Report of Midterm Exam

IF184401 Design & Analysis of Algorithms (H)

MM Irfan Subakti & Nurul Izzatil Ulum

Name : Sandyatama Fransisna Nugraha

NRP : 5025211196

Class : H

**Question :**

1. Download the working file and the task is update and continue writing codes for the files DAA1.java and DAA2.java. Run the code by running the DAA1Test.java and DAA2Test.java

This code is implementing a BST Tree Algorithm and AVL Tree Algorithm

1. Create a function namely isBST() from DAA1.java which has recursive function inside of it. It used to check whether a tree or not.
2. Create a function namely printDescending() from DAA1.java which has a recursive function that used to receives an input of a BST t (where t is MyTree and integer) that can print the values of t in descending order
3. Create a function namely max() from DAA1.java which has a recursive function that used to receives an input of a BST t (where t is MyTree and integer) that can get the maximum value of the t’s values.
4. Create a function namely isHeightBalanced() from DAA2.java which has a recursive function that used to receives an input of MyTree t to check whether t has a balanced height (ALV tree condition)
5. Create a function namely InsertHB() from DAA2.java which has a recursive function that used to receives an input of int n and MyTree t to can insert n into twhile it keeps preserving the AVL condition.
6. Create a function namely Private Static MyTree rebalancceForRight(MyTree t) and update from DAA2.java
7. Create a function namely Private Static MyTree rebalancceForLeft(MyTree t) and update from DAA2.java
8. Create a function namely deleteHB() from DAA2.java which receives the inputs of MyTree t and int x. That can delete x from t while it keeps preserving the AVL condition

**Answer**

**Source Code**

1. public static boolean isBST(MyTree t) {
2. // Write your codes in here
3. return isBST(t, Integer.MIN\_VALUE, Integer.MAX\_VALUE);
4. // Write your codes in here
5. }

**Analysis**

This recursive function is used to checks whether a given binary tree is valid of Binary Search Tree principle or not. The function takes in three parameter , which is t (the current node), Integer.MIN\_VALUE (minimum possible value that a node can hold) , and Integer.MAX\_VALUE (maximum possible value that a node can hold).

First of all, the function check if the node now is null or not. If its null, then return true since empty tree is valid BST by definition. Then, the function is check the current node now is between MIN\_VALUE or MAX\_VALUE. If its doesn’t, then the function return false. Indicating that the tree is not a valid BST. It also check for every node that the left subtree must have values less than the value in the right subtree.

    private static boolean isBST(MyTree t, int lowerBound, int upperBound) {

        // Write your codes in here

        if (t == null) {

            return true;

        }

        int value = t.getValue();

        if (value < lowerBound || value > upperBound) {

            return false;

        }

        return isBST(t.getLeft(), lowerBound, value - 1) && isBST(t.getRight(), value + 1, upperBound);

        // Write your codes in here    }

    }

This recursive function that checks if a given binary tree is a valid binary search tree (BST). The function takes three parameters is MyTree t (the root node of the binary tree), int lowerBound

(the lower bound for the node values), and int upperBound(the upper bound for the node values).

The function starts by checking if the current node ‘t’ is null, which means it is at the end of a branch and returns true since there is no violation of the BST property. If the current node ‘t’ has a value outside of the given ‘lowerBound’ and ‘upperBound, then it means it is violating the BST property and return false. If the current node ‘t’ passes the above two checks, then the function makes a recursive calls to its left and right child nodes, with updated ‘lowerBound’ and ‘upperBound’ values. The ‘lowerBound’ for the left child node remains the same, while the ‘upperBound’ is updated to ‘value-1’ (since all node in the left subtree should be less than the current node ‘t’), and for the right child node , the ‘lowerBound’ is updated to ‘value+1’ and the ‘upperBound’ remains the same (since all node in the left subtree should be greater than the current node ‘t’). And the final, the function return the logical AND operation of the recursive calls made to the left and right child nodes, which return true only if both of the subtrees are valid BST Tree.

