Session 8

Tree Data Structure Part 2

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

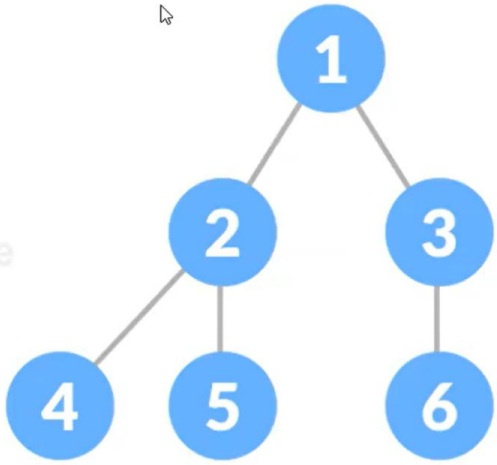
Tree Data Structure

Complete Binary Tree

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

A complete binary tree is just like a full binary tree, but with two major differences

All the leaf elements must lean towards the left.
The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



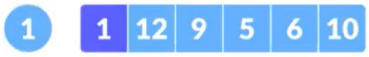
Slide 137 of 166

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

How a Complete Binary Tree is Created?

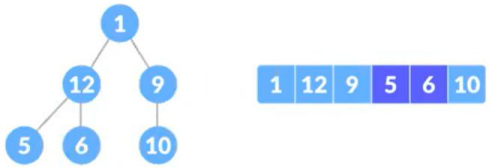
1.Select the first element of the list to be the root node. (no. of elements on level-I: 1)



2. Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)

3. Put the next two elements as children of the left node of the second level. Again, put the next two elements as children of the right node of the second level (no. of elements on level-III: 4) elements).

4. Keep repeating until you reach the last element



Slide 138 of 166

Tree Data Structure

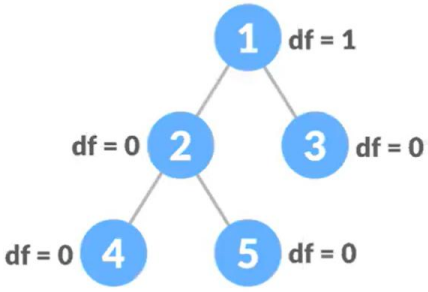
Balanced Binary Tree

A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

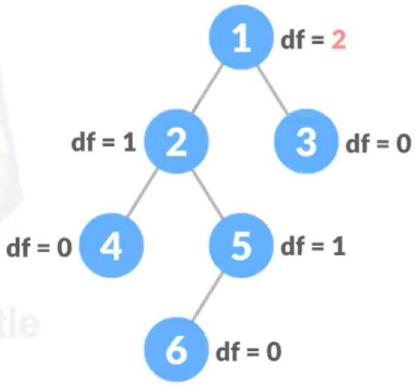
Following are the conditions for a height-balanced binary tree:

1.difference between the left and the right subtree for any node is not more than one
2.the left subtree is balanced
3.the right subtree is balanced

