# Assignment 6 – Andre Godinez

## Code:

## BinaryTreeDemo

```
package assignment_7;

/** M Madden, Feb 2008:
 * Class to demonstrate the use of BinaryTree code.
 * Based on code by Carrano & Savitch.
 * @author Michael Madden.
 */

public class BinaryTreeDemo
{

        static int num=0;


        public static void main(String[] args)
        {
                // Create a tree
                /*System.out.println("Constructing a test tree ...");
                BinaryTree<String> testTree = new BinaryTree<String>();
                createTree1(testTree);*/

                System.out.println("Constructing a test tree ...");
                BinaryTree<String> testTree2 = new BinaryTree<String>();
                testTree2=createTree2(3);

                // Display some statistics about it
                System.out.println("\nSome statistics about the test tree ...");
                displayStats(testTree2);

                //no need for in order

                /*// Perform in-order traversal
                System.out.println("\nIn-order traversal of the test tree, printing each node when visiting it ...");
                testTree2.inorderTraverse();*/

                // Perform pre-order traversal
                System.out.println("\nPre-order traversal of the test tree, printing each node when visiting it ...");
                testTree2.preorderTraverse();

                // Perform post-order traversal
                System.out.println("\nPost-order traversal of the test tree, printing each node when visiting it ...");
                testTree2.postorderTraverse();

                // Perform breadth traversal
                System.out.println("\nBreadth first traversal of the test tree, printing each node when visiting it ...");
                testTree2.TreeBreadthFirst();


        } // end of main

        public static void createTree1(BinaryTree<String> tree)
        {
                // To create a tree, build it up from the bottom:
                // create subtree for each leaf, then create subtrees linking them,
                // until we reach the root.

                System.out.println("\nCreating a treee that looks like this:\n");
                System.out.println("     A      ");
            System.out.println("   /   \\   "); // '\\' is the escape character for backslash
            System.out.println("  B     C   ");
            System.out.println(" / \\   / \\");
            System.out.println("D   E  F  G ");
            System.out.println();

                // First the leaves
                BinaryTree<String> dTree = new BinaryTree<String>();
                dTree.setTree("D");
                // neater to use the constructor the initialisation ...
                BinaryTree<String> eTree = new BinaryTree<String>("E");
                BinaryTree<String> fTree = new BinaryTree<String>("F");
                BinaryTree<String> gTree = new BinaryTree<String>("G");
        // Now the subtrees joining leaves:
                BinaryTree<String> bTree = new BinaryTree<String>("B", dTree, eTree);
                BinaryTree<String> cTree = new BinaryTree<String>("C", fTree, gTree);

                // Now the root
                tree.setTree("A", bTree, cTree);
        } // end createTree1
```

```
    public static BinaryTree<String> createTree2(int height){


        num++; //use this as in index for the nodes
        String node = Integer.toString(num); //

        BinaryTree<String> leaf;

        System.out.println("Height: " + height + "\tNumber: " +node);

        if(height<=1){
            //root node
            leaf = new BinaryTree<String>(node,null,null);
        }

        else{
            //parent nodes with child nodes
            leaf = new BinaryTree<String>(node,createTree2(height-1),createTree2(height-1));
        }
        return leaf;
    }

    public static void displayStats(BinaryTree<String> tree)
    {
        if (tree.isEmpty())
            System.out.println("The tree is empty");
        else
            System.out.println("The tree is not empty");

        System.out.println("Root of tree is " + tree.getRootData());
        System.out.println("Height of tree is " + tree.getHeight());
        System.out.println("No. of nodes in tree is " + tree.getNumberOfNodes());
    } // end displayStats
}
```

