

1. Definition and Tree Terminologies

Introduction

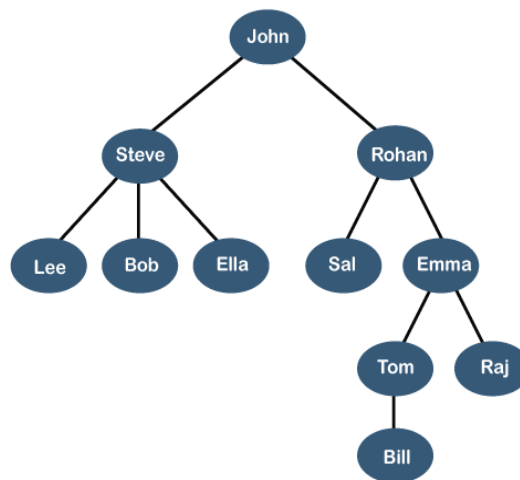
Trees are non linear data structures and are used to impose a hierarchical structure on a collection of data items. For example, we need to impose a hierarchical structure on a collection of data items while preparing organizational charts and to represent the syntactic structure of a source program in compilers.

Definition of a Tree

A tree is a set of one or more nodes T such that:

- i. there is a specially designated node called a root
- ii. The remaining nodes are partitioned into n disjoint set of nodes T_1, T_2, \dots, T_n , each of which is a tree.

A **tree** is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



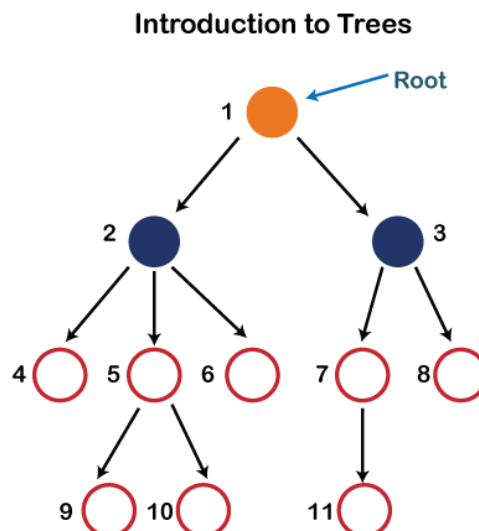
The above tree shows the **organization hierarchy** of some company. In the above structure, **john** is the **CEO** of the company, and John has two direct reports named as **Steve** and **Rohan**. Steve has three direct reports named **Lee**, **Bob**, **Ella** where **Steve** is a manager. Bob has two direct reports named **Sal** and **Emma**. **Emma** has two direct reports named **Tom** and **Raj**. Tom has one direct report named **Bill**. This particular logical structure is known as a **Tree**. Its structure is similar to the real tree, so it is named a **Tree**. In this structure, the **root** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Terminologies

Let's consider the tree structure, which is shown below:



In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a **link** between the two nodes.

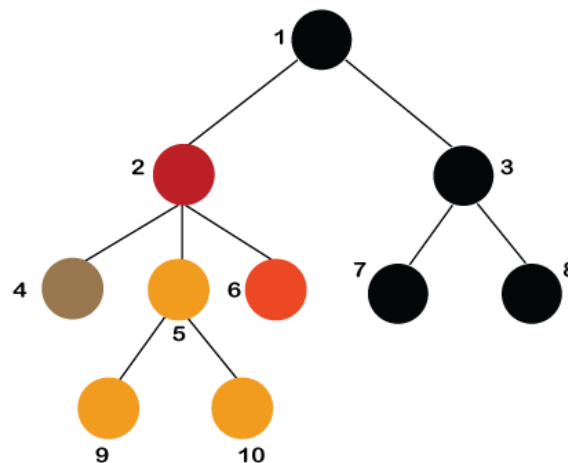
- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree**. If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an **internal node**.

Unit 4: Tree

- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a *recursive data structure*. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a **root node**. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.



- **Number of edges:** If there are n nodes, then there would be $n-1$ edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have at least one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x . One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x . The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

2. General Trees

2.1. Definition and their Applications

Many organizations are hierarchical in nature, such as the military and most businesses. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, and so on. If we wanted to model this company with a data structure, it would be natural to think of the president in the root node of a tree, the vice presidents at level 1, and their subordinates at lower levels in the tree as we go down the organizational hierarchy.

Because the number of vice presidents is likely to be more than two, this company's organization cannot easily be represented by a binary tree. We need instead to use a tree whose nodes have an arbitrary number of children. Unfortunately, when we permit trees to have nodes with an arbitrary number of children, they become much harder to implement than binary trees. To distinguish them from binary trees, we use the term general tree.

A general tree is a tree where each node may have zero or more children (a binary tree is a specialized case of a general tree).

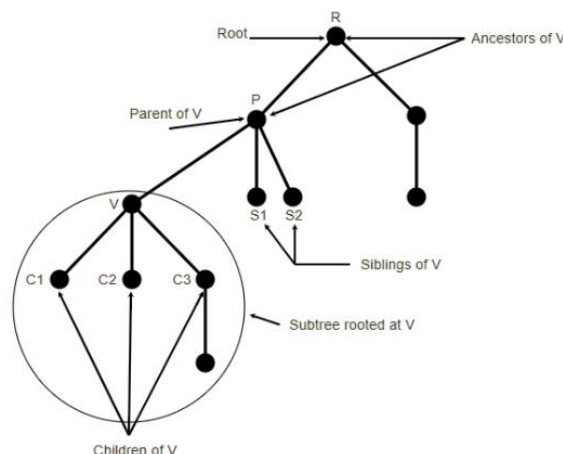


Figure 6.6: Notation for general trees. Node P is the parent of nodes V, S1, and S2. Thus, V, S1, and S2 are children of P. Nodes R and P are ancestors of V. Nodes V, S1, and S2 are called siblings. The oval surrounds the subtree having V as its root.

Each node in a tree has precisely one parent, except for the root, which has no parent. From this observation, it immediately follows that a tree with n nodes must have $n-1$ edges because each node, aside from the root, has one edge connecting that node to its parent.

