Robert, Havin, Raj
Lab 31-33
11/20/22
Written/online sources used: none
Help obtained: none
we confirm that the above list of sources is complete AND that we have not talked to anyone else (e.g., CSC 207 students) about the solution to this problem.

Lab 31 - Binary Tree and Tree Traversal

Exercise 2: Alternate Output

1) Pre-order traversal
2) void preorder(PrintWriter pen, BinaryTreeNode<T> node) {

```
        if (node != null)  {
                pen.print(node.value + " ");
        if ((node.left != null) || (node.right != null)) {
                preorder(pen, node.left);
                preorder(pen, node.right);
        } // if has children
    } // else
} // dump
```

3) aardvark billygoat chinchilla dingo emu frog gnu hippo iguana jackalope koala llama
4) void inorder(PrintWriter pen, BinaryTreeNode<T> node) {

```
        if (node != null)  {
                if(node.left != null) {
                        inorder(pen, node.left);
                }
                pen.print(node.value + " ");
                if ((node.right != null)) {
                        inorder(pen, node.right);
                } // if has children
        } // else
    } // dump
```

Exercise 5: Other orderings

```
// Inorder
public void print(PrintWriter pen) {
        // A collection of the remaining things to print
```

```
        Stack<Object> remaining = new Stack<Object>();
        remaining.push(this.root);
        while (!remaining.isEmpty()) {
          Object next = remaining.pop();
          if (next instanceof BinaryTreeNode<?>) {
            @SuppressWarnings("unchecked")
            BinaryTreeNode<T> node = (BinaryTreeNode<T>) next;
            if (node.right != null) {
                    remaining.push(node.right);
                  }
            remaining.push(node.value);
            if (node.left != null) {
                    remaining.push(node.left);
            }
             // if (node.right!= null)
             // if (node.left != null)
          } else {
            pen.print(next);
            pen.print(" ");
          } // if/else
        } // while
        pen.println();
      } // print(PrintWriter)

// Postorder
public void print2 (PrintWriter pen) {
        // A collection of the remaining things to print
        Stack<Object> remaining = new Stack<Object>();
        remaining.push(this.root);
        while (!remaining.isEmpty()) {
          Object next = remaining.pop();
          if (next instanceof BinaryTreeNode<?>) {
            @SuppressWarnings("unchecked")
            BinaryTreeNode<T> node = (BinaryTreeNode<T>) next;
            remaining.push(node.value);
            if (node.right != null) {
                    remaining.push(node.right);
                  }
            if (node.left != null) {
                    remaining.push(node.left);
```

```
      }
       // if (node.right!= null)
       // if (node.left != null)
    } else {
      pen.print(next);
      pen.print(" ");
    } // if/else
  } // while
  pen.println();
} // print2(PrintWriter)
```

Extra Points <printSideways>

```
public void printSideways() {
		printSideways(root, 0);
	}

	private void printSideways(BinaryTreeNode<T> root, int level) {
		if (root != null) {
			printSideways(root.right, level + 1);
			for (int i = 0; i < level; i++) {
				System.out.print("    ");
			}
			System.out.println(level + ": " + root.value);
			printSideways(root.left, level + 1);
		}
	}
```

Lab32

//BinaryNode.java

```
public String toString() {
        return toString(this);
    }

    public String toString(BinaryNode<AnyType> node) {
        if(node == null) return "...";
        return "[" + toString(node.left) + " | " + node.element + " | " + toString(node.right) + "]";
    }
```

//BinarySearchTree.java

```
public int IPL() {
        return IPL(0, root);
    }

    public int IPL(int depth, BinaryNode<AnyType> node) {
        if(node == null) return 0;
        return IPL(depth + 1, node.left) + depth + IPL(depth + 1, node.right);
    }
public int height()
    {
        return height(this.root);
```

```
}

public int height(BinaryNode<AnyType> root)
{
    if(root == null)
        return 0;

    else if(root.left == null && root.right == null)
        return 1;

    else if(root.left == null)
        return 1 + height(root.right);

    else if(root.right == null)
        return 1 + height(root.left);

    else
    {
        int lt = height(root.left);
        int rt = height(root.right);

        if(lt >= rt)
            return 1+lt;
```

```
                else

                        return 1+rt;

        }

    }

public void print () {
                // A collection of the remaining things to print
                Stack<BinaryNode<AnyType>> remaining = new Stack<>();
                Stack<BinaryNode<AnyType>> toPrint = new Stack<>();
                remaining.push(this.root);
                while (!remaining.isEmpty()) {
                        toPrint.push(remaining.pop());
                        if(toPrint.peek().left != null) remaining.push(toPrint.peek().left);
                        if(toPrint.peek().right != null) remaining.push(toPrint.peek().right);
                } // while
                while (!toPrint.isEmpty()) {
                        System.out.print(toPrint.pop().element + " ");
                }
                System.out.println();
        }

//BinarySearchTreeTest.java

public class BinarySearchTreeTest {
        public static void main (String[] args) {
                BinarySearchTree<Integer> bst = new BinarySearchTree<>();
                bst.insert(42);
                bst.insert(39);
                bst.insert(61);
                bst.insert(58);
                bst.insert(54);
                bst.insert(50);
                System.out.println(bst);
                System.out.println("height: " + bst.height());
                System.out.println("IPL: " + bst.IPL());
                bst.print();
        }
```

```
}

//output
[[... | 39 | ...] | 42 | [[[[... | 50 | ...] | 54 | ...] | 58 | ...] | 61 | ...]]
height: 5
IPL: 11
39 50 54 58 61 42
```