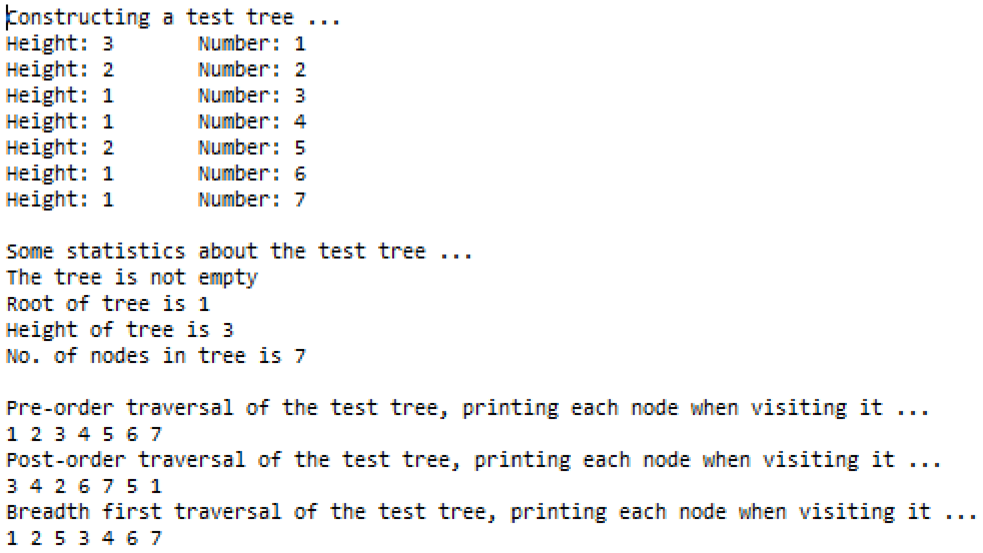
## BinaryTree methods

```
/* PRE ORDER TRAVERSAL
     * visit root before the subtrees.
     * Starts with root
     * checks if root as left child and does preorder traversal on that child
     * if it doesnt it gets the right child
     * Basically if it has a left child keep going left and do preorder traversal on that
     */

    public void preorderTraverse(){
    preorderTraverse(root);
    }

    private void preorderTraverse(BinaryNodeInterface<T> node)       {

    if (node != null){//while the node isn't null

    System.out.print(node.getData() + " ");//print out the node
    preorderTraverse(node.getLeftChild());//go left and do preorder
    preorderTraverse(node.getRightChild());//go right and do preorder
        }
    }

    ////////////////////END PRE ORDER TRAVERSAL/////////////////////////////////

    /*POST ORDER TRAVERSAL
     *visit root after visiting the subtrees
     *Starts with the left sub tree
     *print out the left and right nodes starting from the bottom left subtree
     *Then the right subtree
     *Gets the left child nodes of the subtree first
     *then all the right child nodes
     *and finishes with the root
     */
    public void postorderTraverse(){
    postorderTraverse(root);
    }

    private void postorderTraverse(BinaryNodeInterface<T> node)      {
    if (node != null){
    postorderTraverse(node.getLeftChild());//go left child and do postorder
    postorderTraverse(node.getRightChild());//go right child and do post order
    System.out.print(node.getData() + " ");//print out the root
        }
    }

    /////////////////////END POST ORDER TRAVERSAL//////////////////////////////
```

```
	public void TreeBreadthFirst(){
	/**Begin with root node of the queue
	 * At each iteration, dequeue the node at the head of the queue, and enqueue its children
		to the end of the queue (if it has any), and visit the removed node
		o Repeat this until the queue is empty
	 */

	BinaryNode node = (BinaryNode) this.root;      //Beging with root node of queue
	Queue queue;                  //create temporary queue
	BinaryNodeInterface<T> traverse;      //points which node are processed


	if (node.getData() == null)
	return;  //nothing to traverse

	queue = new LinkedList();   //create queue to hold nodes
	queue.add(node);


		while (!(queue).isEmpty()) {
			traverse = (BinaryNodeInterface<T>) (queue).remove(); //dequeue head of the node and enqueue its childres
if they have any
			System.out.print(traverse.getData() + " ");
				if (traverse.getLeftChild() != null)
					queue.add(traverse.getLeftChild());			//enqueuing the children to the
queue
							if (traverse.getRightChild() != null)
							queue.add(traverse.getRightChild());
							}
				}
```

