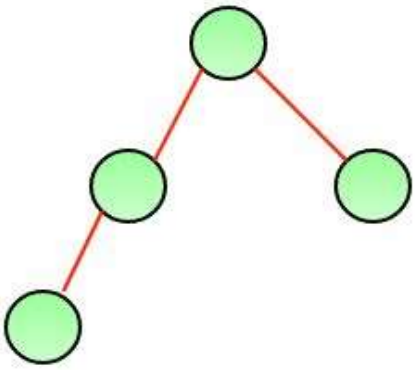
How to determine if a binary tree is height-balanced?

A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

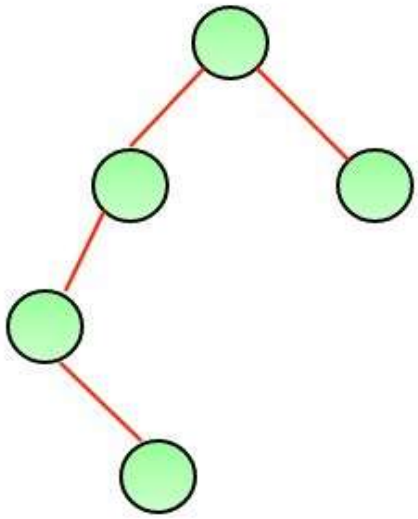
Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

An empty tree is height-balanced. A non-empty binary tree T is balanced if:
1) Left subtree of T is balanced
2) Right subtree of T is balanced
3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.



A height balanced tree



Not a height balanced tree

To check if a tree is height-balanced, get the height of left and right subtrees.

Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

```
/* CPP program to check if
a tree is height-balanced or not */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data,
pointer to left child and
a pointer to right child */
class node {
public:
    int data;
    node* left;
    node* right;
};

/* Returns the height of a binary tree */
int height(node* node);

/* Returns true if binary tree
with root as root is height-balanced */
bool isBalanced(node* root)
{
    int lh; /* for height of left subtree */
    int rh; /* for height of right subtree */

    /* If tree is empty then return true */
    if (root == NULL)
        return 1;

    /* Get the height of left and right sub trees */
    lh = height(root->left);
    rh = height(root->right);

    if (abs(lh - rh) <= 1 && isBalanced(root->left) && isBalanced(root->right))
        return 1;

    /* If we reach here then
```

```
    tree is not height-balanced */
    return 0;
}

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

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b) ? a : b;
}

/* 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 + max(height(node->left),
                   height(node->right));
}

/* Helper function that allocates
a new node with the given data
and NULL left and right pointers. */
node* newNode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}

// Driver code
int main()
{
    node* root = newNode(1);
```

```
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(8);

    if (isBalanced(root))
        cout << "Tree is balanced";
    else
        cout << "Tree is not balanced";
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:
Tree is not balanced

Time Complexity: O(n^2) Worst case occurs in case of skewed tree.

Optimized implementation: Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to O(n).

```
/* C++ program to check if a tree
is height-balanced or not */
#include <bits/stdc++.h>
using namespace std;
#define bool int

/* A binary tree node has data,
pointer to left child and
a pointer to right child */
class node {
public:
    int data;
    node* left;
    node* right;
};

/* The function returns true if root is
balanced else false The second parameter
```

```
is to store the height of tree. Initially,
we need to pass a pointer to a location with
value as 0. We can also write a wrapper
over this function */
bool isBalanced(node* root, int* height)
{

    /* lh --> Height of left subtree
    rh --> Height of right subtree */
    int lh = 0, rh = 0;

    /* l will be true if left subtree is balanced
    and r will be true if right subtree is balanced */
    int l = 0, r = 0;

    if (root == NULL) {
        *height = 0;
        return 1;
    }

    /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in l and r */
    l = isBalanced(root->left, &lh);
    r = isBalanced(root->right, &rh);

    /* Height of current node is max of heights of left and
    right subtrees plus 1*/
    *height = (lh > rh ? lh : rh) + 1;

    /* If difference between heights of left and right
    subtrees is more than 2 then this node is not balanced
    so return 0 */
    if (abs(lh - rh) >= 2)
        return 0;

    /* If this node is balanced and left and right subtrees
    are balanced then return true */
    else
        return l && r;
}

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

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
node* newNode(int data)
```

```
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}

// Driver code
int main()
{
    int height = 0;

    /* Constructed binary tree is
            1
            / \
            2 3
            / \ /
            4 5 6
            /
            7
    */
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->left->left->left = newNode(7);

    if (isBalanced(root, &height))
        cout << "Tree is balanced";
    else
        cout << "Tree is not balanced";

    return 0;
}

// This is code is contributed by rathbhupendra
```

Output
Tree is balanced

Time Complexity: O(n)