Application:

1. **File Systems:** General trees are used to represent the hierarchical structure of directories and files in a computer's file system.
2. **Organization Charts:** General trees are employed to represent the hierarchical structure of organizations, including departments, teams, and reporting relationships.
3. **XML/HTML Parsing:** General trees are used to parse and represent the structure of XML and HTML documents, enabling efficient processing and manipulation.
4. **Decision Trees:** General trees are used in machine learning for constructing decision trees, which are used for classification and regression tasks.
5. **Abstract Syntax Trees (AST):** General trees are utilized in compilers and programming languages to represent the syntax and structure of source code.
6. **Game Trees:** General trees are used in game theory and artificial intelligence to represent game states and possible moves in games.
7. **Data Structures:** General trees serve as the foundation for various data structures like heaps, tries, and suffix trees, providing efficient storage and retrieval mechanisms.

2.2. Game Tree

A game tree is a type of general tree that represents the possible states and moves in a game. It is commonly used in game theory and artificial intelligence to analyze and strategize game-playing algorithms.

In a game tree, each node represents a specific state of the game, and the edges represent possible moves or transitions between states. The root of the tree represents the initial state of the game, and the leaf nodes represent the terminal states, where the game ends.

The branching factor of the tree represents the number of possible moves or actions available in each game state. For example, in chess, the branching factor would represent the number of possible moves for a given chess position.

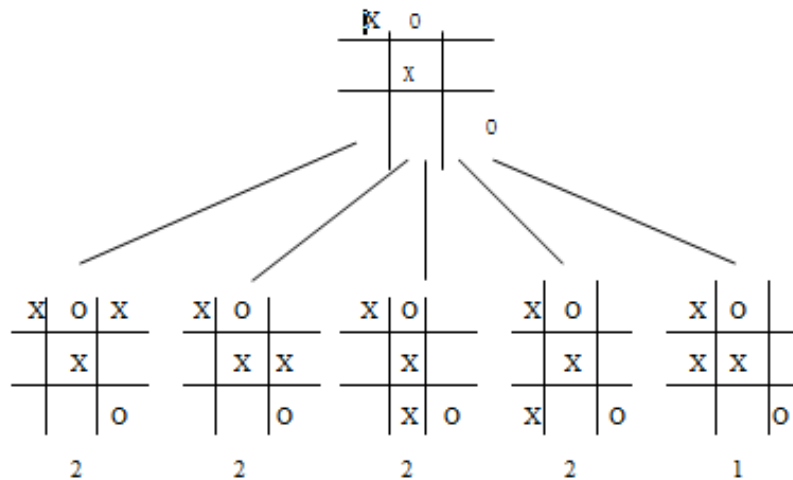
To construct a game tree, the algorithm starts with the initial state and explores all possible moves from that state, creating child nodes for each move. This process continues recursively for each subsequent state, expanding the tree further until reaching the terminal states.

Game trees can be either finite or infinite, depending on the nature of the game. In finite games, such as chess or tic-tac-toe, the tree has a finite number of nodes because the game eventually reaches a terminal state. In infinite games, such as certain card games or board games with random elements, the tree can continue indefinitely as new states and moves are generated.

The game tree provides a systematic way to explore and analyze all possible game paths and outcomes. It enables algorithms to search for optimal moves, evaluate the quality of different strategies, and even determine the overall game-theoretic value of the game.

Common algorithms used with game trees include minimax, alpha-beta pruning, and Monte Carlo Tree Search (MCTS), which help in efficient traversal and evaluation of the tree to make informed decisions in game-playing scenarios.

Eg. Tic-tac-toe



Overall, game trees serve as a powerful tool for analyzing and strategizing games, allowing players and AI agents to make informed decisions by exploring the potential outcomes of different moves and strategies.

3. Binary Trees

3.1. Definition and Types

A general tree has no restrictions on the number of children each node can have. Binary tree is a special type of tree data structure and has a restriction on number of children of a node. Every node in a binary can have maximum of two children. The children are referred to as left child or right child.

In binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children. This is one of the most commonly used trees.

A binary tree is made up of a finite set of elements called nodes. This set either is empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root. Disjoint means that they have no nodes in common. The roots of these subtrees are children of the root.

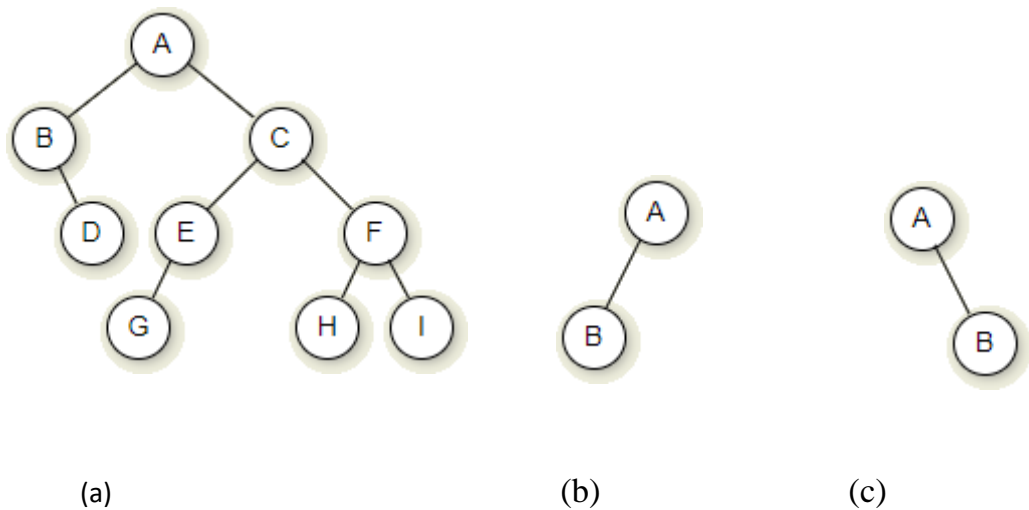
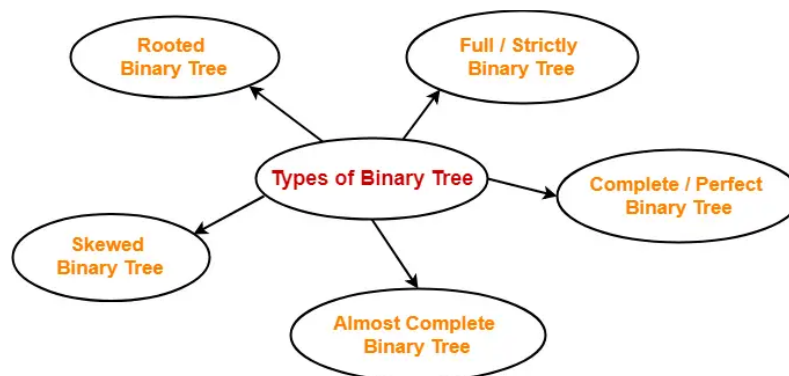


