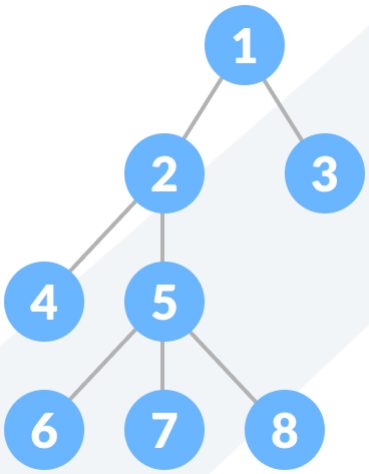
Tree Data Structure

In this tutorial, you will learn about tree data structure. Also, you will learn about different types of trees and the terminologies used in tree.

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



A Tree

Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree Terminologies

Node

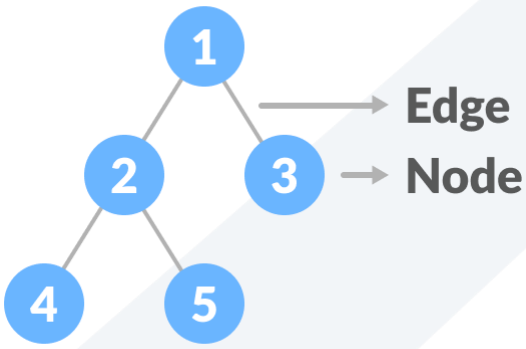
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.

The node having at least a child node is called an **internal node**.

Edge

It is the link between any two nodes.



Nodes and edges of a tree

Root

It is the topmost node of a tree.

Height of a Node

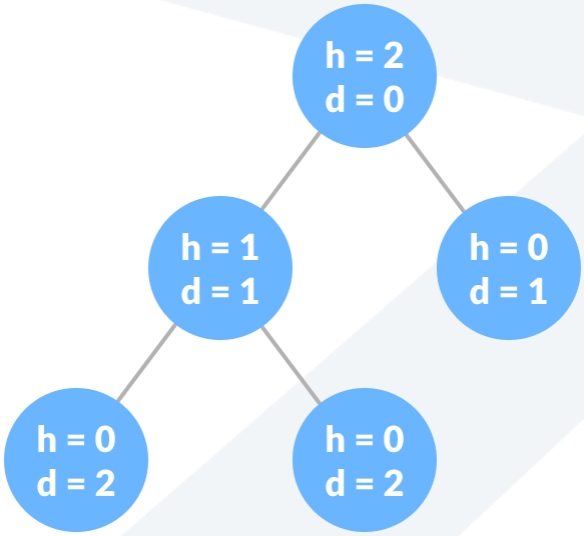
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.



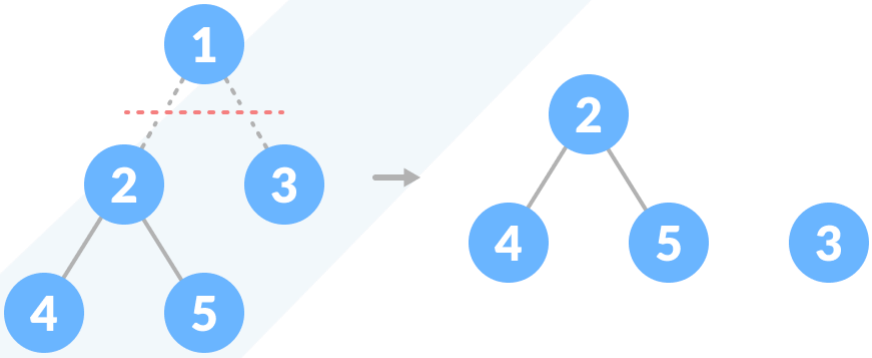
Height and depth of each node in a tree

Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.



Creating forest from a tree

You can create a forest by cutting the root of a tree.

Types of Tree

1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. B-Tree

Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

To learn more, please visit tree traversal.

