert(key, value, n.left);
		return n;
	} else if (key.compareTo(n.key) > 0) { // node should go in right
		n.right = insert(key, value, n.right);
		return n;
	} else { //If the key already exists in the tree, just replace the old value
with the new one.
		n.value = value;
		return n;
	}
}
```

2. Using the above algorithm for insert, draw a diagram of myTree showing its contents after the following series of calls:

BST<String,Integer> myTree = new BST<String, Integer>();

myTree.insert(b,3);
myTree.insert(c,5);
myTree.insert(d,9);
myTree.insert(a,2);

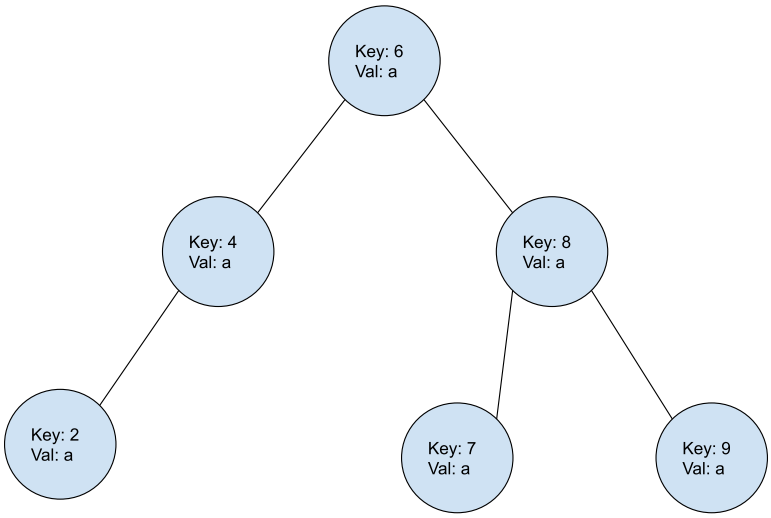


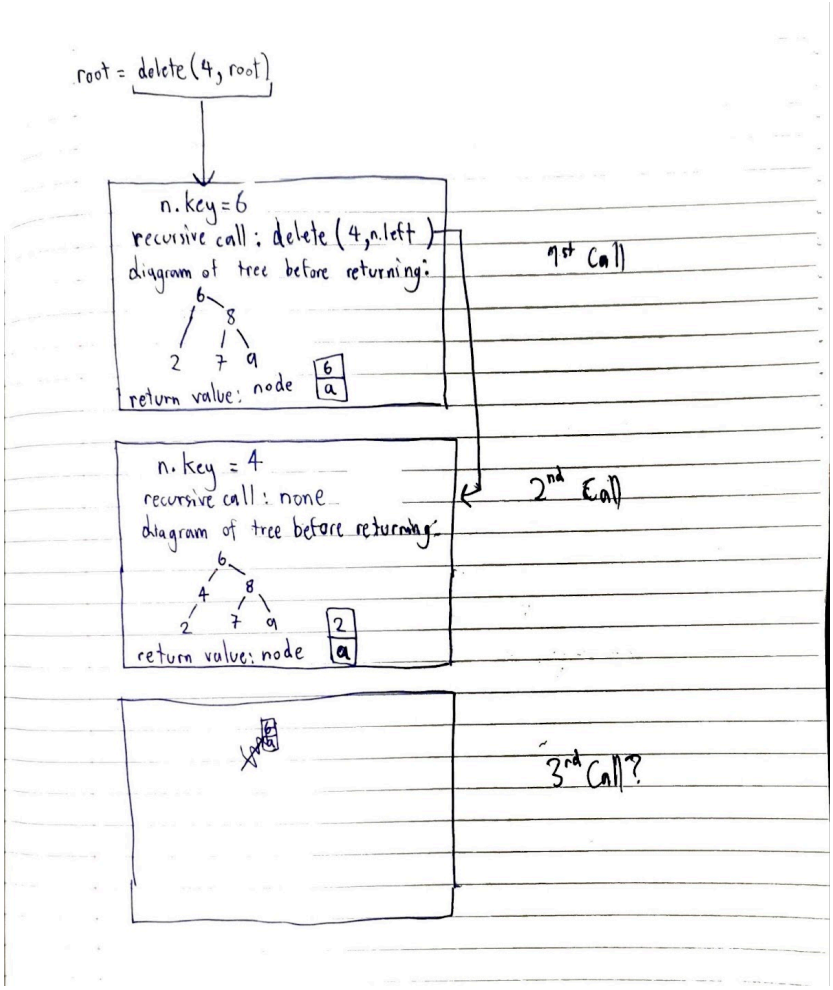
3. Below is the delete algorithm that we learned in class. When we call delete(4) on the following tree, show in each call to the recursive function
    a. The key of the current node n
    b. The diagram of the tree before returning
    c. The return value.