Tree Data Structure



Balanced Binary Tree with depth at each level



df = |height of left child - height of right child|

Unbalanced Binary Tree with depth at each level

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

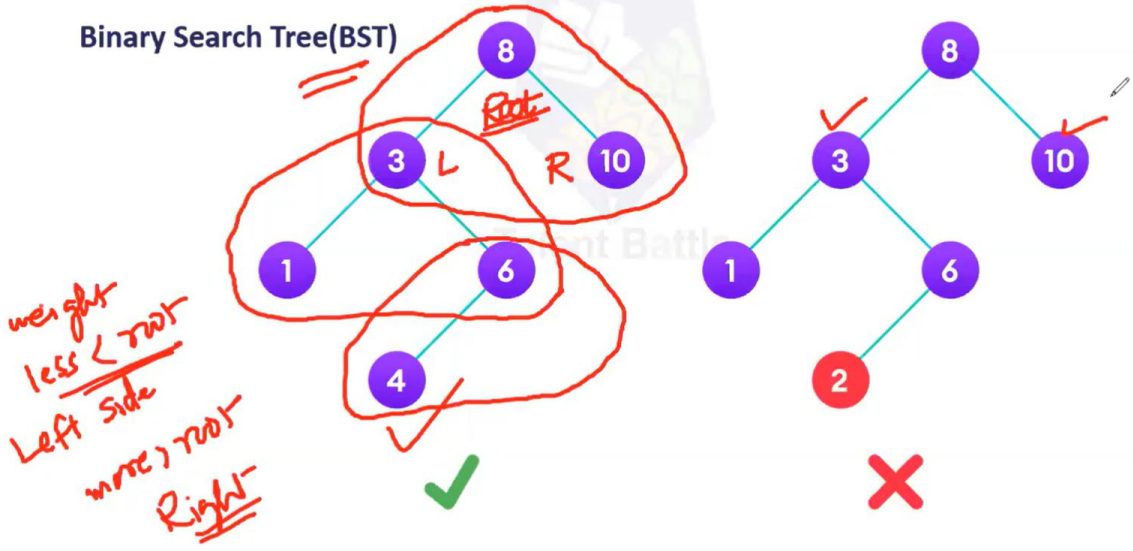
It is called a binary tree because each tree node has a maximum of two children.
It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.
The properties that separate a binary search tree from a regular binary tree is:

All nodes of left subtree are less than the root node
All nodes of right subtree are more than the root node
Both subtrees of each node are also BSTs i.e. they have the above two properties

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)



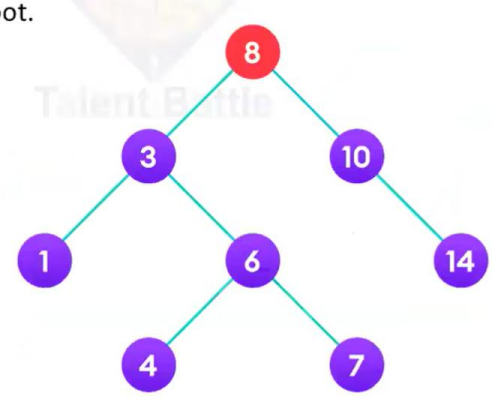
Tree Data Structure — Binary Search Tree(BST)

There are two basic operations that you can perform on a binary search tree:

Search Operation
The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

Algorithm:

```
If root == NULL
   return NULL;
If number == root->data
   return root->data;
If number < root->data
   return search(root->left)
If number > root->data
   return search(root->right)
```



Tree Data Structure — Binary Search Tree(BST)

Insert Operation
Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.
We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

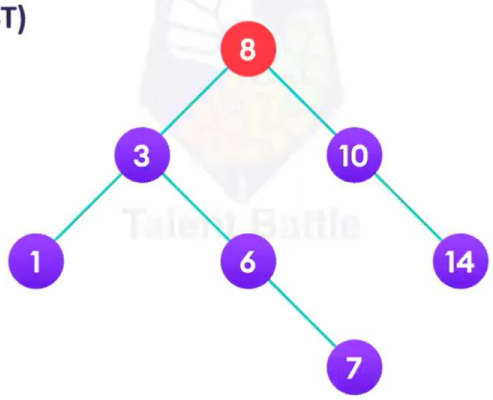
Algorithm:

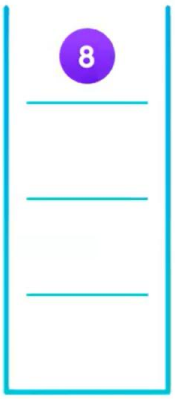
```
If node == NULL
   return createNode(data)
if (data < node->data)
   node->left  = insert(node->left, data);
else if (data > node->data)
   node->right = insert(node->right, data);
return node;
```

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)



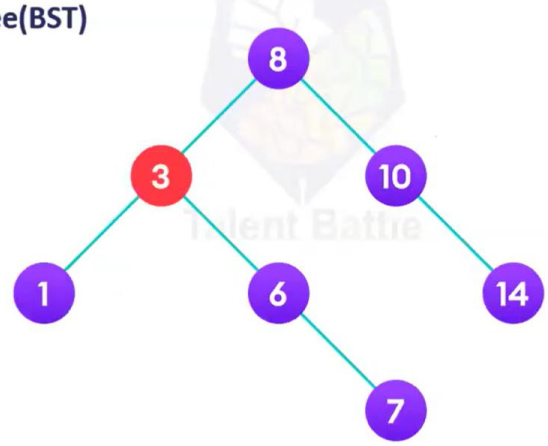


4<8 so, transverse through the left child of 8

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)