Tree Applications

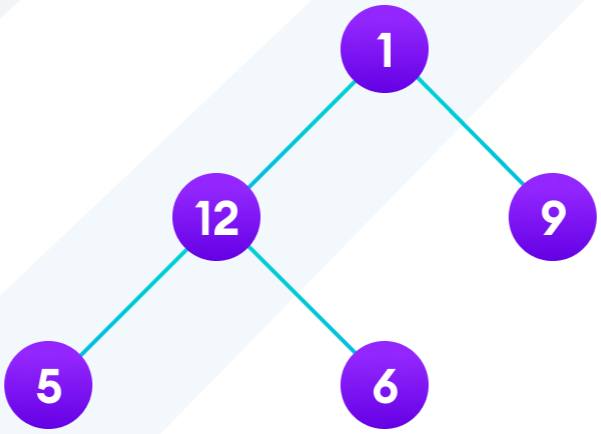
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Tree Traversal - inorder, preorder and postorder

In this tutorial, you will learn about different tree traversal techniques. Also, you will find working examples of different tree traversal methods in C, C++, Java and Python.

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.



Tree traversal

Let's think about how we can read the elements of the tree in the image shown above.

Starting from top, Left to right

```
1 -> 12 -> 5 -> 6 -> 9
```

Starting from bottom, Left to right

```
5 -> 6 -> 12 -> 9 -> 1
```

Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes.

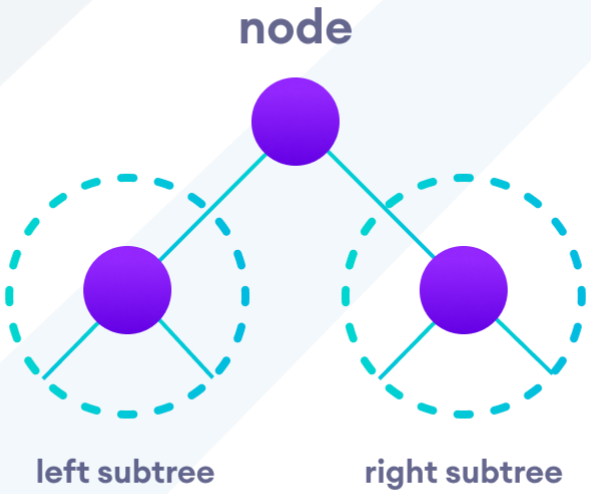
Instead, we use traversal methods that take into account the basic structure of a tree i.e.

```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```

The struct node pointed to by *left* and *right* might have other left and right children so we should think of them as sub-trees instead of sub-nodes.

According to this structure, every tree is a combination of

- A node carrying data
- Two subtrees



Left and Right Subtree

Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.

Depending on the order in which we do this, there can be three types of traversal.

Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

