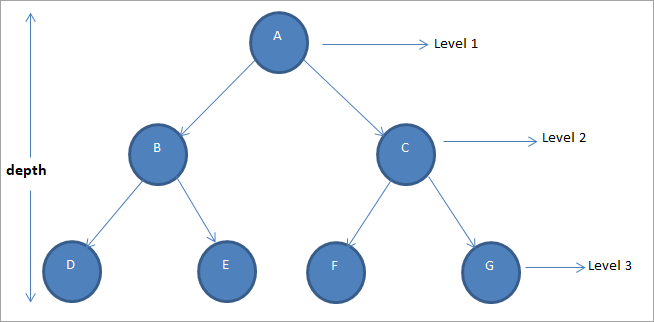
Binary Tree In C++

**A pictorial representation of a binary tree is shown below:**

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/pictorial-representation-of-a-binary-tree.png)

In a given binary tree, the maximum number of nodes at any level is 2l-1where ‘l’ is the level number.

Thus in case of the root node at level 1, the max number of nodes = 2 1-1 = 20 = 1

As every node in a binary tree has at most two nodes, the maximum nodes at the next level will be, 2\*2l-1.

Given a binary tree of depth or height of h, the maximum number of nodes in a binary tree of height h = 2h – 1.

Hence in a binary tree of height 3 (shown above), the maximum number of nodes = 23-1 = 7.

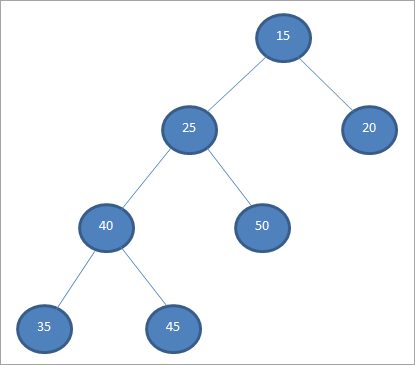
**Now let us discuss the various types of binary trees.**

Types Of Binary Tree

Following are the most common types of binary trees.

#1) Full Binary Tree

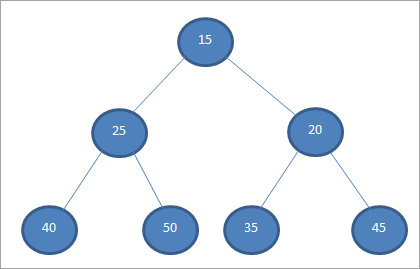
A binary tree in which every node has 0 or 2 children is termed as a full binary tree.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/a-full-binary-tree.png)

Above shown is a full binary tree in which we can see that all its nodes except the leaf nodes have two children. If L is the number of leaf nodes and ‘l’ is the number of internal or non-leaf nodes, then for a full binary tree, L = l + 1.

#2) Complete Binary Tree

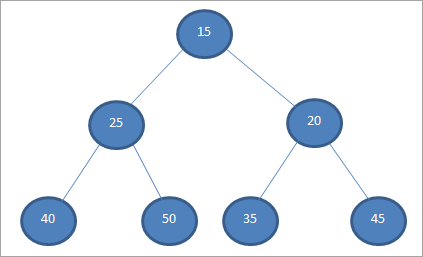
A complete binary tree has all the levels filled except for the last level and the last level has all its nodes as much as to the left.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Complete-binary-tree.png)

The tree shown above is a complete binary tree. A typical example of a complete binary tree is a binary heap which we will discuss in the later tutorials.

#3) Perfect Binary Tree

A binary tree is termed perfect when all its internal nodes have two children and all the leaf nodes are at the same level.

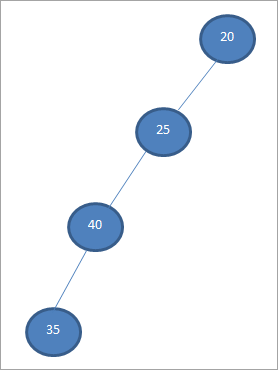
[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Perfect-binary-tree.png)

A binary tree example shown above is a perfect binary tree as each of its nodes has two children and all the leaf nodes are at the same level.

A perfect binary tree of height h has 2h – 1 number of nodes.

#4) A Degenerate Tree

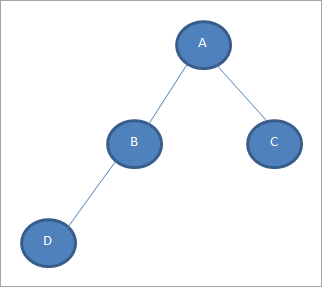
A binary tree where each internal node has only one child is called a degenerate tree.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/A-degenerate-tree.png)

The tree shown above is a degenerate tree. As far as the performance of this tree is concerned, the degenerate trees are the same as linked-lists.