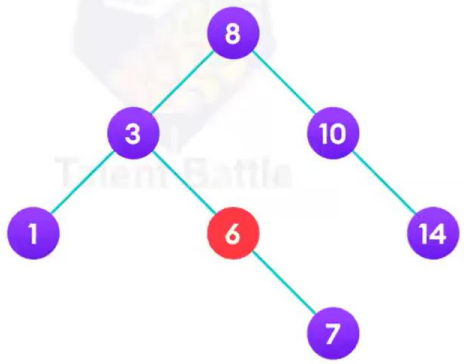


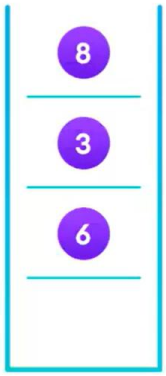
4>3 so, transverse through the right child of 8

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)



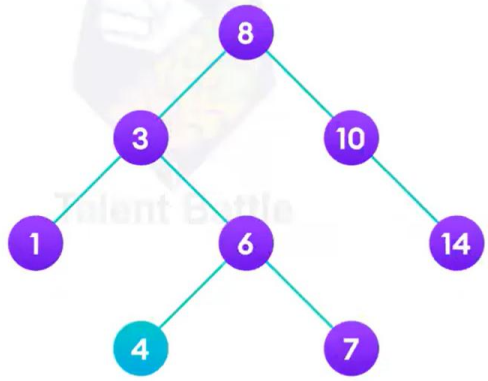


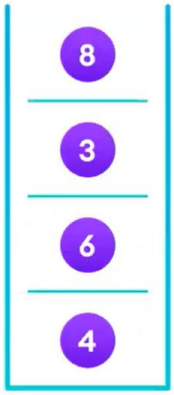
4<6 so, transverse through the left child of 6

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)





Insert 4 as a left child of 6

Tree Data Structure

Binary Search Tree(BST)

Deletion Operation

There are three cases for deleting a node from a binary search tree.

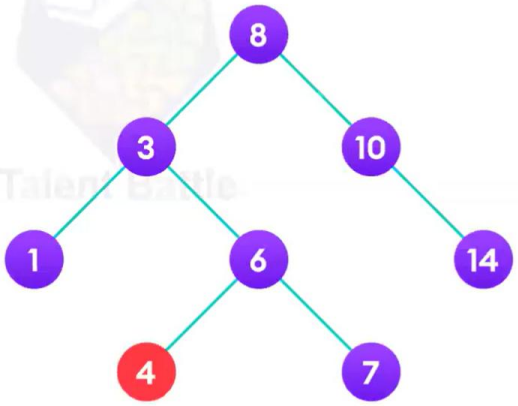
Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

Tree Data Structure

Binary Search Tree(BST)

4 is to be deleted

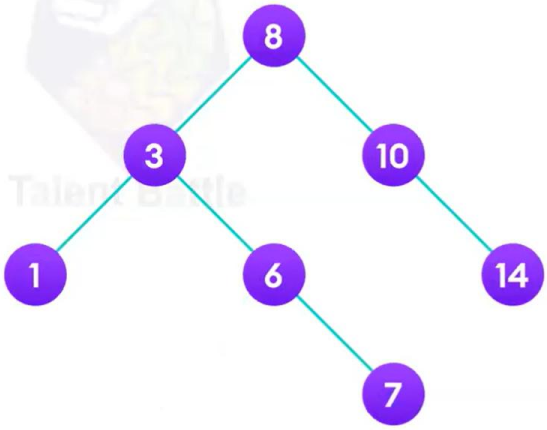


PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)

Delete the node



PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

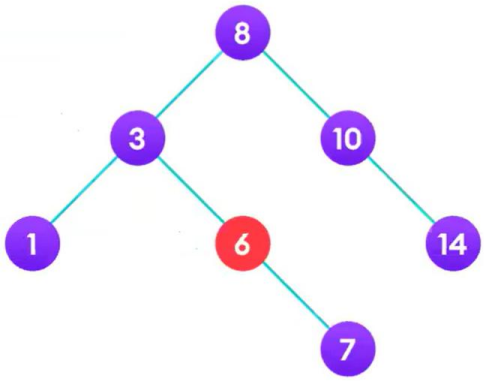
Binary Search Tree(BST)

Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1.Replace that node with its child node.