```
inorder(root->left)
```

```
display(root->data)
inorder(root->right)
```

Preorder traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

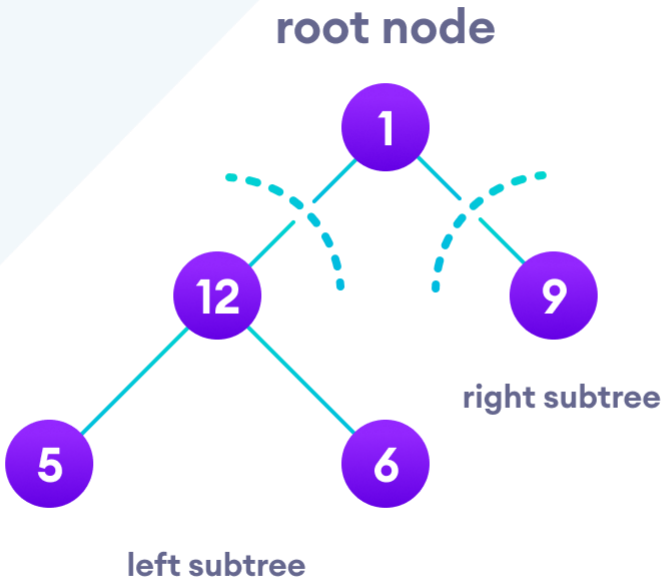
```
display(root->data)
preorder(root->left)
preorder(root->right)
```

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

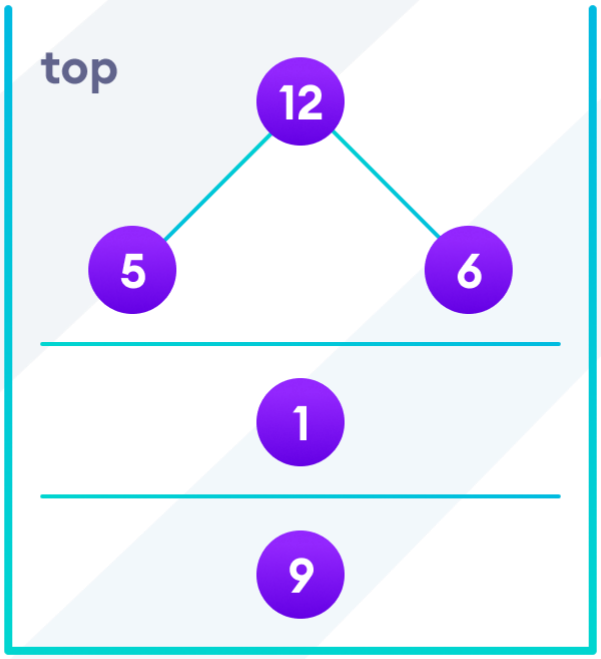
Let's visualize in-order traversal. We start from the root node.



Left and Right Subtree

We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.

Let's put all this in a stack so that we remember.



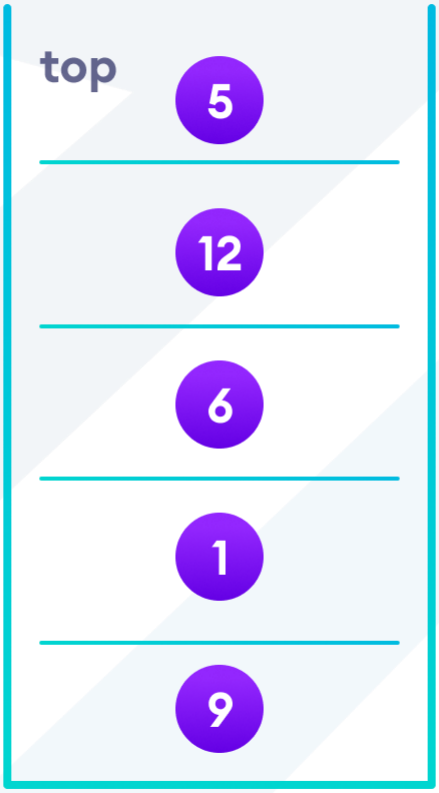
Stack

Now we traverse to the subtree pointed on the TOP of the stack.

Again, we follow the same rule of inorder

```
Left subtree -> root -> right subtree
```

After traversing the left subtree, we are left with



Final Stack

Since the node "5" doesn't have any subtrees, we print it directly. After that we print its parent "12" and then the right child "6".

Putting everything on a stack was helpful because now that the left-subtree of the root node has been traversed, we can print it and go to the right subtree.

After going through all the elements, we get the inorder traversal as

```
5 -> 12 -> 6 -> 1 -> 9
```

We don't have to create the stack ourselves because recursion maintains the correct order for us.

```
// Tree traversal in C++

#include <iostream>
using namespace std;

struct Node {
  int data;
  struct Node *left, *right;
  Node(int data) {
```

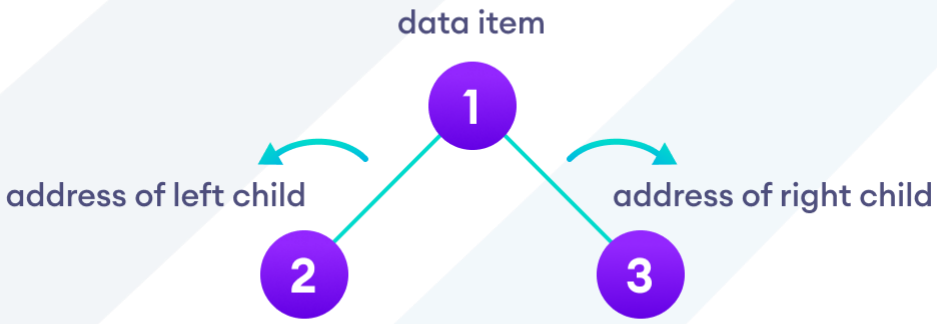
```
    this->data = data;
    left = right = NULL;
  }
};