Figure 6.7: A binary Trees

Types of Binary Trees-

Binary trees can be of the following types-



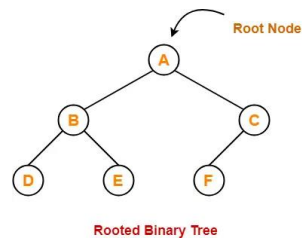
1. Rooted Binary Tree
2. Full / Strictly Binary Tree
3. Complete / Perfect Binary Tree
4. Almost Complete Binary Tree
5. Skewed Binary Tree

1. Rooted Binary Tree-

A **rooted binary tree** is a binary tree that satisfies the following 2 properties-

- It has a root node.
- Each node has at most 2 children.

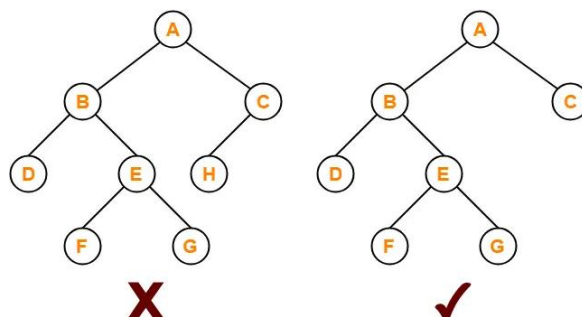
Example-



2. Full / Strictly Binary Tree-

- A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.
- Full binary tree is also called as **Strictly binary tree**.

Example-

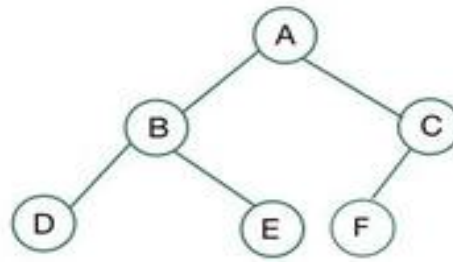


Here,

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

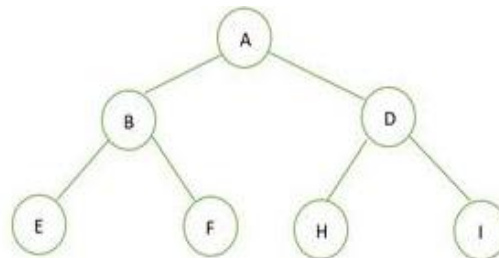
3. Complete Binary Tree

A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.



4. Perfect Binary Tree

- A complete binary tree is a binary tree type where all the leaf nodes must be on the same level
- Every internal node has exactly 2 children.

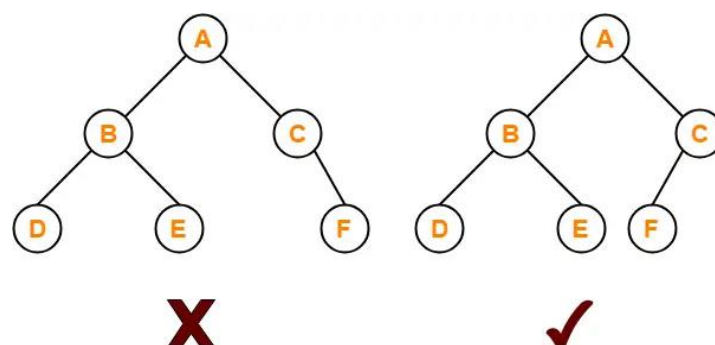


5. Almost Complete Binary Tree-

An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.

Example-



Here,

- First binary tree is not an almost complete binary tree.
- This is because the last level is not filled from left to right.

6. Skewed Binary Tree-

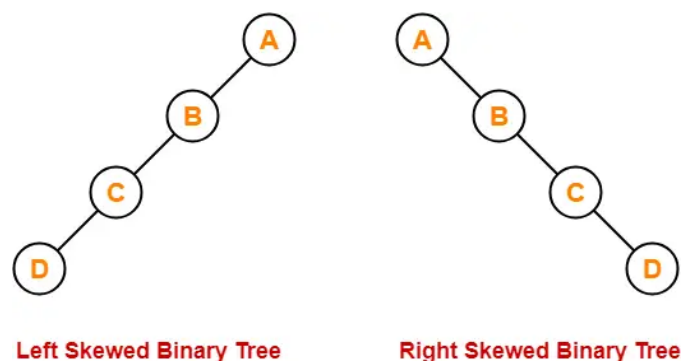
A **skewed binary tree** is a binary tree that satisfies the following 2 properties-

- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

A **skewed binary tree** is a binary tree of n nodes such that its depth is $(n-1)$.

Example-



3.2. Array and Linked List Representation

Array Representation of Binary Tree

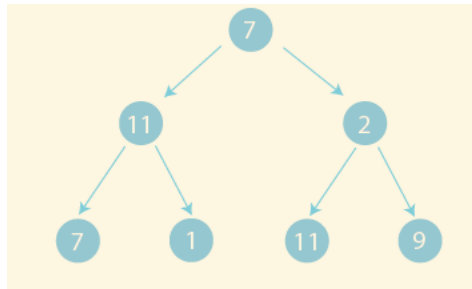
The array representation of binary tree can be done using a technique called level order traversal. In level-order traversal, the elements of the binary tree are stored in the array in the order in which they are visited in a breadth-first search.

The array representation of binary tree allows for efficient access to the elements of the tree. For example, if a binary tree has n nodes, the array representation of the tree can be stored in an array of size n , allowing for constant-time access to each node in the tree.

In this representation, the root node of the binary tree is stored at the first position of the array, and its left and right children are stored at the second and third positions, respectively. The remaining nodes are stored in the same way, with the children of each node stored in consecutive positions in the array.

