Khuyen Le Thi Minh s5128 – ASD 4

TASK DESCRIPTION:

Design an algorithm which tests if an input binary tree is also balanced tree (AVL type). Prove algorithm's correctness and estimate its complexity.

ALGORITHM:

As not every binary tree is balanced binary search tree (AVL) we will test both if the binary tree fulfil the requirement for BST as well as balanced tree.

```
//We assume that the tree is designed as linked list of nodes, where each node contains a value of node label and addresses/ link to its left and right child.
struct node
{
        int data;
        struct node leftNode;
        struct node rightNode;
};

//Algorithm to test if the tree is balanced. It will take 2 variables: a struct node (which is root of the tree) and an interger which define the height at that node
boolean isBalanced(struct node root, int height)
{
  // We declare 2 new variables: leftHeight which will be the height of left subtree and rightHeight which will be the height of right subtree and we set them as default 0 as we will check from bottom left node(leaf) and going up.
        int leftHeight = rightHeight = 0;
  // we declare 2 boolean valrables which will represent the result of recursive function on left side and right side tree and set them as false in default.
         boolean leftTree =  rightTree = false;
// initiatively set the height of node as -1 since we will start from leaf and at each time we increase the height by 1.
        height = -1;
 // base condition for the algorithm, we will start the if loop only if the tree is not empty.
        If (root != null) then
        {
  // we are calling recursive functions for checking the left and after that right side of the subtrees. (The recursive function will iterate through levels of struct left and right nodes in node.)
                leftTree = isBalanced(root->leftNode, &leftHeight);
                rightTree = isBalanced(root->rightNode, &rightHeight);
  //We are calculating the height at the current node, we take the bigger value between height of left and tight subtrees and increase it by 1 since we will go up to next level for comparison.
                height = (leftHeight > rightHeight? leftHeight: rightHeight) + 1;

  // Now we're checking if the difference between the height of left and right side subtrees is greater than 1, if it's then the tree is not balanced
                If ((leftHeight - rightHeight > 1) || (rightHeight - leftHeight > 1)) then
                {
                        return false;
                }

  // If the differences between height of  left and right side of subtree is not greater than 1, then we will return the logical value of left AND right side of subtrees (it will be true only if both of the values are true).
                Else then
                {
                        return leftTree && rightTree;
                }fi;
        }fi;
// If the tree is empty then we return true.
        return true;
}
//Algorithm to test if the tree is a Binary Search Tree. It will take 1 variables: a struct node (which is root of tree). We will use the DFS InOrder traversal.
boolean isBST(struct node root)
{
//Since with checking if the tree is BST is better to travel from root we declare at the beginning a new struct variable prev and set it on default as null to store the previous node for comparison.
```

```
        struct node prev = null;
//We declare 2 boolean variables that will store the result of recursive function on left and right side of subtree and set them to false as default.
        boolean leftTree =  rightTree = false;
  //The base condition, we will start the if loop only if the tree is not empty.
        if (root != null) then
        {
//We start calling recursive function for left side of tree.
                leftTree  = isBST(root->leftNode);
 //We check if the previous node is null (since we set it as null on default) and check if the value at current node is greater than the previous one (it have to be greater), if not then return false.
                if (prev != null && root->data <= prev->data) then
                {
                        return false;
                }fi;
 //We assign the current node to the previous one.
                prev = root;
 //We start calling recursive function for right side of tree.
                rightTree = isBST(root->rightNode);
// We return the logical value of left AND right side of subtrees (it will be true only if both of the values are true).
                return leftTree && rightTree;
        }fi;
 // If the tree is empty then we return true.
        return true;
}
//Algorithm to test if the tree is AVL
boolean isAVL(struct node root, int height)
{
// We return the logical value of isBalanced  AND isBST (it will be true only if both of the values are true).
        return isBST(root) && isBalanced(root, height);
}
```

Justification of an algorithm's correctness:

The specification of the algorithm isAVL becomes a pair <WP, WK>, where WP = (The binary tree where there is a finite number of nodes and each node has at most 2 children) and WK = (the logical value if the tree is an AVL one).