Lab33

```
//AVLTree.java
public AVLNode find(int key)      // find node with given key
        {                         // (assumes non-empty tree)
                AVLNode temp = root;
                while(temp != null) {
                        if(temp.iData == key) return temp;
                        else if (temp.iData > key) temp = temp.leftChild;
                        else temp = temp.rightChild;
                }
                return null;
        }  // end find()

public void insert(int id) {
                //System.out.println("adding " + id);

                // Increase number of elements in tree
                size++;

                AVLNode newNode = new AVLNode();    // make new node

                // Fill in the fields of the newly created AVLNode
                newNode.iData = id;          // insert data
                newNode.balance = 0;              // a leaf node has balance 0
                newNode.height = 0;          // the height of a tree with just one node is 0

                if(root==null)               // no node in root
                        root = newNode;
                else                         // root occupied
                {
                        if( find(id) != null ) {
                                throw new IllegalArgumentException();
                        }
                        AVLNode current = root;       // start at root and walk down the tree
                        AVLNode parent = null;

                        // An AVLStack is a stack that stores AVLNodes. It will be used
                        // to store the search path
                        AVLStack S = new AVLStack(size);
```

```
// Loop to walk down the tree, until current becomes null
// The nodes visited in this walk are stored in AVLStack
while(current != null)
{
        S.push(current);
        parent = current;

        if(id < current.iData)  // go left?
                current = current.leftChild;
        else                    // or go right?
                current = current.rightChild;
}  // end while

//System.out.println(S);
/**
 * TODO: Use the knowledge or state from the while-loop above to insert
the new node at the  appropriate place.
 */
//AVLNode temp = S.pop();
if(parent.iData > newNode.iData) {
        parent.leftChild = newNode;
}
else {
        parent.rightChild = newNode;
}


// Walk back up the tree, by popping items from the AVLStack S
int leftHeight, rightHeight;
AVLNode pathNode = null;
while(!S.isEmpty())
{
        // This is the current node of the search path
        pathNode = S.pop();

        // Compute its height and balance
        // First find the height of the left subtree
        if(pathNode.leftChild == null)
                leftHeight = -1;
```

```
                    else
                            leftHeight = pathNode.leftChild.height;

                    // Then find the height of the right subtree
                    if(pathNode.rightChild == null)
                            rightHeight = -1;
                    else
                            rightHeight = pathNode.rightChild.height;

                    // Set the balance of the node
                    pathNode.balance = rightHeight - leftHeight;

                    // Set the height of the subtree rooted at the node
                    pathNode.height = 1 + max(leftHeight, rightHeight);

                    // Check if balance is out of bounds.
                    if((pathNode.balance < -1) || (pathNode.balance > 1))
                    {
                            System.out.println("Tree is no longer AVL.");
                            //System.out.println(S);
                            if(pathNode.balance < -1) {
                                    /**
                                     * TODO: call either the rotateWithLeftChild
                                     * or doubleRotateWithLeftChild methods.
                                     * Then, push appropriate AVLNodes into
AVLStack
                                     * to update their balance and height fields.
                                     */
                                    if(parent.iData > newNode.iData) {

                                            //System.out.println("single " +
pathNode.iData);
                                            AVLNode node =
rotateWithLeftChild(pathNode);
                                            if(!S.isEmpty()) {
                                                    AVLNode parentNode = S.pop();
                                                    parentNode.leftChild = node;
                                                    S.push(parentNode);
                                            }
                                            S.push(node);
```

```
                                                } else {
                                                        //System.out.println("double " +
pathNode.iData);
                                                        AVLNode node =
doubleRotateWithLeftChild(pathNode);
                                                        S.push(node);
                                                }
                                        }
                                }
                        } // end while stack is non-empty
                        root = pathNode;
                }  // end else not root
                //System.out.println();
        }  // end insert()

//AVLTreeTester.java

public class AVLTreeTester {

        public static void main(String[] args) {
                AVLTree at1 = new AVLTree();
                at1.insert(20);
                at1.insert(15);
                at1.insert(10);
                at1.displayTree();

                System.out.println();

                AVLTree at2 = new AVLTree();
                at2.insert(20);
                at2.insert(15);
                at2.insert(17);
                at2.displayTree();

                System.out.println();

                AVLTree at3 = new AVLTree();
                at3.insert(20);
                at3.insert(15);
```

```
                at3.insert(17);
                at3.insert(10);
                at3.insert(8);
                at3.displayTree();
        }

}

//output
Tree is no longer AVL.
Tree is no longer AVL.
15
10
20
------

Tree is no longer AVL.
17
15
20
------

Tree is no longer AVL.
Tree is no longer AVL.
Tree is no longer AVL.
17
10
8
15
20
------
```