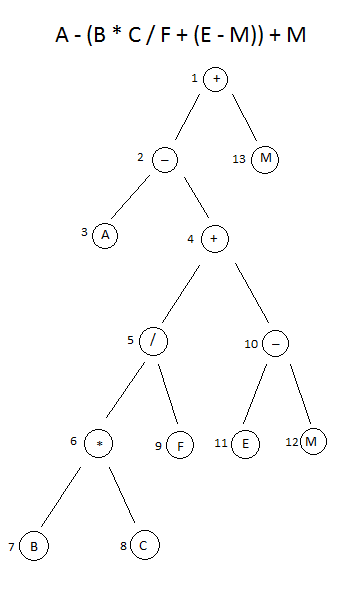
CS146Setion8Lab4 Name: Michael Huang Due Date: March 2nd, 2016

Ex-1: What is the expression tree for: A – (B \* C / F + (E – M)) + M?

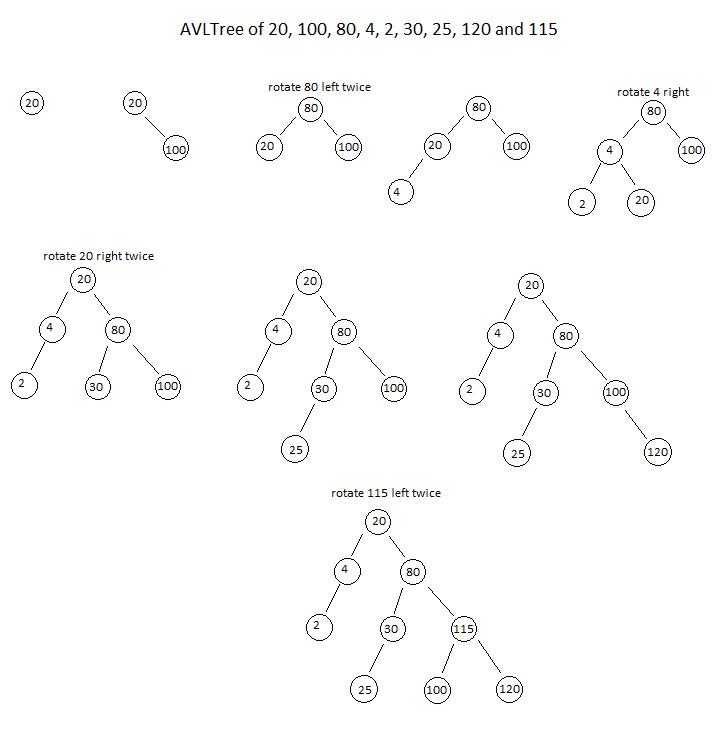
Using the expression tree, show the prefix expression for it.



Using preorder travel, the prefix expression is: + - A + / \* B C F – E M M.

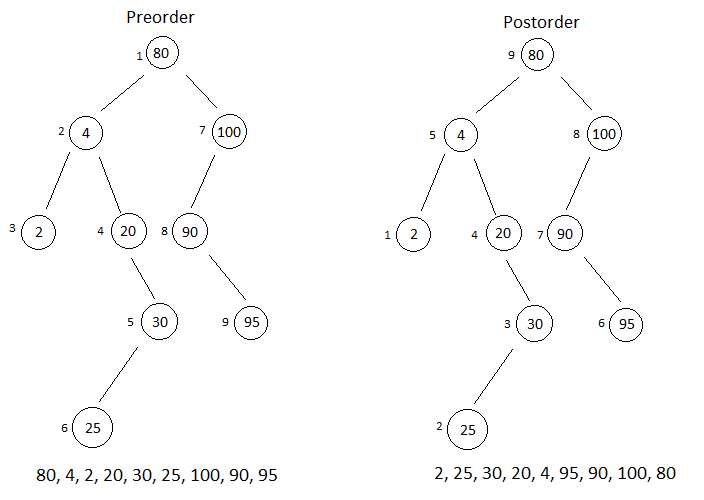
This was drawn in Microsoft Paint.

Ex-2: Show the result of inserting 20, 100, 80, 4, 2, 30, 25, 120 and 115 into an initially empty AVL tree. List the type of rotations needed and when they happen. Show the resulting AVL tree.



This was drawn in Microsoft Paint.

Ex-3: Assuming the preorder traversal is in the following order, show the postorder traversal of 80, 4, 2, 20, 30, 25, 100, 90, 95.



This was drawn in Microsoft Paint.

Ex-4: Write the implementation code for a BST. Include recursive methods to find the number of nodes, number of leaves and height of the tree. Test your code by entering values given in exercise 2.