**2).**

**Source Code**

   public static void printDescending(MyTree t) {

        // Write your codes in here

        if (t.getEmpty() == false) {

          printDescending(t.getRight());

          System.out.print(t.getValue() + " ");

          printDescending(t.getLeft());

        }

        // Write your codes in here

    }

**Analysis**

This function is used to print the values of each nodes of a binary search tree in descending order. It has a paramater a MyTree which is the root node of the binary tree. The function start with check if the current node is not empty. If it is not, then make a order to print descending value of right subtree. And calling t.GetValue( ) Method from MyTree.java followed by a space. And the final is print descending value of left subtree.

**3).**

**Source Code**

  public static int max(MyTree t) {

        // Write your codes in here

         if (t.getEmpty()) {

             return Integer.MIN\_VALUE;

         }

         int maxLeft = max(t.getLeft());

         int maxRight = max(t.getRight());

         return Math.max(t.getValue(), Math.max(maxLeft, maxRight));

        // Write your codes in here

    }

**Analysis**

This recursive function is used to find the maximum value in BST Tree. It started by calling t.getEmpty( ) which is checking the tree if is empty or not. If it’s empty, then it will returns the smallest possible integer value because there is no maximum value in a empty tree.

If it’s no empty, then recursively call the max() function on the left or right subtree to find the maximum value in each of those subtrees. Then, it’s return the maximum value of the current node, maximum value in the left subtree, and maximum value in the right subtree

4).

**Source Code**

public static boolean isHeightBalanced(MyTree t) {

        if (t.getEmpty()) {

            return true;

        }

        else {

             int leftHeight = MyTreeOps.height(t.getLeft());

             int rightHeight = MyTreeOps.height(t.getRight());

             int diff = Math.abs(leftHeight - rightHeight);

             return diff <= 1 && isHeightBalanced(t.getLeft()) && isHeightBalanced(t.getRight());

        }

    }

**Analysis**

This recursive function is used to check whether a given binary tree is height-balanced or not. A binary tree is height-balanced if the difference between the heights of its left and rightsubtree is not more than one for every node in the tree.

The function is start with calling t.getEmpty( ) to check if the tree is empty or not. If its empty, then it is height balanced by definition and return true statement. If it’s not then the height of the left and right subtree are calculated using the height() method. Inside the height( ) method using parameter whicch is called the getLeft() method and getRight() method. The diff variable is contain the result of these leftHeight() and rightHeight() using mathematical operation. Then the final, the code return true if the diff is less than or equal to 1, and both the left and right subtree as also height balanced (recursive called isHeightBalanced() ).

5).

**Source Code**

    public static MyTree insertHB(int n, MyTree t) {

        if (t.getEmpty()) {

            return new MyTree(n, emptyTree, emptyTree);

        }

        else if (n < t.getValue()) {

            t = new MyTree(t.getValue(), insertHB(n, t.getLeft()), t.getRight());

        }

        else if (n > t.getValue()) {

            t = new MyTree(t.getValue(), t.getLeft(), insertHB(n, t.getRight()));

        }

            t = rebalanceForLeft(t);

            t = rebalanceForRight(t);

        return t;

    }

**Analysis**

This recursive function is used to insert a new node with value n into a BST Tree represented by the MyTree t while ensuring that the resulting tree is height balanced.

The function is start with calling t.getEmpty( ) to check if the tree is empty or not. If its empty, then a new MyTree objecct is created with value n, and empty left and right subtrees. If the n is less than or gretaer than the current node’s value, then make a new MyTree object which has a parameter node value, getLeft() method, getRight() method, and recursive call method insertHB().

After inserting the new node, the function then checks whether the tree is height-imbalanced on the left or right subtree and applies a rebalancing operation if necessary. The rebalanccing operations will adjust the height of the left and right subtrees so that the difference in height between them is at most one, which is the definition of a height-balanced tree.

Finally, the function return the result of new updated tree inside the MyTree object t which now includes the newly inserted node and any rebalancing operations that were applied.

6).