#5) Balanced Binary Tree

A binary tree in which the depth of the two subtrees of every node never differs by more than 1 is called a balanced binary tree.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Balanced-binary-tree.png)

The binary tree shown above is a balanced binary tree as the depth of the two subtrees of every node is not more than 1. AVL trees which we are going to discuss in our subsequent tutorials are a common balanced tree.

Binary Tree Representation

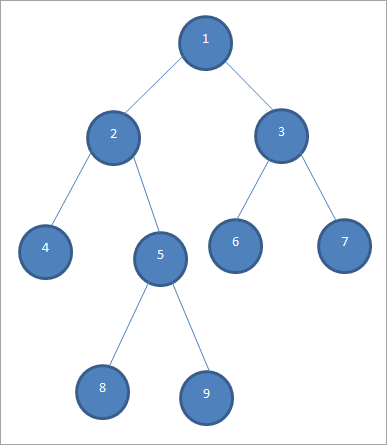
**A binary tree is allocated memory in two ways.**

#1) Sequential Representation

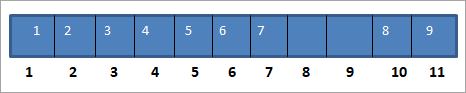
This is the simplest technique to store a tree data structure. An array is used to store the tree nodes. The number of nodes in a tree defines the size of the array. The root node of the tree is stored at the first index in the array.

In general, if a node is stored at the ith location then it’s left and right child is stored at 2i and 2i+1 location respectively.

**Consider the following Binary Tree.**

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/binary-tree-example.png)

**The sequential representation of the above binary tree is as follows:**

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/sequential-representation.png)

In the above representation, we see that the left and right child of each node is stored at locations 2\*(node\_location) and 2\*(node\_location)+1 respectively.

**For Example,** the location of node 3 in the array is 3. So it’s left child will be placed at 2\*3 = 6. Its right child will be at the location 2\*3 +1 = 7. As we can see in the array, children of 3 which are 6 and 7 are placed at location 6 and 7 in the array.

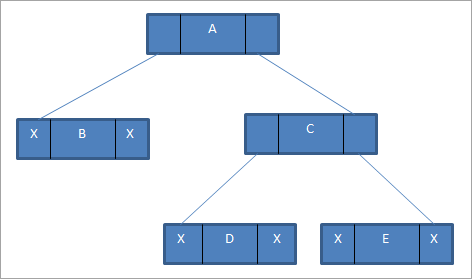
The sequential representation of the tree is inefficient as the array which is used to store the tree nodes takes up lots of space in memory. As the tree grows, this representation becomes inefficient and difficult to manage.

This drawback is overcome by storing the tree nodes in a linked list. Note that if the tree is empty, then the first location storing the root node will be set to 0.

#2) Linked-list Representation

In this type of representation, a linked list is used to store the tree nodes. Several nodes are scattered in the memory in non-contiguous locations and the nodes are connected using the parent-child relationship like a tree.

**Following diagram shows a linked list representation for a tree.**

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/Linked-list-representation.png)

**As shown in the above representation, each linked list node has three components:**

* Left pointer
* Data part
* Right pointer

The left pointer has a pointer to the left child of the node; the right pointer has a pointer to the right child of the node whereas the data part contains the actual data of the node. If there are no children for a given node (leaf node), then the left and right pointers for that node are set to null as shown in the above figure.

Binary Tree Implementation in C++

Next, we develop a binary tree program using a linked list representation in C++. We use a structure to declare a single node and then using a class, we develop a linked list of nodes.

|  |
| --- |
| #include<iostream>  using namespace std;    struct bintree\_node{      bintree\_node \*left;      bintree\_node \*right;      int data;  } ;  class bst{      bintree\_node \*root;      public:      bst(){          root=NULL;      }      int isempty() {          return(root==NULL);      }      void insert(int item);      void displayBinTree();      void printBinTree(bintree\_node \*);    };  void bst::insert(int item){      bintree\_node \*p=new bintree\_node;      bintree\_node \*parent;      p->data=item;      p->left=NULL;      p->right=NULL;      parent=NULL;      if(isempty())          root=p;      else{          bintree\_node \*ptr;          ptr=root;          while(ptr!=NULL){              parent=ptr;              if(item>ptr->data)                  ptr=ptr->right;              else                  ptr=ptr->left;          }          if(item<parent->data)              parent->left=p;          else              parent->right=p;      }  }  void bst::displayBinTree(){      printBinTree(root);  }  void bst::printBinTree(bintree\_node \*ptr){      if(ptr!=NULL){          printBinTree(ptr->left);          cout<<"  "<<ptr->data<<"     ";          printBinTree(ptr->right);      }  }  int main(){      bst b;      b.insert(20);      b.insert(10);      b.insert(5);      b.insert(15);      b.insert(40);      b.insert(45);      b.insert(30);        cout<<"Binary tree created: "<<endl;      b.displayBinTree();  } |