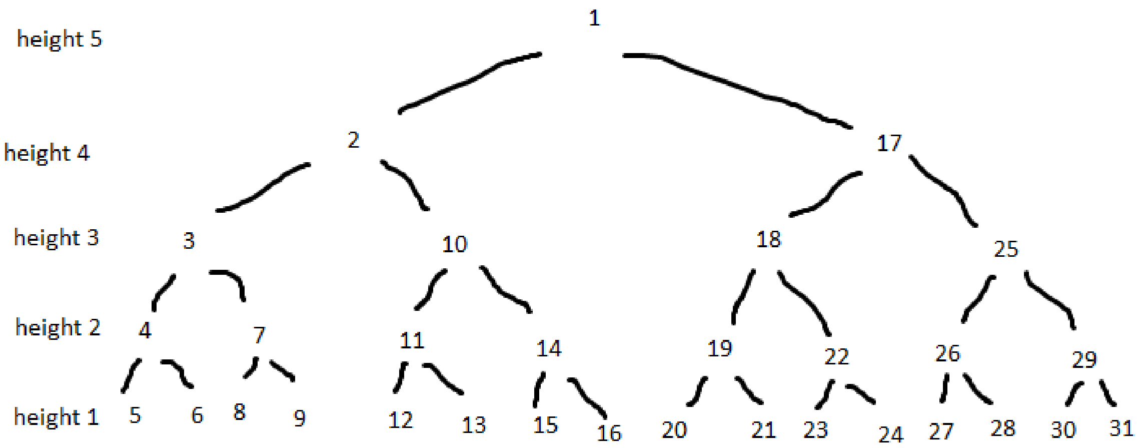
I will only verify for height 5. No need to verify both.

## Result for height 3



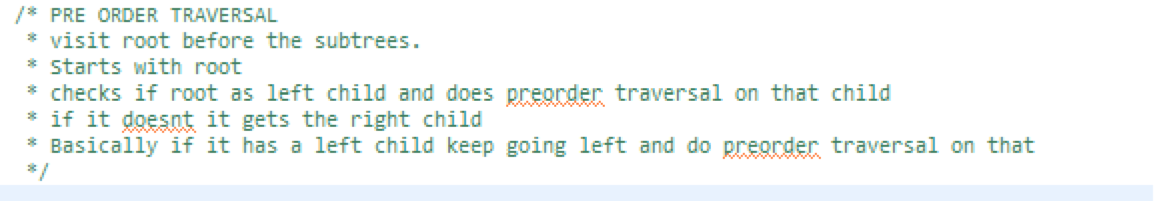
```
Constructing a test tree ...
Height: 3       Number: 1
Height: 2       Number: 2
Height: 1       Number: 3
Height: 1       Number: 4
Height: 2       Number: 5
Height: 1       Number: 6
Height: 1       Number: 7

Some statistics about the test tree ...
The tree is not empty
Root of tree is 1
Height of tree is 3
No. of nodes in tree is 7

Pre-order traversal of the test tree, printing each node when visiting it ...
1 2 3 4 5 6 7
Post-order traversal of the test tree, printing each node when visiting it ...
3 4 2 6 7 5 1
Breadth first traversal of the test tree, printing each node when visiting it ...
1 2 5 3 4 6 7
```