For example, consider the following binary tree:

Unit 4: Tree



The level order representation of this tree in an array would be [1,11,2,7,1,11,9].

Using this representation, it is possible to access the children of a node at position i by using the following formulas:

The left child of node i can be found at position $2i + 1$

The right child of node i can be found at position $2i + 2$

It is also possible to determine the parent of a node at position i using the formula:

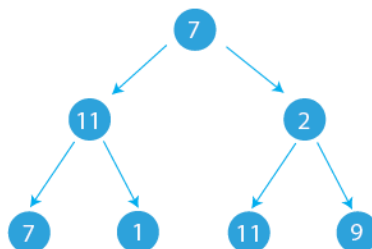
The parent of node i can be found at position $(i - 1) / 2$

The array representation of binary tree can be useful in algorithms that require fast access to the elements of a tree, such as searching or sorting algorithms. However, it is important to note that not all binary trees can be efficiently represented in an array, as the number of elements in the array must be known in advance. Additionally, some algorithms may require a different representation of a binary tree, such as a linked list representation.

Dry Run of Array Representation of Binary Tree with Example

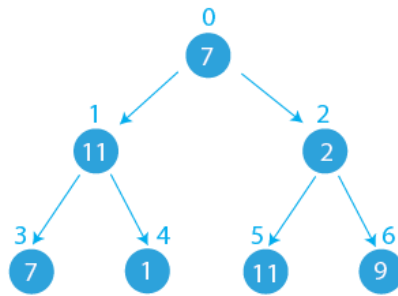
For the array representation of binary tree, we will form an array of size $2^{n+1} - 1$ size where n is the number of nodes the binary tree. Now we will move step by step for the array representation of binary tree.

1. First, suppose we have a binary tree with seven nodes



2. Now, for the array representation of binary tree, we have to give numbering to the corresponding nodes. The standard form of numbering the nodes is to start from the root node and move from left to right at every level. After numbering the tree and nodes will look like this:

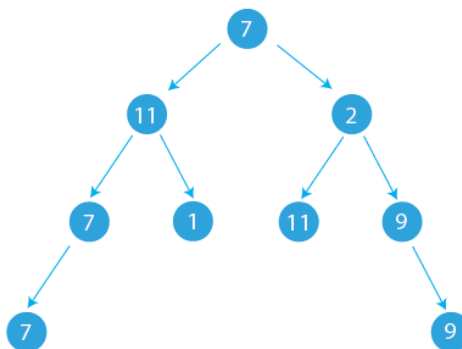
Unit 4: Tree



3. Now as we have numbered from zero you can simply place the corresponding number in the matching index of an array then the array will look like this:

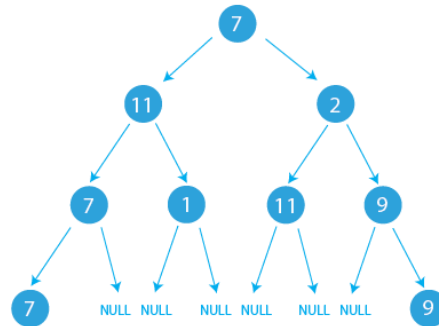
0	1	2	3	4	5	6
7	11	2	7	1	11	9

4. That is the array representation of binary tree but this is the easy case as every node has two child so what about other cases? We will discuss other cases below.
5. In these cases, we will discuss the scenarios for the array representation of binary tree where there the lead node is not always on the last level.
6. So consider the binary tree given below:

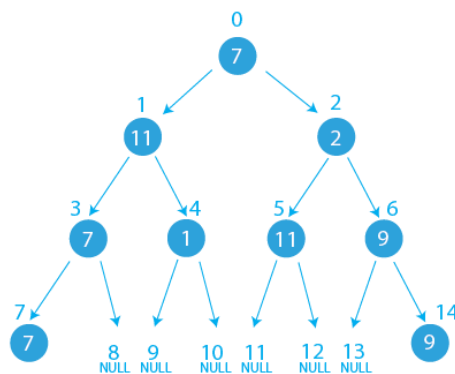


7. While giving a number you are stuck with the cases where you encounter a leaf node so just make the child node of the leaf nodes as NULL then the tree will look like this:

Unit 4: Tree



8. Now just number the nodes as done above for the array representation of binary tree after that the tree will look like this:

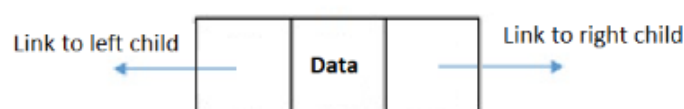


9. Now we have the number on each node we can easily use the tree for array representation of binary tree and the array will look like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	11	2	7	1	11	9	7							9

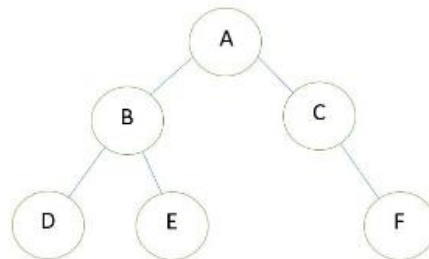
Linked List Representation of Binary Tree

We can represent the binary tree using linked list data structure. For this each node in linked list represent a node of the binary tree. Each list node needs three fields. One field to hold data and the other two fields to hold the addresses of the left child node and right child node.

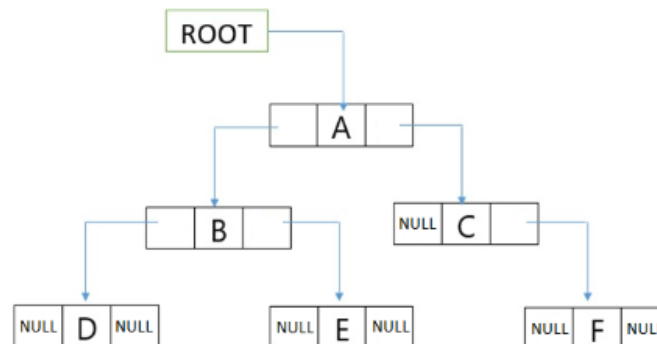


Unit 4: Tree

This node can be recursively used to represent a binary tree. If a node doesn't have a left child, it will be assigned a null value. If the node doesn't have a right child, it will be assigned a null value. Consider a binary tree as:



This binary tree is represented using linked list as shown as



Node representation in C++

```
struct Node {  
    int value;  
    Node* left;  
    Node* right;  
  
    Node(int val) {  
        value = val;  
        left = nullptr;  
        right = nullptr;  
    }  
};
```

In this code, the Node struct represents a node in a binary tree. It has three member variables: value, left, and right. The value variable holds the value of the node, while left and right are pointers to the left and right child nodes, respectively.