**Source Code**

    private static MyTree rebalanceForLeft(MyTree t) {

        // Write your codes in here

        if(MyTreeOps.height(t.getLeft()) <= (MyTreeOps.height(t.getRight()) + 1)) {

            return t;

        }

        else {

            //Double node is checked

            MyTree leftleftNode = t.getLeft().getLeft();

            MyTree leftrightNode = t.getLeft().getRight();

                if(MyTreeOps.height(leftleftNode) > MyTreeOps.height(t.getRight())) {

                    return new MyTree(t.getLeft().getValue(), leftleftNode, new MyTree(t.getValue(),

                            leftrightNode, t.getRight()));

                }

                else {

                    return new MyTree(leftrightNode.getValue(), new MyTree(t.getLeft().getValue(), t.getLeft().getLeft(), leftrightNode.getLeft()),

                            new MyTree(t.getValue(), leftrightNode.getRight(), t.getRight()));

                }

            }

        // Write your codes in here

    }

**Analysis**

This recursive function is used to implements a method to rebalance an unbalanced binary search tree (BST) that is leaning to the left side. This function takes a BST t as input and returns a rebalanced version of the tree. The rebalancing is done by performing a left rotation on the unbalanced node or subtree, as described in the previous explanation.

The code first checks if the height of the left subtree is no more than one greater than the height of the right subtree. If this is the case, the tree is already balanced and it can be returned as is. If the tree is unbalanced towards the left, the code checks whether a double left rotation is needed. For this, it retrieves the left-left and left-right nodes of the left subtree of t.

If the height of the left-left node is greater than the height of the right subtree of t, a double left rotation is performed. This involves first performing a left rotation on the left subtree of t, and then performing a left rotation on t. In the first rotation, a new tree is created with the value of the left subtree as the value of the new root, and with the left-left node as the new left subtree and the left-right node as the new right subtree.

In the second rotation, a new tree is created with the value of the left-right node as the value of the new root, and with a new left subtree consisting of the old left subtree of t and the left subtree of the left-right node, and a new right subtree consisting of the right subtree of the left-right node and the right subtree of t.

If the height of the left-left node is not greater than the height of the right subtree of t, a single left rotation is performed, as in the previous code. The new tree is created by making the left subtree of the old node the left subtree of the new node, and making the right subtree of the left subtree the right subtree of the new node. The value of the left subtree becomes the value of the new node, and the value of the old node becomes the value of the right subtree.

7).

**Source Code**

private static MyTree rebalanceForRight(MyTree t) {

        // Write your codes in here

        if(MyTreeOps.height(t.getRight()) <= (MyTreeOps.height(t.getLeft()) + 1)) {

            return t;

        }

        else {

            MyTree rightleftNode = t.getRight().getLeft();

            MyTree rightrightNode = t.getRight().getRight();

                if (MyTreeOps.height(rightrightNode) > MyTreeOps.height(t.getLeft())) {

                    return new MyTree(t.getRight().getValue(), new MyTree(t.getValue(), t.getLeft(), rightleftNode), rightrightNode);

                }

                else {

                    return new MyTree(rightleftNode.getValue(), new MyTree(t.getValue(), t.getLeft(), rightleftNode.getLeft()),

                            new MyTree(t.getRight().getValue(), rightleftNode.getRight(), t.getRight().getRight()));

                }

            }

        // Write your codes in here

    }

**Analysis**

This recursive function is used to implements a method to rebalance an unbalanced binary search tree (BST) that is leaning to the left side. The input to the method is the root node of the BST, denoted by t, and the output is a rebalanced version of the BST. The code first checks if the height of the left subtree of t is no more than one greater than the height of its right subtree. If this condition is true, then the tree is already balanced and the original tree is returned.

If the tree is unbalanced towards the left, then the code performs a left rotation to rebalance the tree. To perform a left rotation, the code retrieves the left and right subtrees of t, as well as the left-left and left-right nodes of the left subtree. If the height of the left-left node is greater than the height of the right subtree of t, then a single left rotation is performed.

The new root node is created by making the left subtree of the old root node the left subtree of the new root node, and making the right subtree of the left subtree the right subtree of the new root node. The value of the left subtree becomes the value of the new root node, and the value of the old root node becomes the value of the right subtree.

If the height of the left-left node is not greater than the height of the right subtree of t, then a double left rotation is performed. This involves first performing a left rotation on the left subtree of t, and then performing a left rotation on t. The new left subtree of t becomes the right subtree of the new left node, and the new right subtree of t becomes the right subtree of the new right node.

In this updated code, the left and right subtrees of t are not retrieved separately but the left-left and left-right nodes are obtained directly from t. Additionally, the node containing the value of t is reconstructed by creating a new MyTree object and passing in the values of the left and right subtrees.

8).