Code:

public class BinarySearchTree<T extends Comparable<T>> {

private BinaryNode<T> root;

@SuppressWarnings("hiding")

class BinaryNode<T> {

T data;

BinaryNode<T> left;

BinaryNode<T> right;

public BinaryNode(T data) {

this.data = data;

left = null;

right = null;

}

}

public BinarySearchTree() {

root = null;

}

public void add(T object) {

BinaryNode<T> adding = new BinaryNode<T>(object);

if (root == null) {

root = adding;

} else {

addNode(root, adding);

}

}

public void addNode(BinaryNode<T> root, BinaryNode<T> adding) {

int comparison = adding.data.compareTo(root.data);

if (comparison < 0) {

if (root.left == null) {

root.left = adding;

} else {

addNode(root.left, adding);

}

} else if (comparison > 0) {

if (root.right == null) {

root.right = adding;

} else {

addNode(root.right, adding);

}

}

}

public int getLeafCount() {

if (root == null) {

return 0;

} else {

return leaves(root);

}

}

public int leaves(BinaryNode<T> root) {

if (root.left == null && root.right == null) {

return 1;

} else {

if (root.left == null && root.right != null) {

return leaves(root.right);

} else if (root.left != null && root.right == null) {

return leaves(root.left);

} else {

return leaves(root.left) + leaves(root.right);

}

}

}

public int getHeight() {

return height(root);

}

private int height(BinaryNode<T> node) {

if (node == null) {

return 0;

} else {

return 1 + Math.max(height(node.left), height(node.right));

}

}

public int getNodeCount() {

return nodeCount(root);

}

private int nodeCount(BinaryNode<T> node) {

if (node == null) {

return 0;

} else {

return 1 + nodeCount(node.left) + nodeCount(node.right);

}

}

public T findMin() {

return findMin(root).data;

}

private BinaryNode<T> findMin(BinaryNode<T> root) {

if (root != null) {

while (root.left != null) {

root = root.left;

}

}

return root;

}

public T findMax() {

return findMax(root).data;

}

private BinaryNode<T> findMax(BinaryNode<T> root) {

if (root != null) {

while (root.right != null) {

root = root.right;

}

}

return root;

}

public boolean contains(T object) {

return contains(object, root);

}

private boolean contains(T object, BinaryNode<T> root) {

if (root == null) {

return false;

} else {

int comparison = object.compareTo(root.data);

if (comparison < 0) {

return contains(object, root.left);

} else if (comparison > 0) {

return contains(object, root.right);

} else {

return true;

}

}

}

public static void main(String[] args) {

BinarySearchTree<Integer> bst = new BinarySearchTree<Integer>();

bst.add(20);

bst.add(100);

bst.add(80);

bst.add(4);

bst.add(2);

bst.add(30);

bst.add(25);

bst.add(120);

bst.add(115);

System.out.println("Using exercise 2's inputs of:\n20, 100, 80, 4, 2, 30, 25, 120, 115.\n");

System.out.println("Leaves: " + bst.getLeafCount());

System.out.println("Tree height: " + bst.getHeight());

System.out.println("Number of nodes: " + bst.getNodeCount());

System.out.println("Max: " + bst.findMax());

System.out.println("Min: " + bst.findMin());

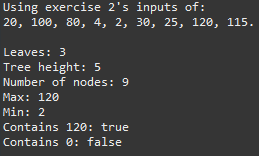
System.out.println("Contains 120: " + bst.contains(120));

System.out.println("Contains 0: " + bst.contains(0));

}

}

Console output:



Ex-5: Exercise 4.46 page 166. Test your code by having two similar and two not similar binary trees.

Code (add to code from Ex-5):

public boolean compareTrees(BinarySearchTree<T> tree) {

return similar(root, tree.root);

}

private boolean similar(BinaryNode<T> root1, BinaryNode<T> root2) {

if (root1 == null && root2 == null) {

return true;

} else if (root1 != null && root2 != null) {

return similar(root1.left, root2.left) && similar(root1.right, root2.right);

} else {

return false;

}

}

public static void main(String[] args) {

BinarySearchTree<Integer> bst1 = new BinarySearchTree<Integer>();

bst1.add(20);

bst1.add(100);

bst1.add(80);

bst1.add(4);

bst1.add(2);

bst1.add(30);

bst1.add(25);

bst1.add(120);

BinarySearchTree<Integer> bst2 = new BinarySearchTree<Integer>();

bst2.add(10);

bst2.add(90);

bst2.add(70);

bst2.add(2);

bst2.add(1);

bst2.add(20);

bst2.add(15);

bst2.add(110);

System.out.println("Tree 1: 20, 100, 80, 4, 2, 30, 25, 120.");

System.out.println("Tree 2: 10, 90, 70, 2, 1, 20, 15, 110.");

System.out.println("Similar: " + bst1.compareTrees(bst2));

bst1 = new BinarySearchTree<Integer>();

bst1.add(10);

bst1.add(90);

bst1.add(70);

bst1.add(2);

bst2 = new BinarySearchTree<Integer>();

bst2.add(20);

bst2.add(100);

bst2.add(80);

bst2.add(99);

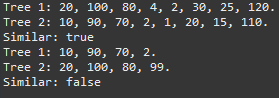
System.out.println("Tree 1: 10, 90, 70, 2.");

System.out.println("Tree 2: 20, 100, 80, 99.");

System.out.println("Similar: " + bst1.compareTrees(bst2));

}

Console output:



The method is O(2n), because the similar method is recursive and it makes 2 additional calls of itself recursively if both roots are not null, and the number of calls grow exponentially.