// Preorder traversal
void preorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  cout << node->data << "->";
  preorderTraversal(node->left);
  preorderTraversal(node->right);
}

// Postorder traversal
void postorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  postorderTraversal(node->left);
  postorderTraversal(node->right);
  cout << node->data << "->";
}

// Inorder traversal
void inorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  inorderTraversal(node->left);
  cout << node->data << "->";
  inorderTraversal(node->right);
}

int main() {
  struct Node* root = new Node(1);
  root->left = new Node(12);
  root->right = new Node(9);
  root->left->left = new Node(5);
  root->left->right = new Node(6);

  cout << "Inorder traversal ";
  inorderTraversal(root);

  cout << "\nPreorder traversal ";
  preorderTraversal(root);

  cout << "\nPostorder traversal ";
  postorderTraversal(root);
```

Binary Tree

In this tutorial, you will learn about binary tree and its different types. Also, you will find working examples of binary tree in C, C++, Java and Python.

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

- data item
- address of left child
- address of right child

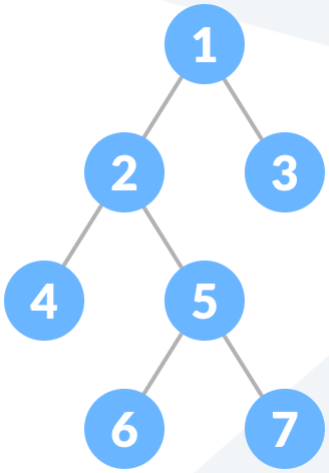


Binary Tree

Types of Binary Tree

1. Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

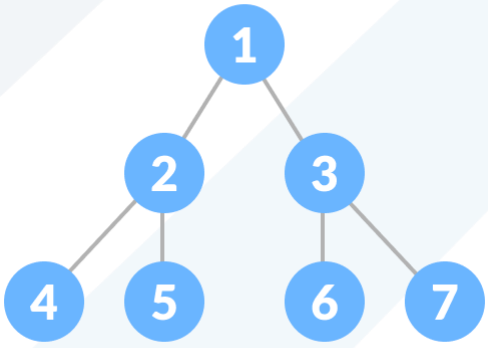


Full Binary Tree

To learn more, please visit full binary tree.

2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



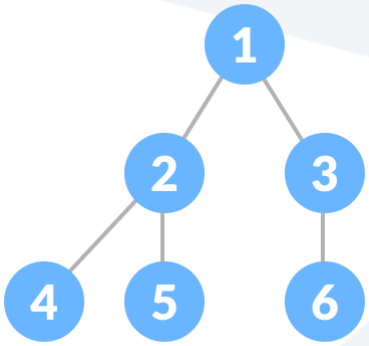
Perfect Binary Tree

To learn more, please visit perfect binary tree.

3. Complete Binary Tree

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

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. 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.

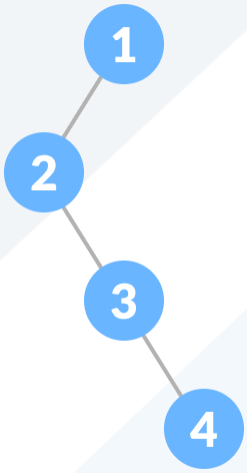


Complete Binary Tree

To learn more, please visit complete binary tree.

4. Degenerate or Pathological Tree

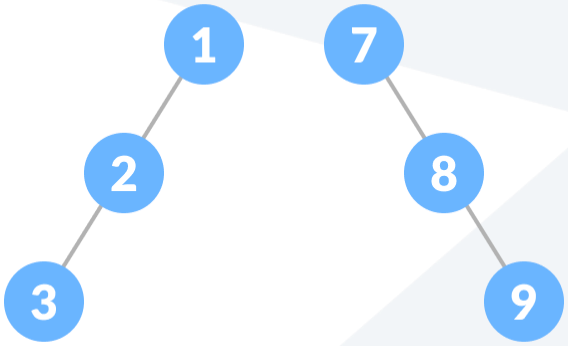
A degenerate or pathological tree is the tree having a single child either left or right.



Degenerate Binary Tree

5. Skewed Binary Tree

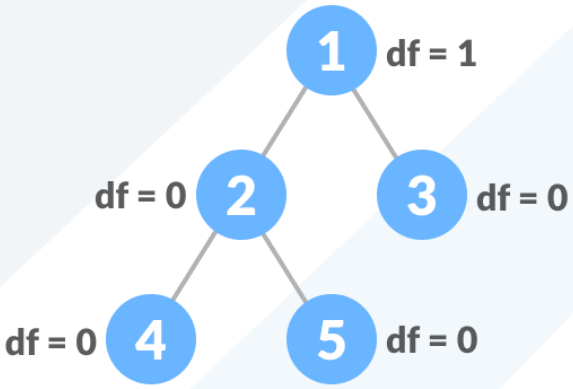
A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.



Skewed Binary Tree

6. Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



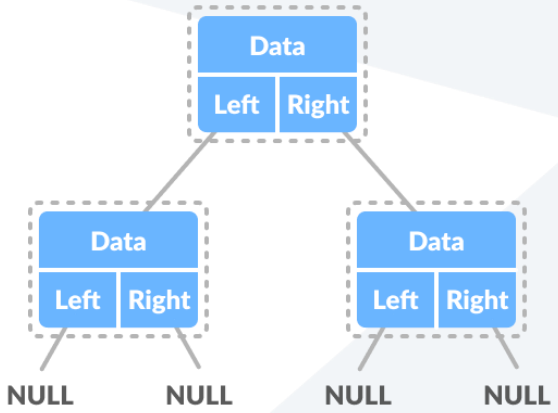
Balanced Binary Tree

To learn more, please visit balanced binary tree.

Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