## Result for height 5

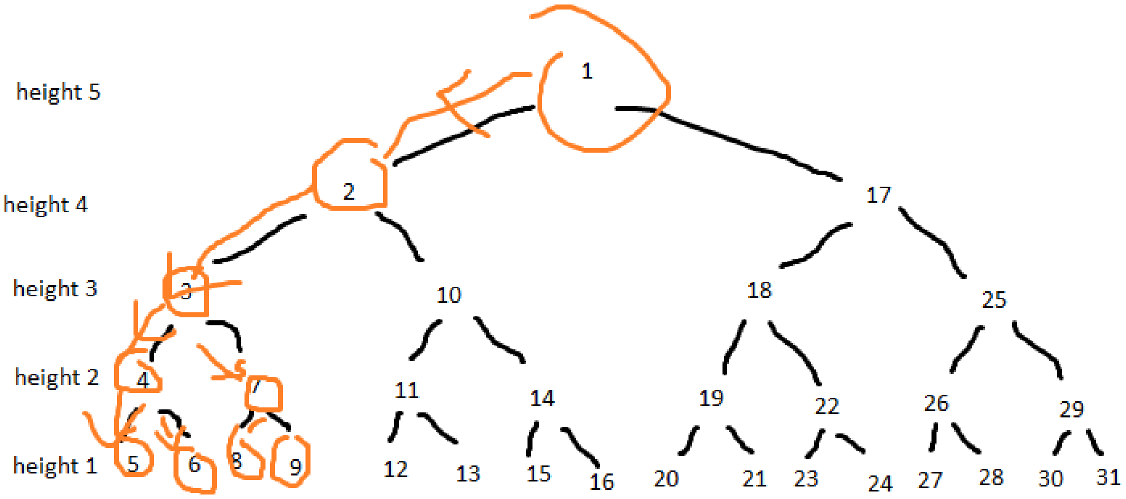


Verifying :

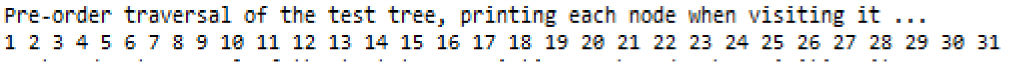
Pre order traversal



```
/* PRE ORDER TRAVERSAL
 * visit root before the subtrees.
 * Starts with root
 * checks if root as left child and does preorder traversal on that child
 * if it doesnt it gets the right child
 * Basically if it has a left child keep going left and do preorder traversal on that
 */
```