2.Remove the child node from its original position.

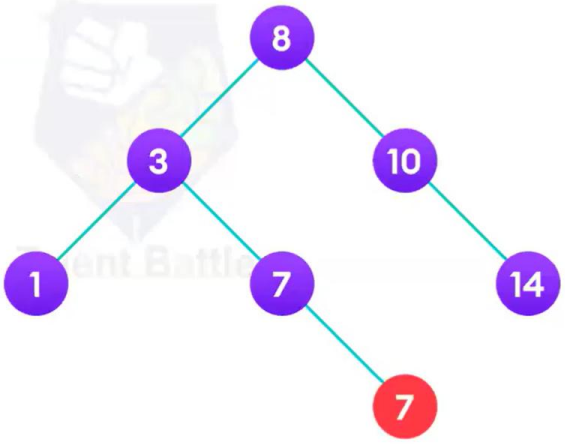


6 is to be deleted

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)



copy the value of its child to the node and delete the child

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

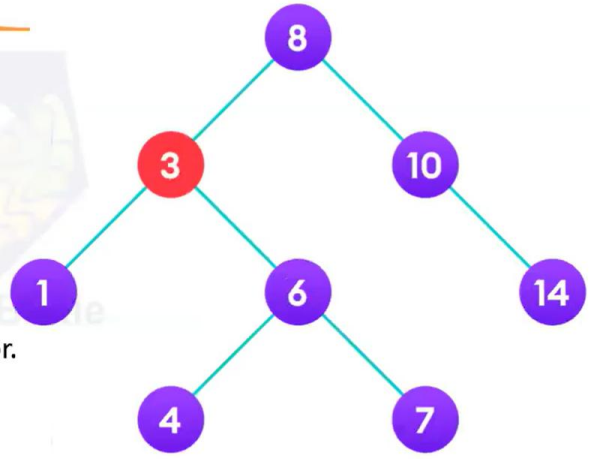
Tree Data Structure

Binary Search Tree(BST)

Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

1.Get the inorder successor of that node.
2.Replace the node with the inorder successor.
3.Remove the inorder successor from its original position.

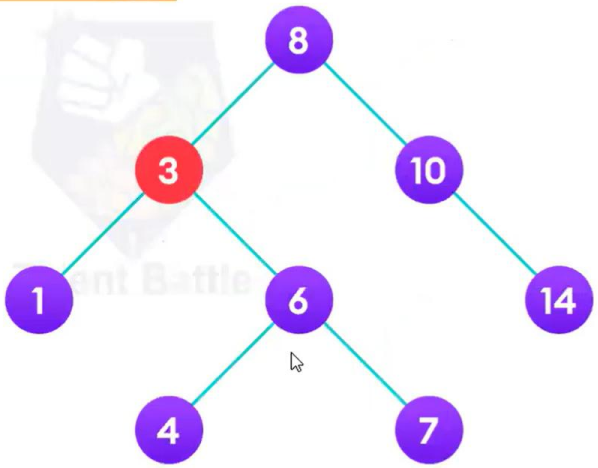


3 is to be deleted

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)

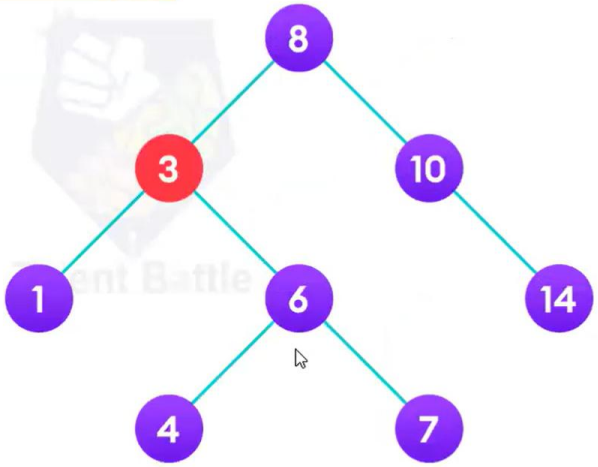


Copy the value of the inorder successor (4) to the node

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)