The constructor `Node(int val)` initializes the node with the given value and sets the `left` and `right` pointers to `nullptr` to indicate that the node has no children initially.

You can use this `Node` struct to create a binary tree by creating instances of `Node` and linking them together using the `left` and `right` pointers.

3.3. Traversal Algorithms: pre-order, in-order and post- order traversal

Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a traversal. Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship.

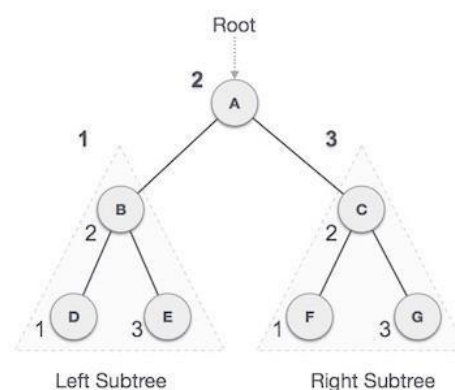
There are three ways of traversing a tree:

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be –

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

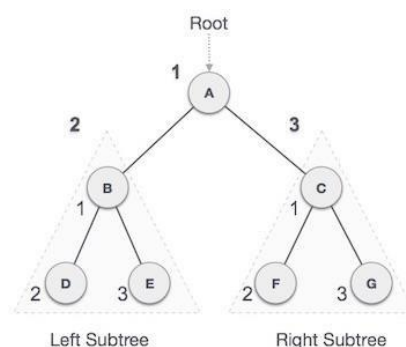
Step 3 – Recursively traverse right subtree.

```
void inOrderTraversal(Node* node) {  
    if (node == nullptr) {  
        return;  
    }  
  
    // Traverse the left subtree  
    inOrderTraversal(node->left);  
  
    // Process the value of the current node  
    cout << node->value << " ";  
  
    // Traverse the right subtree  
    inOrderTraversal(node->right);  
}
```

To use the above code, you would need to create a binary tree and call the `inOrderTraversal` function, passing the root node of the tree as an argument. The function will then perform an in-order traversal, printing (or processing) the values of the tree's nodes in the correct order.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed –

Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

```
void preorderTraversal(Node* node) {
    if (node == nullptr) {
        return;
    }

    // Process the value of the current node
    cout << node->value << " ";

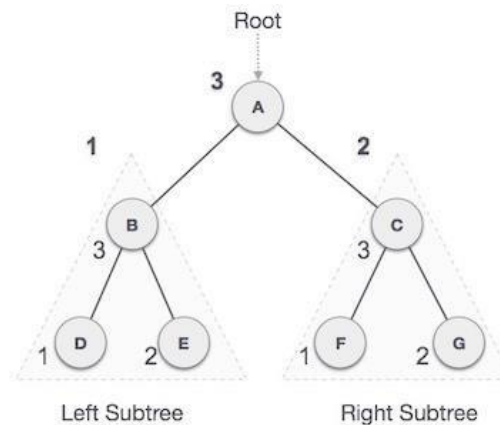
    // Traverse the left subtree
    preorderTraversal(node->left);

    // Traverse the right subtree
    preorderTraversal(node->right);
}
```

To use the above code, you would need to create a binary tree and call the `preorderTraversal` function, passing the root node of the tree as an argument. The function will then perform a preorder traversal, printing (or processing) the values of the tree's nodes in the correct order.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

```
void postOrderTraversal(Node* node) {
    if (node == nullptr) {
        return;
    }

    // Traverse the left subtree
    postOrderTraversal(node->left);

    // Traverse the right subtree
    postOrderTraversal(node->right);

    // Process the value of the current node
    cout << node->value << " ";
}
```

To use the above code, you would need to create a binary tree and call the `postOrderTraversal` function, passing the root node of the tree as an argument. The function will then perform a post-order traversal, printing (or processing) the values of the tree's nodes in the correct order.

Program to implement preorder, inorder and postorder traversals.

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

struct Node {
    int value;
    Node* left;
    Node* right;

    // Constructor
    Node(int val) {
        value = val;
        left = NULL;
        right = NULL;
    }
};

void inOrderTraversal(Node* node) {
    if (node == NULL) {
        return;
    }

    // Traverse the left subtree
    inOrderTraversal(node->left);

    // Process the value of the current node
    cout << node->value << " ";

    // Traverse the right subtree
    inOrderTraversal(node->right);
}

void preorderTraversal(Node* node) {
    if (node == NULL) {
        return;
    }

    // Process the value of the current node
    cout << node->value << " ";

    // Traverse the left subtree
    preorderTraversal(node->left);

    // Traverse the right subtree
    preorderTraversal(node->right);
}
```

```
}

void postOrderTraversal(Node* node) {
    if (node == NULL) {
        return;
    }

    // Traverse the left subtree
    postOrderTraversal(node->left);

    // Traverse the right subtree
    postOrderTraversal(node->right);

    // Process the value of the current node
    cout << node->value << " ";
}

int main() {
    // Create a binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    // Perform in-order traversal
    cout << "In-order traversal: ";
    inOrderTraversal(root);
    cout << endl;

    // Perform pre-order traversal
    cout << "Pre-order traversal: ";
    preorderTraversal(root);
    cout << endl;