Pre order traversal result

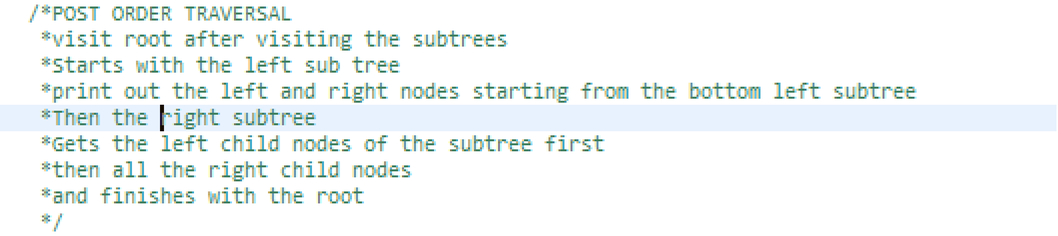


```
Pre-order traversal of the test tree, printing each node when visiting it ...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

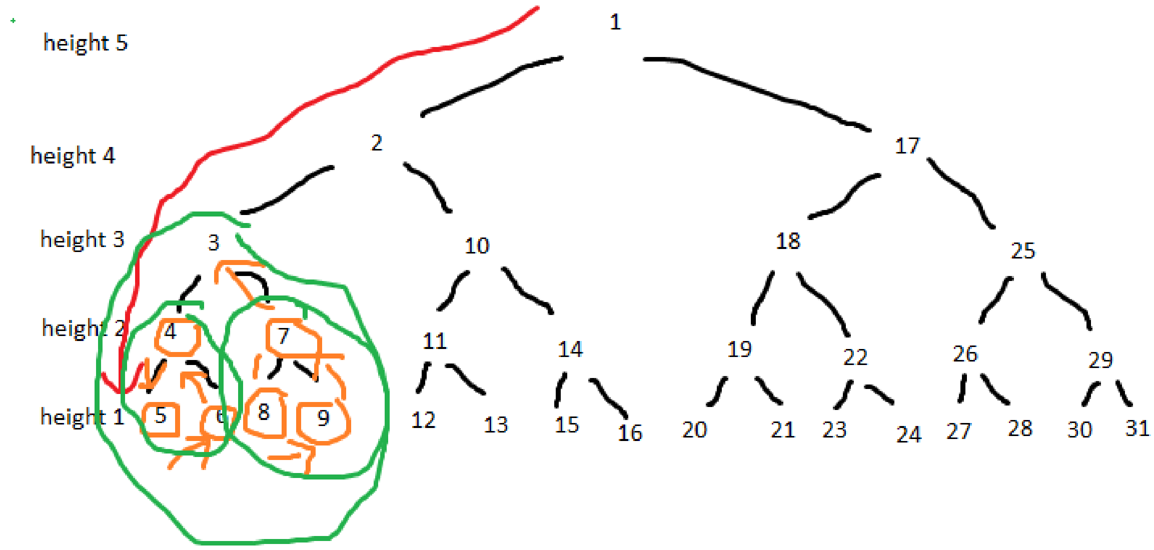
From our diagram we see that it works. No need to explain.

Verifying :

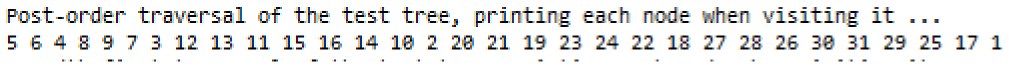
Post order traversal



```
  /*POST ORDER TRAVERSAL
   *visit root after visiting the subtrees
   *Starts with the left sub tree
   *print out the left and right nodes starting from the bottom left subtree
   *Then the right subtree
   *Gets the left child nodes of the subtree first
   *then all the right child nodes
   *and finishes with the root
   */
```



Post order traversal result

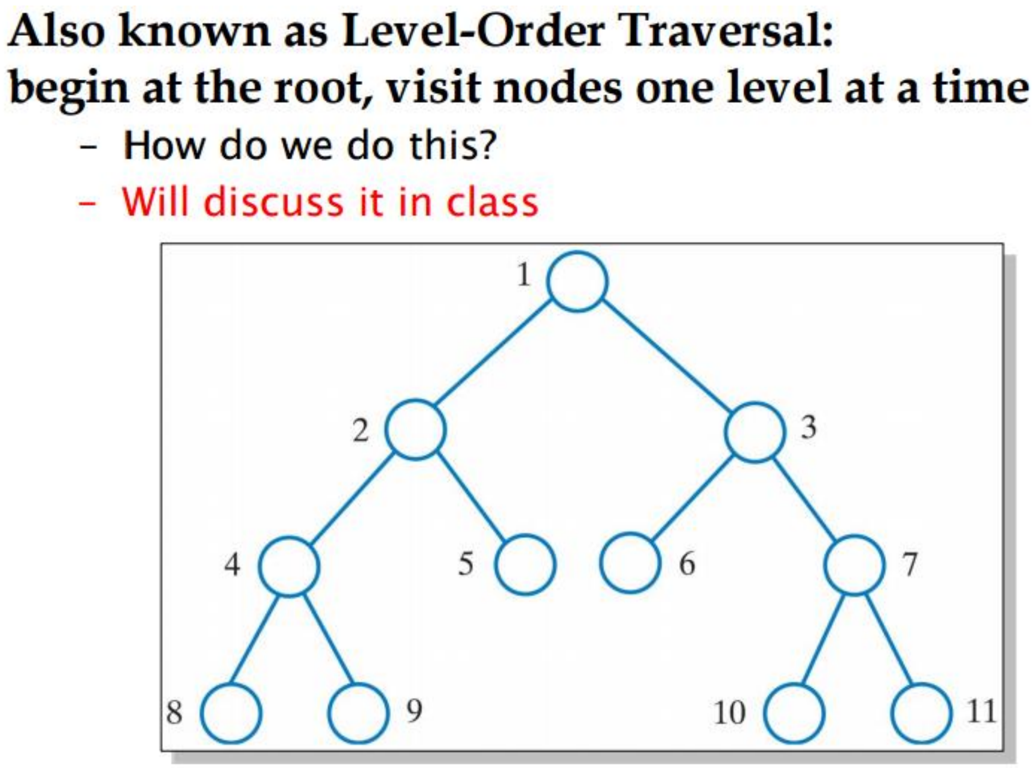