Copy the value of the inorder successor (4) to the node

This video is sole property of Talent Battle Pvt. Ltd. Strict penal action will be taken against unauthorized piracy of this video

Slide 156 of 166

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree Complexities
Time Complexity

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Space Complexity
The space complexity for all the operations is O(n).

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

Tree Data Structure

Binary Search Tree(BST)

Binary Search Tree Applications

1.In multilevel indexing in the database

2.For dynamic sorting

3.For managing virtual memory areas in Unix kernel

Tree Data Structure — AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor **Georgy Adelson-Velsky and Landis**.

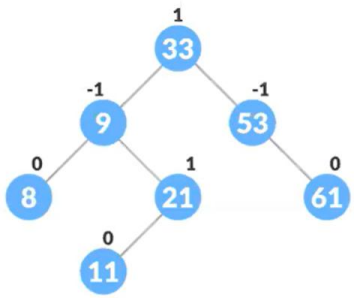
Tree Data Structure — AVL Tree

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be **-1, 0 or +1**.



Tree Data Structure | AVL Tree

Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

Rotating the subtrees in an AVL Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

Left Rotate
Right Rotate
Left-Right & Right-Left Rotate

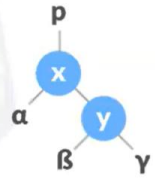
Tree Data Structure | AVL Tree

Left Rotate

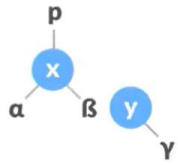
In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

Algorithm

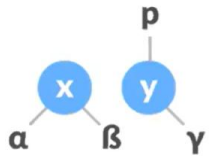
1. Let the initial tree be:
2. If y has a left subtree, assign x as the parent of the left subtree of y.
3. If the parent of x is NULL, make y as the root of the tree.
4. Else if x is the left child of p, make y as the left child of p.
5. Else assign y as the right child of p.
6. Make y as the parent of x.



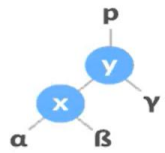
Left rotate



Assign x as the parent of the left subtree of y



Change the parent of x to that of y



Assign y as the parent of x.

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

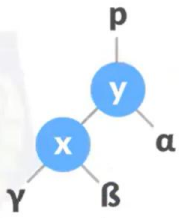
Tree Data Structure — AVL Tree

Right Rotate

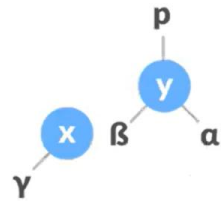
In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

Algorithm

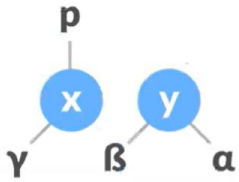
1. Let the initial tree be:
2. If x has a left subtree, assign y as the parent of the right subtree of x.
3. If the parent of y is NULL, make x as the root of the tree.
4. Else if y is the right child of p, make x as the right child of p.
5. Else assign x as the left child of p.
6. Make x as the parent of y.



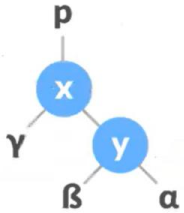
Initial Tree



Assign y as the parent of the right subtree of x



Assign parent of y as parent of x



Assign x as the parent of y.