    // Perform post-order traversal
    cout << "Post-order traversal: ";
    postOrderTraversal(root);
    cout << endl;

    return 0;
}
```

3.4. Application of Full Binary Tree: Huffman algorithm

A full binary tree is a special type of binary tree in which every node has either zero or two children. It has several applications in computer science and algorithms. Here are a few notable applications of full binary trees:

1. **Heap Data Structure:** Full binary trees are commonly used as the underlying data structure for implementing heap data structures such as binary heaps and binary search heaps. The structure and properties of a full binary tree make it efficient for heap operations like insertion, deletion, and heapify.
2. **Huffman Coding:** Huffman coding is a widely used data compression algorithm. In the process of constructing Huffman trees, full binary trees are utilized. The Huffman tree is a full binary tree where each leaf node represents a character, and the path from the root to a leaf node determines the binary code assigned to that character.
3. **Binary Search Tree:** A binary search tree (BST) is a binary tree with a specific ordering property: for any node, all the values in its left subtree are less than its value, and all the values in its right subtree are greater than its value. A full binary tree can be used as the underlying structure for implementing a binary search tree efficiently.
4. **Expression Trees:** In the domain of expression evaluation and parsing, expression trees are used to represent arithmetic or logical expressions. Full binary trees are often employed to construct expression trees, where the internal nodes represent operators, and the leaf nodes represent operands.
5. **Binary Decision Diagrams (BDDs):** BDDs are data structures commonly used in computer-aided design (CAD), verification, and optimization of digital circuits. Full binary trees are utilized to represent BDDs, where the internal nodes represent Boolean variables, and the edges correspond to the Boolean functions AND, OR, and NOT.
6. **Perfect Binary Trees:** Perfect binary trees are a type of full binary tree where all the levels are completely filled with nodes. Perfect binary trees have applications in indexing structures like B-trees and space partitioning algorithms like quad trees and kd-trees.

These are just a few examples highlighting the applications of full binary trees in various areas of computer science. The unique properties and structure of full binary trees make them useful in many algorithms and data structures.

Huffman Algorithm

Huffman coding is an algorithm for compressing data with the aim of reducing its size without losing any of the details. This algorithm was developed by **David Huffman**.

Huffman coding is typically useful for the case where data that we want to compress has frequently occurring characters in it. Huffman coding is a specific implementation of variable-length encoding. It is a well-known algorithm for constructing optimal prefix codes based on variable-length encoding principles. Huffman coding assigns shorter codes to more frequent symbols and longer codes to less frequent symbols, thereby achieving efficient data compression.

How it works

Let assume the string data given below is the data we want to compress -



The length of the above string is 15 characters and each character occupies a space of 8 bits. Therefore, a total of 120 bits (8 bits x 15 characters) is required to send this string over a network. We can reduce the size of the string to a smaller extent using Huffman Coding Algorithm.

In this algorithm first we create a tree using the frequencies of characters and then assign a code to each character. The same resulting tree is used for decoding once encoding is done.

Using the concept of prefix code, this algorithm avoids any ambiguity within the decoding process, i.e. a code assigned to any character shouldn't be present within the prefix of the opposite code.

Steps to Huffman Coding

1. First, we calculate the count of occurrences of each character in the string.

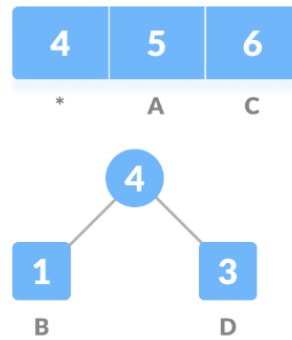
1	6	5	3
B	C	A	D

2. Then we sort the characters in the above string in increasing order of the count of occurrence. Here we use PriorityQueue to store.

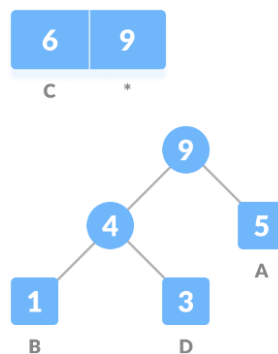
1	3	5	6
B	D	A	C

3. Now we mark every unique character as a Leaf Node.
4. Let's create an empty node n. Add characters having the lowest count of occurrence as the left child of n and second minimum count of occurrence as the right child of n, then assign the sum of the above two minimum frequencies to **n**.

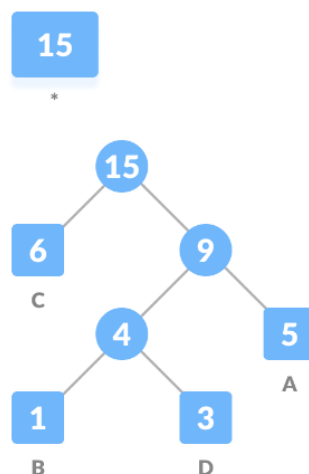
Unit 4: Tree



- Now remove these two minimum frequencies from Queue and append the sum into the list of frequencies.
- Add node **n** into the tree.
- Just like we did for **B** and **D**, we repeat the same steps from 3 to 5 for the rest of the characters (**A** and **C**). For A -

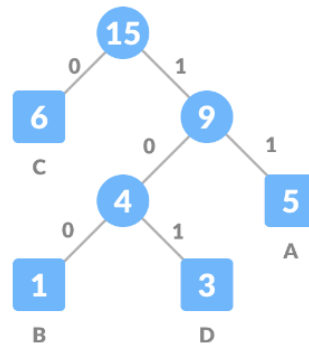


Repeat for C -



- We got our resulting tree, now we assign **0 to the left edge** and **1 to the right edge** of every non-leaf node.

Unit 4: Tree



9. Now for generating codes of each character we traverse towards each leaf node representing some character from the root node and form code of it.

All the data we gathered until now is given below in tabular form -

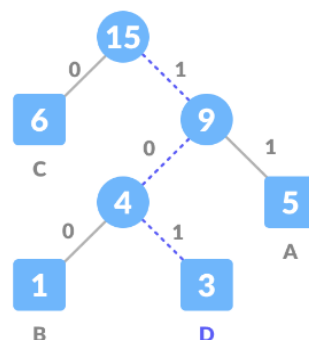
Character	Frequency in string	Assigned Code	Size
B	1	100	$1 \times 3 = 3$ bits
D	3	101	$3 \times 3 = 9$ bits
A	5	11	$5 \times 2 = 10$ bits
C	6	0	$6 \times 1 = 6$ bits
$4 \times 8 = 32$ bits	Total = 15 bits		Total = 28 bits

Before compressing the total size of the string was **120 bits**. After compression that size was reduced to **75 bits** (28 bits + 15 bits + 32 bits).

Steps to Huffman Decoding

To decode any code, we take the code and traverse it in the tree from the root node to the leaf node, each code will make us reach a unique character.

Let assume code **101** needs to be decoded, for this we will traverse from the root as given below -



As per the Huffman encoding algorithm, for **every 1 we traverse towards the right child** and for **0 we traverse towards the left one**, if we follow this and traverse, we will **reach leaf node 3 which represents D**. Therefore, **101 is decoded to D**.

Algorithm