```
Post-order traversal of the test tree, printing each node when visiting it ...
5 6 4 8 9 7 3 12 13 11 15 16 14 10 2 20 21 19 23 24 22 18 27 28 26 30 31 29 25 17 1
```

From the diagram we see that after it travels to the very left most node and get the right node and then the parent node. Then it goes to the right node of parent node 3 and goes to the very left of that which only happens to be 8 and then gets 9. It does the same thing for parent node 10. It goes to the very left which is 12 etc.

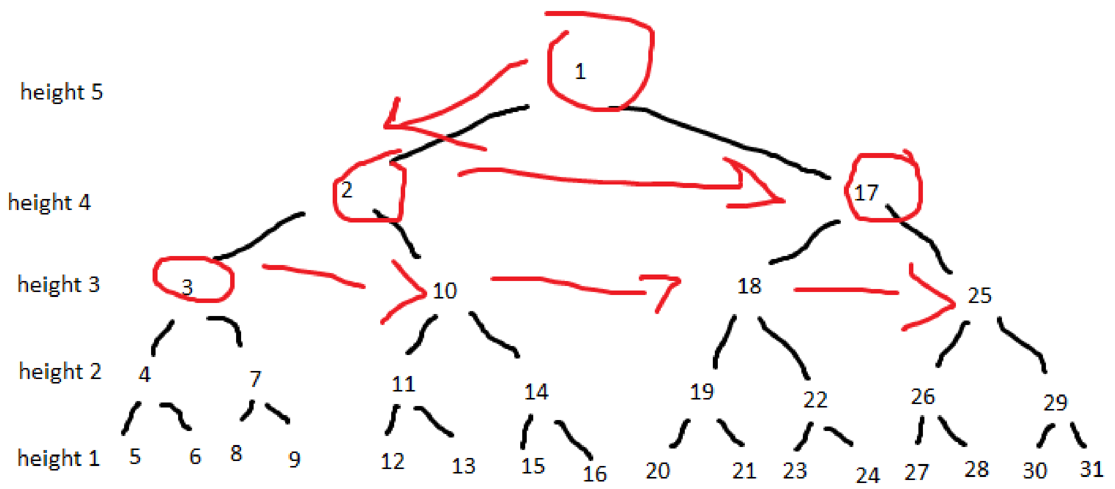
Verifying :

Breadth traversal



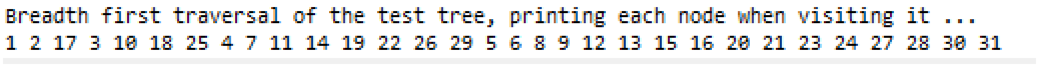
**Also known as Level-Order Traversal:**
**begin at the root, visit nodes one level at a time**

- How do we do this?
- Will discuss it in class



We can see how breadth traversal works aswell.

Breadth first traversal result



```
Breadth first traversal of the test tree, printing each node when visiting it ...
1 2 17 3 10 18 25 4 7 11 14 19 22 26 29 5 6 8 9 12 13 15 16 20 21 23 24 27 28 30 31
```