PowerPoint Slide Show - [DSA Technical Training] - PowerPoint

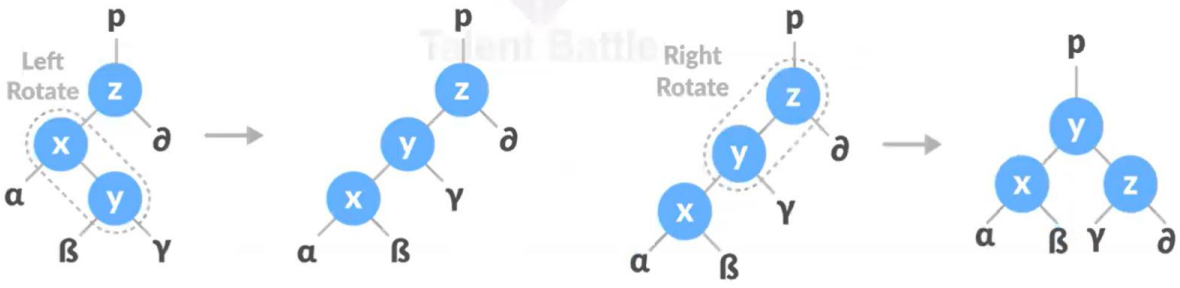
Tree Data Structure — AVL Tree

Left-Right Rotate

In left-right rotation, the arrangements are first shifted to the left and then to the right.

1.Do left rotation on x-y

2.Do right rotation on y-z



Left rotate x-y

Right rotate z-y

```
// perfect binary tree

/*
#include<iostream>
using namespace std;

struct Node{
    int key;
    struct Node *left, *right;
};

int depth(Node *node){
    int d=0;
    while(node != NULL){
        d++;
        node = node->left;
    }
    return d;
}

bool isPerfectR(struct Node *root, int d, int level=0){
    if(root == NULL){
        return true;
    }

    if (root -> left == NULL && root->right == NULL){
        return (d == level+1);
    }

    if(root->left == NULL || root->right == NULL){
        return false;
    }

    return isPerfectR(root->left, d, level+1 &&  isPerfectR(root->right,
d, level+1));
}

bool isPerfect(Node *root){
    int d = depth(root);
    return isPerfectR(root, d);
```

```
}

struct Node *newNode(int k){
    struct Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

int main(){
    struct Node *root = NULL;
    root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);

    root->left->left = newNode(40);
    root->left->right = newNode(60);

    if(isPerfect(root)){
        cout << "\ntree is perfect binary tree";
    }
    else {
        cout << "\ntree is not perfect binary tree";
    }

    return 0;
}

*/


// ------------ Output ----------------------

// PS C:\Users\hp\Desktop\TCS IT\DSA-WORDSPACE\c++\session8> g++
main.cpp -o main
// PS C:\Users\hp\Desktop\TCS IT\DSA-WORDSPACE\c++\session8>
./main

// tree is not perfect binary tree
```

```cpp
//++++++++++++++++++++++ other method ++++++++++

// full binary tree

#include<iostream>
using namespace std;

struct Node{
    int key;
    struct Node *left, *right;
};

// Function to calculate the depth of the leftmost node
int depth(Node *node){
    int d = 0;
    while(node != NULL){
        d++;
        node = node->left;
    }
    return d;
}

// Function to check if the tree is perfect
bool isPerfectR(struct Node *root, int d, int level = 0){
    if(root == NULL){
        return true;
    }

    if (root->left == NULL && root->right == NULL){
        return (d == level + 1);
    }

    if(root->left == NULL || root->right == NULL){
        return false;
    }
```

```
    return isPerfectR(root->left, d, level + 1) && isPerfectR(root-
>right, d, level + 1);
}

// Wrapper function to check if the tree is perfect
bool isPerfect(Node *root){
    int d = depth(root);
    return isPerfectR(root, d);
}

// Helper function to create a new node
Node* newNode(int key){
    Node* node = new Node();
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

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->right->left = newNode(6);
    root->right->right = newNode(7);

    if(isPerfect(root)){
        cout << "The tree is a perfect binary tree" << endl;
    } else {
        cout << "The tree is not a perfect binary tree" << endl;
    }

    return 0;
}

// ---------- output ----------------

// PS C:\Users\hp\Desktop\TCS IT\DSA-WORDSPACE\c++\session8> g++
main.cpp -o main
```

```
// PS C:\Users\hp\Desktop\TCS IT\DSA-WORDSPACE\c++\session8> ./main
// The tree is a perfect binary tree
```