```
create and initialize a PriorityQueue Queue consisting of each
unique character.
sort in ascending order of their frequencies.
for all the unique characters:
    create a new_node
    get minimum_value from Queue and set it to left child of
    new_node
    get minimum_value from Queue and set it to right child of
    new_node
    calculate the sum of these two minimum values as
    sum_of_two_minimum
    assign sum_of_two_minimum to the value of new_node
    insert new_node into the tree
return root_node
```

4. Binary Search Tree:

4.1. Definition and Operations on Binary Search Tree: insertion, deletion, searching and traversing

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

Advantages of Binary search tree

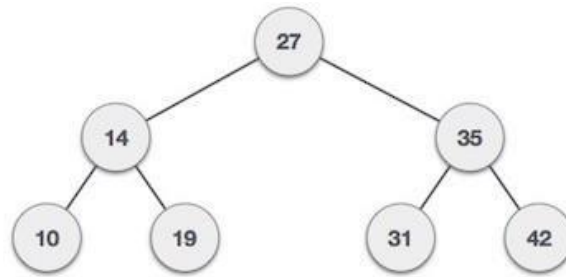
- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Unit 4: Tree

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

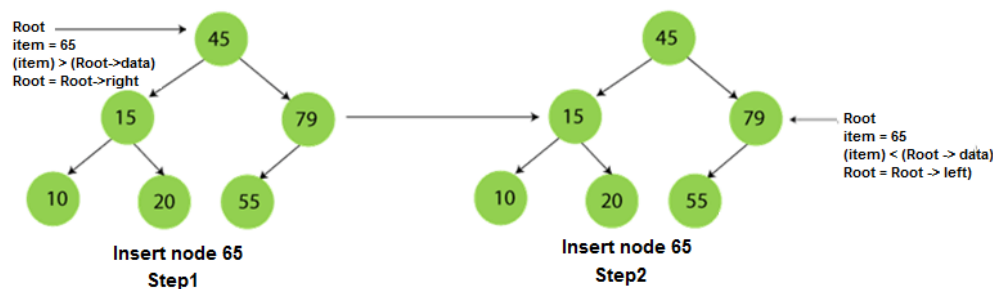
Following are the basic operations of a tree –

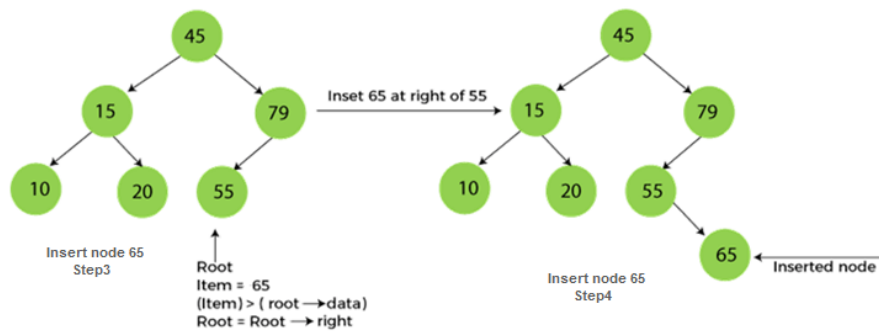
- **Insert** – Inserts an element in a tree.
- **Delete** – Deletes an element from a tree.
- **Search** – Searches an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.





Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

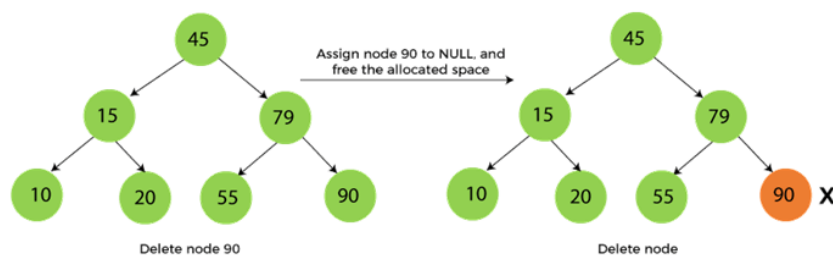
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

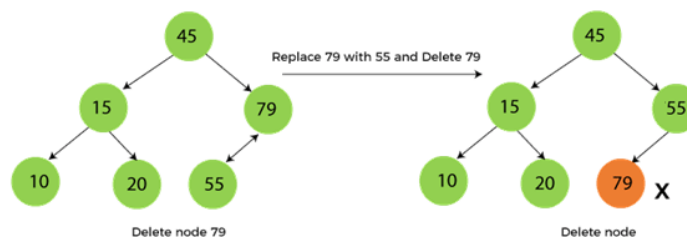


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



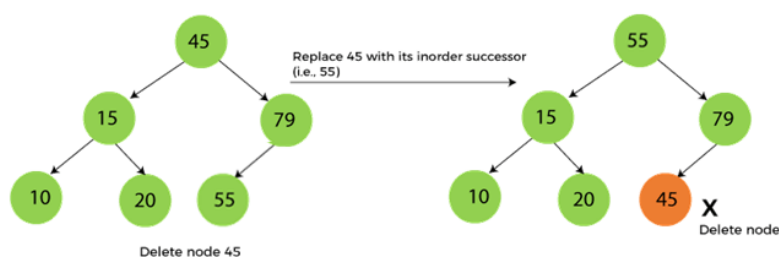
When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



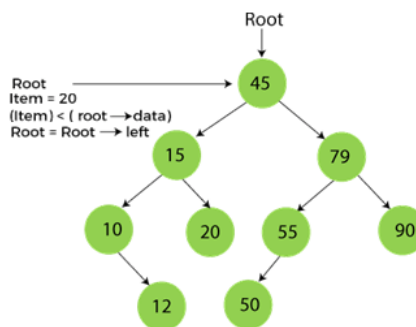
Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

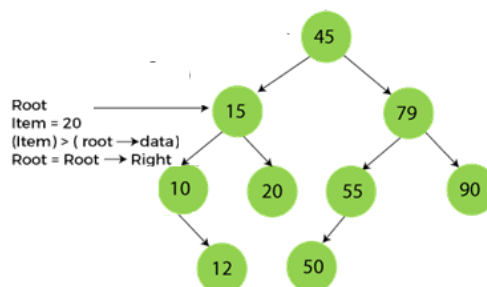
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

Step1:



Step2:



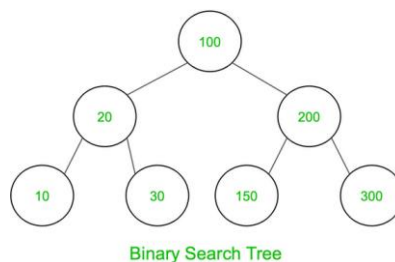
Step3:



Algorithm to search an element in Binary search tree

