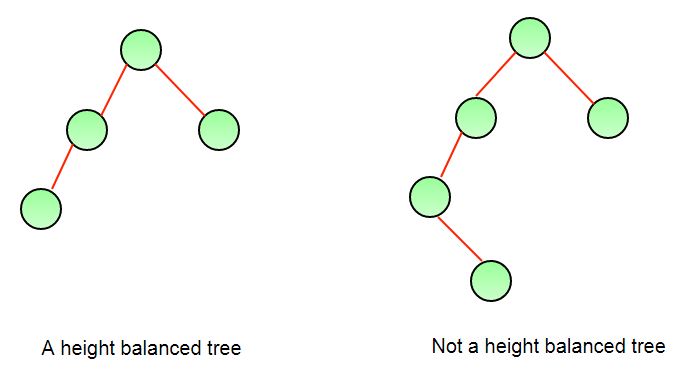
***Check for Balanced Binary Tree***

*A****height balanced binary tree****is a binary tree in which the height of the left subtree and right subtree of any node does not differ by more than 1 and both the left and right subtree are also height balanced.*

In this article, we will look into methods how to determine if given Binary trees are height-balanced

***Examples:****The tree on the left is a height balanced binary tree. Whereas the tree on the right is not a height balanced tree. Because the left subtree of the root has a height which is 2 more than the height of the right subtree.*



**Naive Approach:** To check if a tree is height-balanced:

*Get the height of left and right subtrees using****dfs****traversal. Return true if the difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.*

Below is the implementation of the above approach.

C++Java

/\* Java program to determine if binary tree is

height balanced or not \*/

/\* A binary tree node has data, pointer to left child,

and a pointer to right child \*/

class Node {

int data;

Node left, right;

Node(int d)

{

data = d;

left = right = null;

}

}

class BinaryTree {

Node root;

/\* Returns true if binary tree with root as root is

\* height-balanced \*/

boolean isBalanced(Node node)

{

int lh; /\* for height of left subtree \*/

int rh; /\* for height of right subtree \*/

/\* If tree is empty then return true \*/

if (node == null)

return true;

/\* Get the height of left and right sub trees \*/

lh = height(node.left);

rh = height(node.right);

if (Math.abs(lh - rh) <= 1 && isBalanced(node.left)

&& isBalanced(node.right))

return true;

/\* If we reach here then tree is not height-balanced

\*/

return false;

}

/\* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION \*/

/\* The function Compute the "height" of a tree. Height

is the number of nodes along the longest path from

the root node down to the farthest leaf node.\*/

int height(Node node)

{

/\* base case tree is empty \*/

if (node == null)

return 0;

/\* If tree is not empty then height = 1 + max of

left height and right heights \*/

return 1

+ Math.max(height(node.left),

height(node.right));

}

public static void main(String args[])

{

BinaryTree tree = new BinaryTree();

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

tree.root.left.left.left = new Node(8);

if (tree.isBalanced(tree.root))

System.out.println("Tree is balanced");

else

System.out.println("Tree is not balanced");

}

}

**Output**

Tree is not balanced

**Time Complexity:** O(n^2) in case of full binary tree.  
**Auxiliary Space**: O(n) space for call stack since using recursion

**Efficient implementation:** Above implementation can be optimized by

*Calculating the height in the same recursion rather than calling a height() function separately.*

* For each node make two recursion calls – one for left subtree and the other for the right subtree.
* Based on the heights returned from the recursion calls, decide if the subtree whose root is the current node is height-balanced or not.
* If it is balanced then return the height of that subtree. Otherwise, return **-1** to denote that the subtree is not height-balanced.

Below is the implementation of the above approach.

C++Java

// Java code to implement the approach

import java.io.\*;

import java.lang.\*;

import java.util.\*;

// Class to define the tree node

class Node {

int key;

Node left;

Node right;

Node(int k)

{

key = k;

left = right = null;

}

}

class GFG {

// Driver code

public static void main(String args[])

{

Node root = new Node(10);

root.left = new Node(5);

root.right = new Node(30);

root.right.left = new Node(15);

root.right.right = new Node(20);

if (isBalanced(root) > 0)

System.out.print("Balanced");

else

System.out.print("Not Balanced");

}

// Function to check if tree is height balanced

public static int isBalanced(Node root)

{

if (root == null)

return 0;

int lh = isBalanced(root.left);

if (lh == -1)

return -1;

int rh = isBalanced(root.right);

if (rh == -1)

return -1;

if (Math.abs(lh - rh) > 1)

return -1;

else

return Math.max(lh, rh) + 1;

}

}

**Output**

Balanced

**Time Complexity:** O(n)

* Because we are only one dfs call and utilizing the height returned from that to determine the height balance, it is performing the task in linear time.

**Auxiliary Space:**O(n)