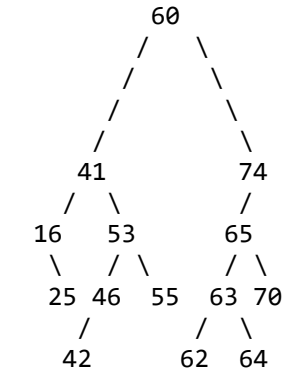
Stop condition of the algorithm – for both sub-algorithm we have base condition where the loop will start only if the tree is not empty. In both sub-algorithm, we called the recursive functions on left and right side nodes of the tree. Since as at our WP condition we have the binary tree with finite number of nodes, so our recursive functions will iterate till to leaf and we go back to un-pause the action and at the end return the result. Thus, both the algorithm will end. Thus, the algorithm stops for any data satisfying the initial condition WP.

Partial correctness of the algorithm – for the initial condition WP = (The binary tree were there is a finite number of nodes and each node has at most 2 children) and WK = (the logical value if the tree is an AVL one).

In the first algorithm to check if the tree is a balanced one, we go from the most left side leaf. We calculate between number of edges at the longest path from left leaf to the current node and the longest path from right leaf to the current node. If the different is greater than 1 then we return false, if the different is 0 or 1 then we calculating the height of the current node by taking the number of edges of the longest path from leaf to the node (we start at the -1 and since we check from the leaf, after each checking we'll increase the height of node by 1 so we'll start with height = 0 for the leaf nodes). The result will be true only if all the results from left sub-trees and right sub-trees are true.

In the second algorithm to check if the binary tree is a binary search one we start from root and we go with DFS InOrder traversal. We will go from left side -> root -> right side and with the recursive calling function we can go deeper till the leaf and from there we'll check if the value at current node is greater than the previous one.