```
struct node
{
 int data;
 struct node *left;
 struct node *right;
};
```



Binary Tree Representation

```
// Binary Tree in C++

#include <stdlib.h>

#include <iostream>

using namespace std;

struct node {
  int data;
  struct node *left;
  struct node *right;
};

// New node creation
struct node *newNode(int data) {
  struct node *node = (struct node *)malloc(sizeof(struct node));

  node->data = data;

  node->left = NULL;
  node->right = NULL;
  return (node);
}

// Traverse Preorder
void traversePreOrder(struct node *temp) {
  if (temp != NULL) {
    cout << " " << temp->data;
    traversePreOrder(temp->left);
    traversePreOrder(temp->right);
  }
}

// Traverse Inorder
void traverseInOrder(struct node *temp) {
  if (temp != NULL) {
    traverseInOrder(temp->left);
```

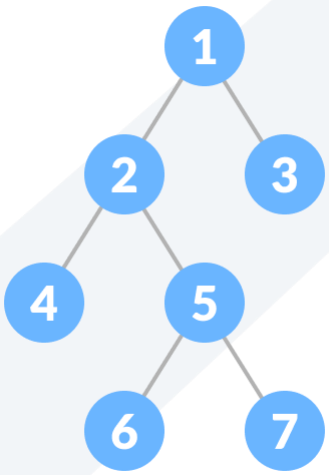
```
    cout << " " << temp->data;
    traverseInOrder(temp->right);
  }
}

// Traverse Postorder
void traversePostOrder(struct node *temp) {
  if (temp != NULL) {
    traversePostOrder(temp->left);
    traversePostOrder(temp->right);
    cout << " " << temp->data;
  }
}

int main() {
  struct node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);

  cout << "preorder traversal: ";
  traversePreOrder(root);
  cout << "\nInorder traversal: ";
  traverseInOrder(root);
  cout << "\nPostorder traversal: ";
  traversePostOrder(root);
}
```

Full Binary Tree

In this tutorial, you will learn about full binary tree and its different theorems. Also, you will find working examples to check full binary tree in C, C++, Java and Python.

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

It is also known as **a proper binary tree**.



Full Binary Tree

Full Binary Tree Theorems