**Source Code**

public static MyTree deleteHB(MyTree t, int x) {

        // Write your codes in here

        if(t.getEmpty()) {

            return t;

        }

        else {

            if(x > t.getValue()) {

                MyTree newRight = deleteHB(t.getRight(), x);

                return rebalanceForLeft(new MyTree(t.getValue(), t.getLeft(), newRight));

            }

            else if(x < t.getValue()) {

                MyTree newLeft = deleteHB(t.getLeft(), x);

                return rebalanceForRight(new MyTree(t.getValue(), newLeft, t.getRight()));

            }

            else {

                if(t.getLeft().getEmpty()) {

                    return t.getRight();

                }

                else if(t.getRight().getEmpty()) {

                    return t.getLeft();

                }

                else {

                    int last = max(t.getLeft());

                    return rebalanceForRight(new MyTree(last, deleteHB(t.getLeft(), last), t.getRight()));

                }

            }

        }

        // Write your codes in here

    }

**Analysis**

This recursive function is used to implements a method to delete a node from a height-balanced binary search tree (BST). The input to the method is the root node of the BST, denoted by t, and the value of the node to be deleted, denoted by x. The output is the root node of the updated BST after the deletion has been performed.

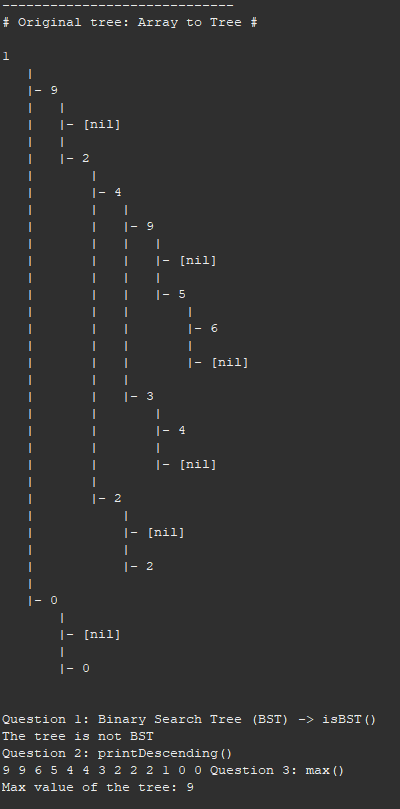
The code first checks if t is an empty node. If t is empty, then the method simply returns t . If t is not empty, then the code checks if x is greater than the value of t. If this is the case, then the method recursively calls itself on the right subtree of t, passing in x as the second argument. The return value of this call is then used to create a new node with the same value as t, but with its left subtree unchanged and its right subtree set to the return value of the recursive call. This new node is then passed to a rebalancing method rebalanceForLeft to ensure that the BST remains height-balanced.

If x is less than the value of t, then the method recursively calls itself on the left subtree of t, passing in x as the second argument. The return value of this call is then used to create a new node with the same value as t, but with its left subtree set to the return value of the recursive call and its right subtree unchanged. This new node is then passed to a rebalancing method rebalanceForRight to ensure that the BST remains height-balanced.

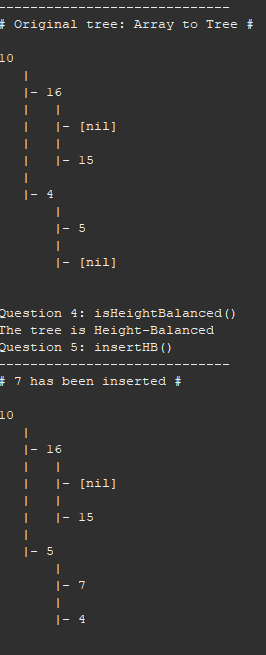
If x is equal to the value of t, then the method checks if the left subtree of t is empty. If it is, then the right subtree of t is returned as the new root of the BST. If the right subtree of t is empty, then the left subtree of t is returned as the new root of the BST. If neither the left nor right subtree of t is empty, then the method recursively finds the maximum value in the left subtree of t using a helper method max. The value of the maximum node is then used to create a new node with its value set to the maximum value, its left subtree set to the return value of a recursive call to delete the maximum node from the left subtree of t, and its right subtree set to the right subtree of t. This new node is then passed to a rebalancing method rebalanceForRight to ensure that the BST remains height-balanced.

}

**DAA1**

****

**DAA2**

****