The Binary Tree:




```
boolean isBalanced(struct node root, int height)
{
        int leftHeight = rightHeight = 0;
        boolean leftTree =  rightTree = false;
        height = -1;
//We start the if loop since the tree is not empty
        If (root != null) then
        {
//*calling the recursive function for the left side of the tree. We start at root (60) after that go to (41) -> (16) at hear we pause and got to recursive function calling right side.
//**We go back and un-pause function for node (16)
//***We go back to node (41) and go to next recursive function
//****We go to nodes (46) -> (42) now we go to calculations
//*****We un-pause function for node (46)
//******We go back to (53) and go to next recursive function
//*******We go back to node (53)
//********We go back to (41)
//*********We go back to (60) and go to next recursive function
//**********We go to node (65)-> (63)-> (62)
//***********We go back to (63) and go to next recursive function
//************We go back to (63)
//*************We go back to (65) and go to next recursive function
//**************We go back to (65)
//**************We go back to (74)
//***************We go back to (60)

                leftTree = isBalanced(root->leftNode, &leftHeight);
//*recursive function calling for right side and we pick from previous function so next node is (25). We stop here for a moment to calculate the height and the difference between height of left side and right side (diff)
//*** we go to node (53) and since there are still other nodes at left side we don't calculate anything and go back to previous function.
//******We go to node (55)
//*********We go to node (74) and since there are still other nodes at left side we don't calculate anything and go back to previous function.
//***********We go to (64)
//*************We go to (70)

                rightTree = isBalanced(root->rightNode, &rightHeight);
//*Since the height for both right and left side for (25) is the default = -1 so we increase it by 1 and now it's 0
//**We calculate the height for node (16). Since the height at left side is -1 and right side is 0, we take 0 and increase it by 1 so now we have 1.
//****We calculate the height for node (42) = 0 (since it's a leaf)
//***** height (46) = 0+1 = 1
//******height (55) = 0 (it's a leaf)
//*******height (53) = 1+1=2
//********height (41) = 2+1=3
//**********height (62) = 0 (it's a leaf)
//***********height (64) = 0(it's a leaf)
//************height (63) = 0+1=1
```

```
//************height (70) =0 (it's a leaf)
//*************height (65) =1+1=2
//*************height (74) =2+1=3
//**************height (60) =3+1=4

                        height = (leftHeight > rightHeight? leftHeight: rightHeight) + 1;
//*Wee check if the diff between height of right side and left side of (25) is greater than 1. Since both heights are -1 so it's 0 we return true.
//** Wee check if the diff between height of right side and left side of (16) =-1-0=-1. And the logical value for the right side sub-tree (node 25) is true then we return true.
//****the diff for node (42) is 0 since it's leaf and we return true.
//*****the diff for (46) = 0+1 = 1 and the logical value of left sub-tree is true so we return true.
//******the diff (55) = -1+1=0 and we return true.
//*******diff (53) =1-0=1, both logical values for sub-tree are true, we return true.
//********diff (41) =1-2=-1, both logical values for sub-tree are true, we return true.
//********** diff (62) is 0 since it's leaf and we return true.
//*********** the (64) is 0 since it's leaf and we return true.
//************diff (63-9 = 0-0=0, both logical values for sub-tree are true, we return true.
//************* diff (70) is 0 since it's leaf and we return true.
//*************diff (65) =1-0=1, both logical values for sub-tree are true, we return true.
//*************diff (74) =2-0=2, we return false.
//**************diff (60) =3-3=0, however since the right side sub-tree return false then we return false.

                        If ((leftHeight - rightHeight > 1) || (rightHeight - leftHeight > 1)) then
                        {
                                return false;
                        }
                        Else then
                        {
                                return leftTree && rightTree;
                        }fi;
        }fi;
        return true;
}
//The tree is not balanced one.


boolean isBST(struct node root)
{
        struct node prev = null;
        boolean leftTree =  rightTree = false;
  //We start the if loop since the tree is not empty
        if (root != null) then
        {
//*We start at the root (60) and call for recursive function -> (41)-> (16) we go to comparison and since prev is still null so we have leftTree(16) = true and set prev = 16
//***we go back to (41) and since prev = 25 we return leftTree (41) = true and set prev = 41
//****we go further left-> (53)-> (46)-> (42). Since prev = 41 we have leftTree(42)=true and prev = 42
//****at node (46) prev =42 so we return leftTree (46) = true and set prev = 46
//*****we go back to (53) prev = 46 we return leftTree (53) = true and set prev = 53
//******* we are at (60) and prev = 55 so we return leftTree (60) = true and set prev = 60
//********we go to (74)-> (65)-> (63)-> (62) ate (62) since prev = 60 we return leftTree (62)=true and set prev = 62
//*********we go back to (63) since prev = 62 we return leftTree (63) = true and set prev = 63
//**********we go back to (65) and prev = 64 so we return leftTree (65) = true and set prev = 65
//*********we go back to (74), prev = 70 so leftTree(74)=true and we set prev =74, since we went through all nodes so we go further
                        leftTree  = isBST(root->leftNode);
                        if (prev != null && root->data <= prev->data) then
                        {
                                return false;
                        }fi;
                        prev = root;
 //**we call for (25) and since prev = 16 we have rightTree (25) =true and we set prev = 25
//******we call for (55) and prev=53 so we return rightTree (55) =true and we set prev = 55
//**********we go to (64) since prev = 63 we return rightTree (64) =true and set prev = 64
```

```
//***********we go to (70), prev = 65 so we return rightTree (70) = true and set prev = 70
                    rightTree = isBST(root->rightNode);
//since both leftTree and rightTree are true we return true
                    return leftTree && rightTree;
          }fi;
          return true;
}
//The tree is a binary search one.

boolean isAVL(struct node root, int height)
{
//isBST = true and isBalanced = false we return false
          return isBST(root) && isBalanced(root, height);
}
//The tree is not AVL one.
```

Hence the result is that the tree is not AVL one.

Because the isAVL algorithm is partially correct in terms of specification <WP,WK> and we have shown the stop condition for this algorithm, it is also absolutely correct with respect to the considered specification.

Estimation of algorithm's complexity:

Time complexity – assuming that the dominant operation in the isAVL is the natural numbers arithmetic operations and comparison, then T(n) = n + n = 2n so we have O(n)

Space complexity – the isAVL algorithm only uses auxiliary variables that are independently of the value of the argument n, therefore S should be S(n) = 1 so we have O(1)