```
Let, i = the number of internal nodes
       n = be the total number of nodes
       l = number of leaves
      λ = number of levels
```

1. The number of leaves is `i + 1`.
2. The total number of nodes is `2i + 1`.
3. The number of internal nodes is *(n – 1) / 2*.
4. The number of leaves is `(n + 1) / 2`.
5. The total number of nodes is `2l - 1`.
6. The number of internal nodes is `l - 1`.
7. The number of leaves is at most $2^{\lambda - 1}$.

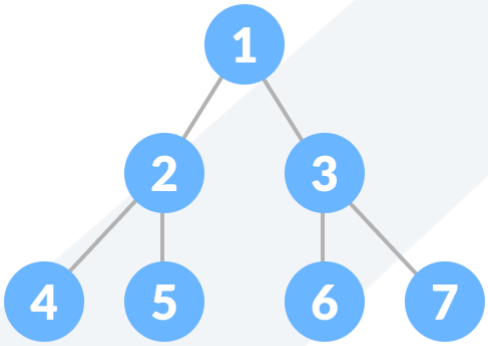
```
// Checking if a binary tree is a full binary tree in C++
```

```
#include <iostream>
using namespace std;

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

// New node creation
struct Node *newNode(char k) {
  struct Node *node = (struct Node *)malloc(sizeof(struct Node));
  node->key = k;
  node->right = node->left = NULL;
  return node;
}

bool isFullBinaryTree(struct Node *root) {

  // Checking for emptiness
  if (root == NULL)
    return true;

  // Checking for the presence of children
  if (root->left == NULL && root->right == NULL)
    return true;

  if ((root->left) && (root->right))
    return (isFullBinaryTree(root->left) && isFullBinaryTree(root->right));

  return false;
}

int main() {
  struct Node *root = NULL;
  root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->left->right->left = newNode(6);
  root->left->right->right = newNode(7);

  if (isFullBinaryTree(root))
    cout << "The tree is a full binary tree\n";
  else
    cout << "The tree is not a full binary tree\n";
}
```

Perfect Binary Tree

In this tutorial, you will learn about the perfect binary tree. Also, you will find working examples for checking a perfect binary tree in C, C++, Java and Python.

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

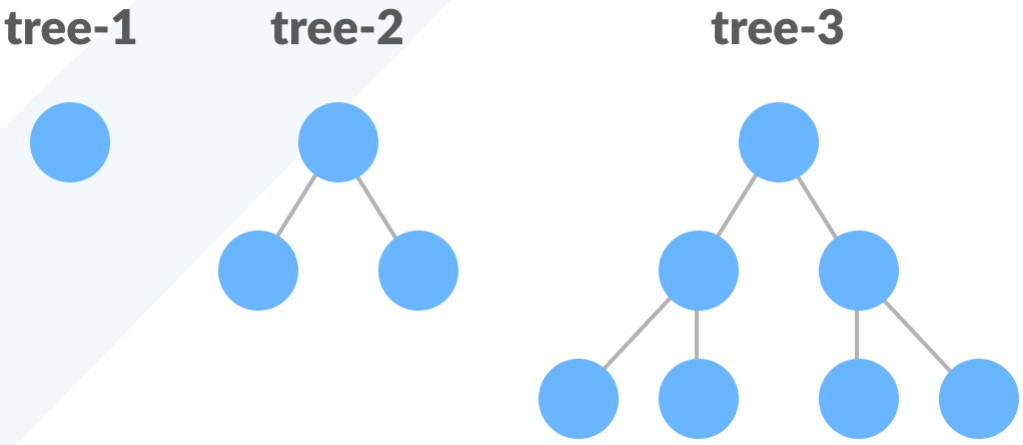


Perfect Binary Tree

All the internal nodes have a degree of 2.

Recursively, a perfect binary tree can be defined as:

1. If a single node has no children, it is a perfect binary tree of height `h = 0`,
2. If a node has `h > 0`, it is a perfect binary tree if both of its subtrees are of height `h - 1` and are non-overlapping.



Perfect Binary Tree (Recursive Representation)