```
public void delete(K key) {
	root = delete(key,root);
}

private Node delete(K key, Node n) {
	if (n == null) {
		return null;
	} else if (key.compareTo(n.key) < 0) { // Go left!
		n.left = delete(key, n.left);
		return n;
	} else if (key.compareTo(n.key) > 0) { // Go right!
		n.right = delete(key, n.right);
		return n;
	} else { //Aha! n is the node that has to be deleted!
		if (n.left == null && n.right == null) {
			return null;
```

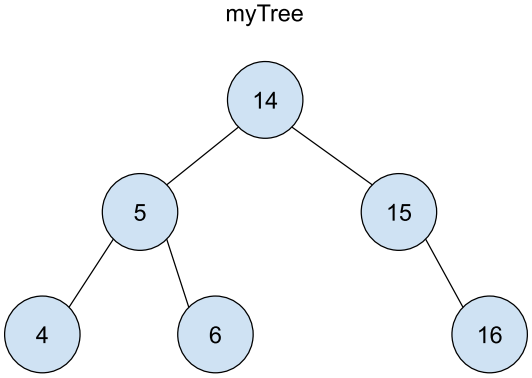
```
            } else if (n.left == null) {
                  return n.right;
            } else if (n.right == null) {
                  return n.left;
            } else { //If n has two children, then…
                  //ANSWER:
                  //Node s = smallest(n.right);
                  //Node newRoot = new Node(s.key, s.value);
                  //newRoot.left = n.left;
                  //newRoot.right = delete(s.key, n.right);
                  //return newRoot;
            }
      }
}
```



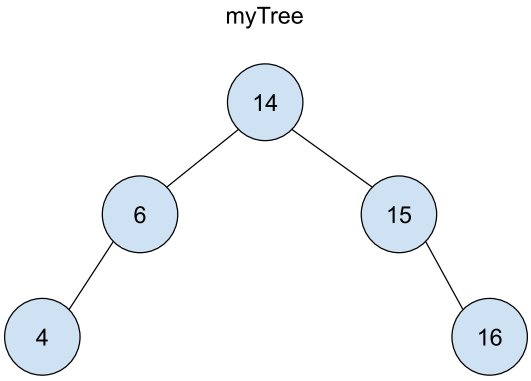


4. In the delete function above, fill in one of the base cases.
5. What would myTree look like after each line of the following code:

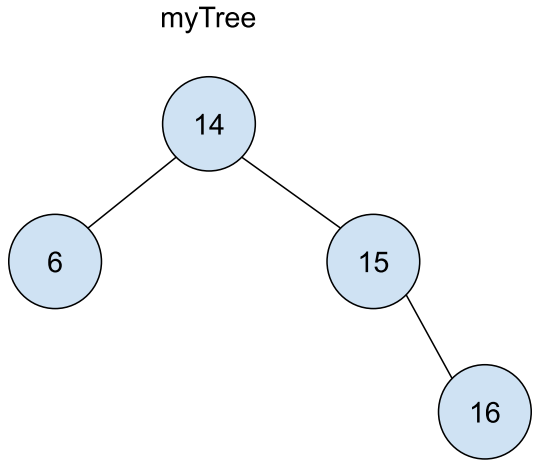
myTree.delete(10);



myTree.delete(5);



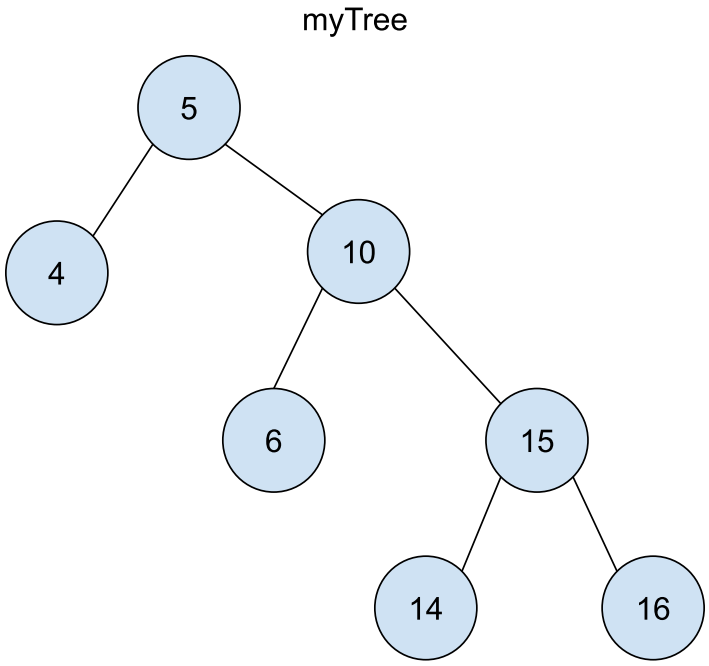
myTree.delete(4):



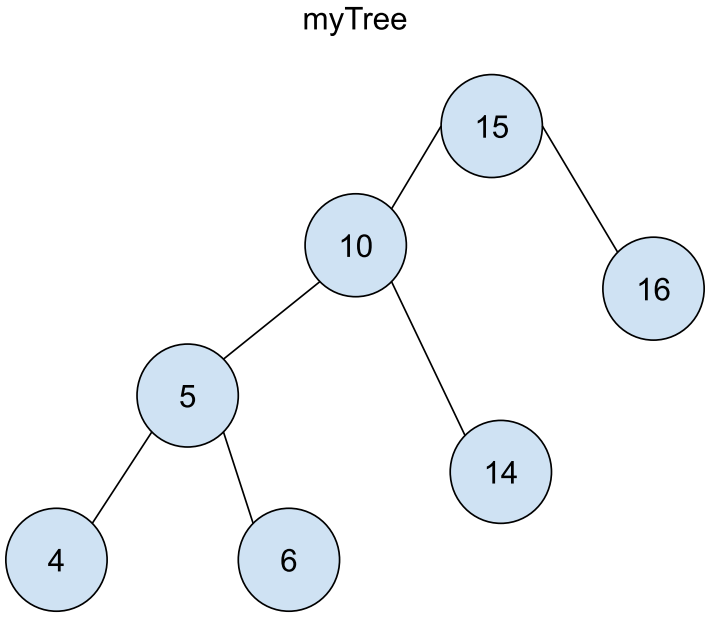
AVL trees

1. What condition must be true for any AVL binary tree?
   Answer: for any node in the AVL tree, the difference in height of the left and right subtree can't be greater than 1. In other words, the balance factor ($Height_{Left}$ - $Height_{Right}$) can't be greater or smaller than 1 or -1.