**Output:**

Binary tree created:

5       10       15       20       30       40       45

Binary Tree Traversal

We have already discussed traversals in our basic tutorial on trees. In this section, let us implement a program that inserts nodes in the binary tree and also demonstrates all the three traversals i.e. inorder, preorder and postorder, for a binary tree.

|  |
| --- |
| #include<iostream>  using namespace std;  //binary tree node declaration  struct bintree\_node{      bintree\_node \*left;      bintree\_node \*right;      char data;  } ;  class bintree\_class{      bintree\_node \*root;      public:      bintree\_class(){          root=NULL;      }      int isempty() {          return(root==NULL);      }      void insert\_node(int item);      void inorder\_seq();      void inorder(bintree\_node \*);      void postorder\_seq();      void postorder(bintree\_node \*);      void preorder\_seq();      void preorder(bintree\_node \*);  };  void bintree\_class::insert\_node(int item){      bintree\_node \*p=new bintree\_node;      bintree\_node \*parent;      p->data=item;      p->left=NULL;      p->right=NULL;      parent=NULL;      if(isempty())          root=p;      else{          bintree\_node \*ptr;          ptr=root;          while(ptr!=NULL)        {              parent=ptr;              if(item>ptr->data)                  ptr=ptr->right;              else                  ptr=ptr->left;          }          if(item<parent->data)              parent->left=p;          else              parent->right=p;      }  }  void bintree\_class::inorder\_seq()  {      inorder(root);  }  void bintree\_class::inorder(bintree\_node \*ptr)  {      if(ptr!=NULL){          inorder(ptr->left);          cout<<"  "<<ptr->data<<"     ";          inorder(ptr->right);      }  }  void bintree\_class::postorder\_seq()  {      postorder(root);  }  void bintree\_class::postorder(bintree\_node \*ptr)  {      if(ptr!=NULL){          postorder(ptr->left);          postorder(ptr->right);          cout<<"  "<<ptr->data<<"     ";      }  }  void bintree\_class::preorder\_seq()  {      preorder(root);  }  void bintree\_class::preorder(bintree\_node \*ptr)  {      if(ptr!=NULL){          cout<<"  "<<ptr->data<<"     ";          preorder(ptr->left);          preorder(ptr->right);      }  }  int main()  {      bintree\_class bintree;      bintree.insert\_node('A');      bintree.insert\_node('B');      bintree.insert\_node('C');      bintree.insert\_node('D');      bintree.insert\_node('E');      bintree.insert\_node('F');      bintree.insert\_node('G');      cout<<"Inorder traversal:"<<endl;      bintree.inorder\_seq();      cout<<endl<<"Postorder traversal:"<<endl;      bintree.postorder\_seq();      cout<<endl<<"Preorder traversal:"<<endl;      bintree.preorder\_seq();  } |

**Output:**

Inorder traversal:

A       B       C       D       E       F       G

Postorder traversal:

G       F       E       D       C       B       A

Preorder traversal:

A       B       C       D       E       F       G

Applications Of Binary Tree

A binary tree is used in many applications for storing data.

**Some of the important applications of binary trees are listed below:**

* **Binary Search Trees:**Binary trees are used to construct a binary search tree that is used in many searching applications like sets and maps in many language libraries.
* **Hash Trees:**Used to verify hashes mainly in specialized image signature applications.
* **Heaps:**Heaps are used for implementing priority queues that are used for routers, scheduling processors in the operating system, etc.
* **Huffman Coding:**Huffman coding tree is used in compression algorithms (like image compression) as well as cryptographic applications.
* **Syntax Tree:**Compilers often construct syntax trees which are nothing but binary trees to parse expressions used in the program.

Conclusion

Binary trees are widely used data structures across the software industry. Binary trees are the trees whose nodes have at most two child nodes. We have seen various types of binary trees like a full binary tree, a complete binary tree, a perfect binary tree, a degenerated binary tree, a balanced binary tree, etc.

Binary tree data can also be traversed using inorder, preorder and postorder traversal techniques which we have seen in our previous tutorial. In memory, a binary tree can be represented using a linked list (non-contiguous memory) or arrays (sequential representation).

Linked list representation is more efficient when compared to arrays, as arrays take up a lot of space.