```
// Checking if a binary tree is a perfect binary tree in C++

#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(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->right->left = newNode(6);

  if (isPerfect(root))
    cout << "The tree is a perfect binary tree\n";
```

```
  else
    cout << "The tree is not a perfect binary tree\n";
}
```

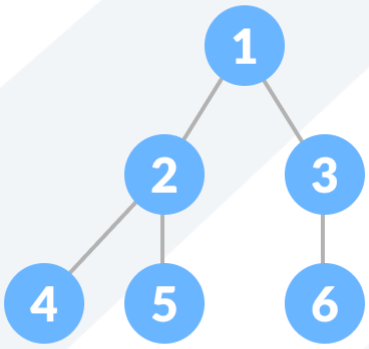
Complete Binary Tree

In this tutorial, you will learn about a complete binary tree and its different types. Also, you will find working examples of a complete binary tree in C, C++, Java and Python.

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.

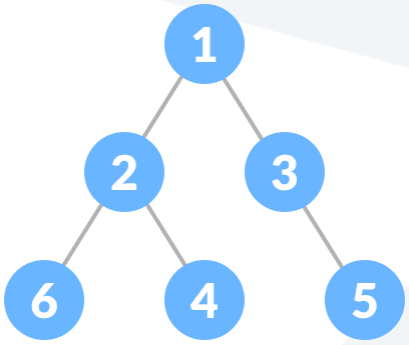
A complete binary tree is just like a full binary tree, but with two major differences

1. All the leaf elements must lean towards the left.
2. 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.



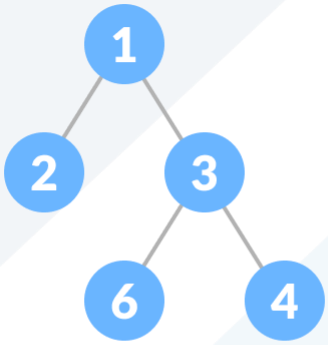
Complete Binary Tree

Full Binary Tree vs Complete Binary Tree



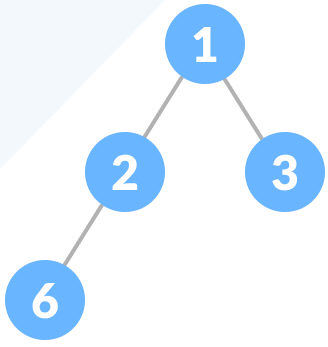
✖ Full Binary Tree
✖ Complete Binary Tree

Comparison between full binary tree and complete binary tree



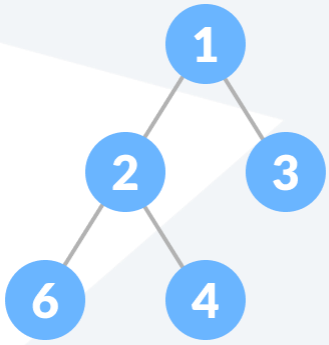
✔ Full Binary Tree
✖ Complete Binary Tree

Comparison between full binary tree and complete binary tree



✖ Full Binary Tree
✔ Complete Binary Tree

Comparison between full binary tree and complete binary tree



✔ Full Binary Tree
✔ Complete Binary Tree

and complete binary tree Comparison between full binary tree and complete binary tree

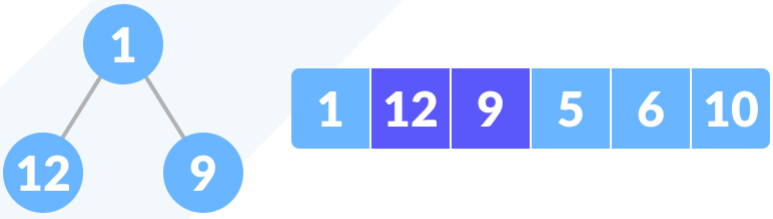
---

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)

Select the first element as root

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)



12 as a left child and 9 as a right child

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.

5 as a left child and 6 as a right child