2. Using the myTree diagram above, draw diagram(s) to illustrate the steps required to perform a right rotation at 10.



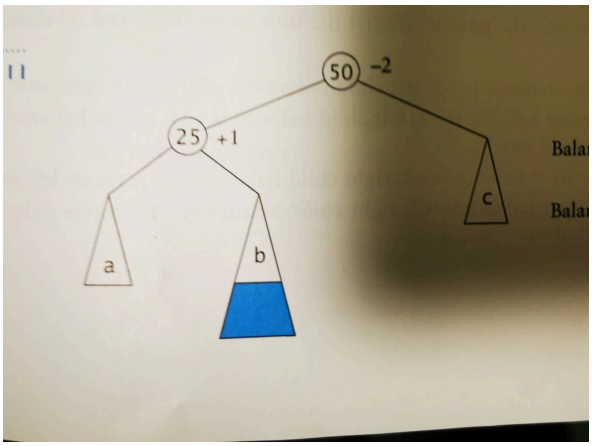
3. Using the myTree diagram above, draw diagram(s) to illustrate the steps required to perform a left rotation at 10.

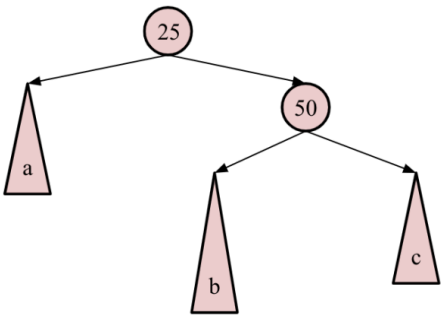


4. What are four types of unbalanced trees? What type is your tree in 2. and 3.?
    a. Root is left-heavy, and left subtree is also left heavy (Left-left)
    b. Root is left-heavy and left subtree is right-heavy (Left-Right)
    c. Root is right-heavy and right subtree is right heavy (Right-right)
    d. Root is right heavy and right subtree is left heavy (Right-left)

5. Show the effect of just rotating right at 50 on the tree in Figure 9.11 in the textbook. Why doesn't this fix the problem? (Question from Koffman and Elliot)

Figure 9.11





This doesn't work because it does not move the level that the unweighted subtree is at, and instead simply shuffles around how they're organized.

6. How does the AVL insert method maintain tree balance?
    a. Every time that we insert a new node, we update the balance of the nodes affected. For example, if we add 2 to myTree, then node 4 should have a balance of -1. Then, call the function rebalanceLeft or rebalanceRight if balance is less than 1 or more than 1.

7. Build an AVL tree that inserts the integers 30, 40, 15, 25, 90, 80, 70, 85, 15, 72 in the given order. (Question from Koffman and Elliot)