```
// Checking if a binary tree is a complete binary tree in C++

#include <iostream>

using namespace std;

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

// Create node
struct Node *newNode(char k) {
  struct Node *node = (struct Node *)malloc(sizeof(struct Node));
  node->key = k;
  node->right = node->left = NULL;
  return node;
}

// Count the number of nodes
int countNumNodes(struct Node *root) {
  if (root == NULL)
    return (0);
  return (1 + countNumNodes(root->left) + countNumNodes(root->right));
}

// Check if the tree is a complete binary tree
bool checkComplete(struct Node *root, int index, int numberNodes) {

  // Check if the tree is empty
  if (root == NULL)
    return true;

  if (index >= numberNodes)
    return false;
```

```
  return (checkComplete(root->left, 2 * index + 1, numberNodes) &&
checkComplete(root->right, 2 * index + 2, numberNodes));
}

int main() {
  struct Node *root = NULL;
  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);

  int node_count = countNumNodes(root);
  int index = 0;

  if (checkComplete(root, index, node_count))
    cout << "The tree is a complete binary tree\n";
  else
    cout << "The tree is not a complete binary tree\n";
}
```

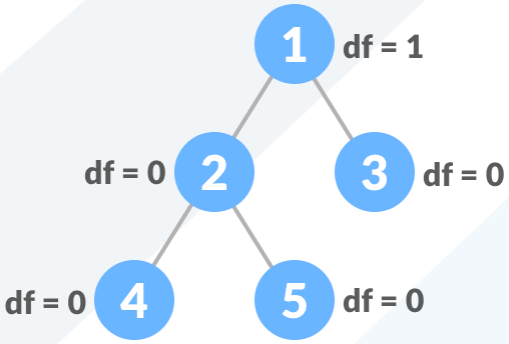
Balanced Binary Tree

In this tutorial, you will learn about a balanced binary tree and its different types. Also, you will find working examples of a balanced binary tree in C, C++, Java and Python.

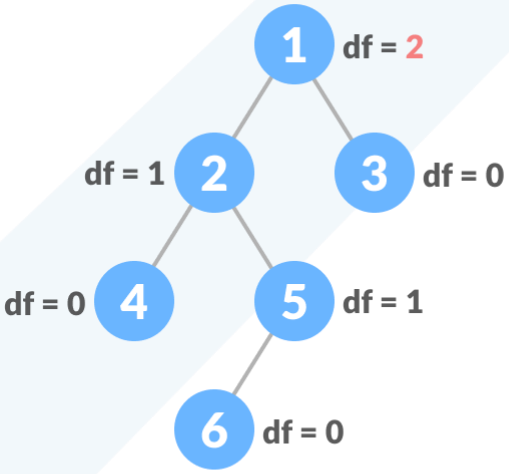
A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

To learn more about the height of a tree/node, visit Tree Data Structure.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



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

```
// Checking if a binary tree is height balanced in C++

#include
using namespace std;

#define bool int

class node {
   public:
  int item;
  node *left;
  node *right;
};

// Create anew node
node *newNode(int item) {
  node *Node = new node();
  Node->item = item;
  Node->left = NULL;
  Node->right = NULL;

  return (Node);
}

// Check height balance
bool checkHeightBalance(node *root, int *height) {
  // Check for emptiness
  int leftHeight = 0, rightHeight = 0;

  int l = 0, r = 0;

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

  l = checkHeightBalance(root->left, &leftHeight);
  r = checkHeightBalance(root->right, &rightHeight);

  *height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

  if (std::abs(leftHeight - rightHeight >= 2))
    return 0;

  else
    return l && r;
}

int main() {
  int height = 0;

  node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
```

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

  if (checkHeightBalance(root, &height))
    cout << "The tree is balanced";
  else
    cout << "The tree is not balanced";
}
```