1. Search (root, item)
2. Step 1 - if (item = root \rightarrow data) or (root = NULL)
3. return root
4. else if (item < root \rightarrow data)
5. return Search(root \rightarrow left, item)
6. else
7. return Search(root \rightarrow right, item)
8. END if
9. Step 2 - END

Traversal:



Inorder Traversal: 10 20 30 100 150 200 300

Preorder Traversal: 100 20 10 30 200 150 300

Postorder Traversal: 10 30 20 150 300 200 100

The complexity of the Binary Search tree

Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	$O(\log n)$	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$	$O(n)$

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

Operations	Space complexity
Insertion	$O(n)$
Deletion	$O(n)$
Search	$O(n)$

- The space complexity of all operations of Binary search tree is $O(n)$.

4.2. Construction of Binary Search Tree

Suppose the data elements are - **45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

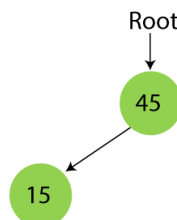
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.



Step 2 - Insert 15.

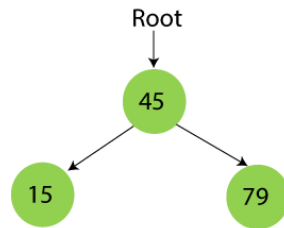
As 15 is smaller than 45, so insert it as the root node of the left subtree.



Step 3 - Insert 79.

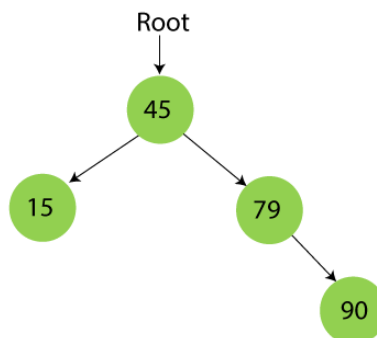
As 79 is greater than 45, so insert it as the root node of the right subtree.

Unit 4: Tree



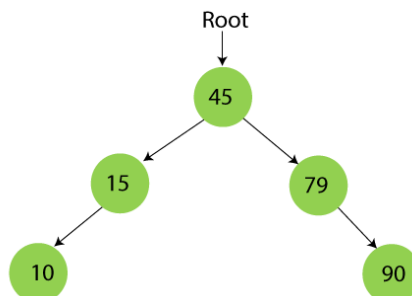
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



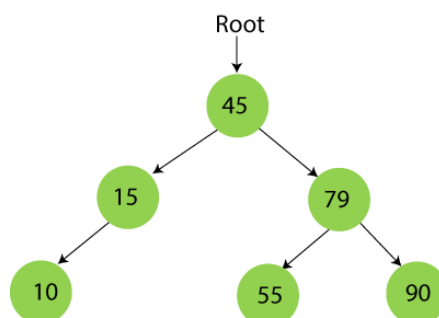
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



Step 6 - Insert 55.

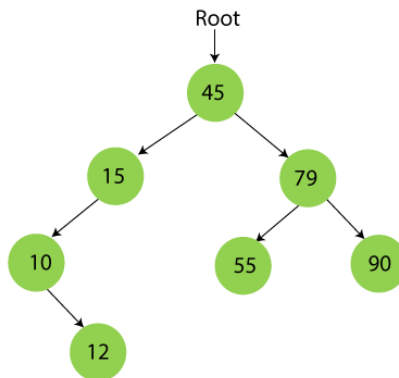
55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



Unit 4: Tree

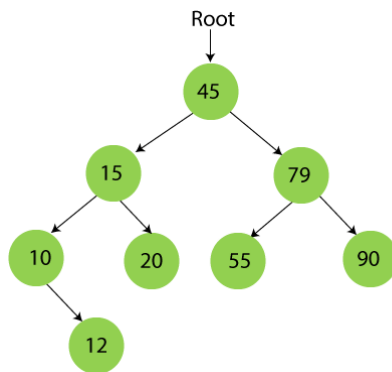
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



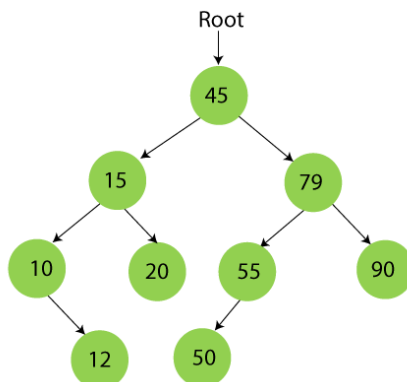
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.

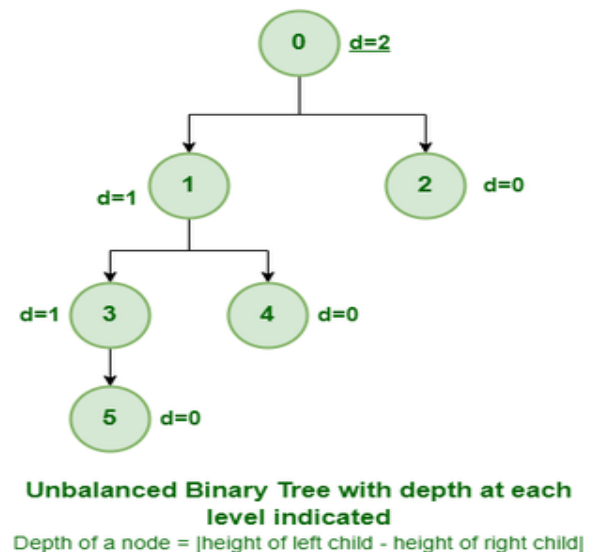
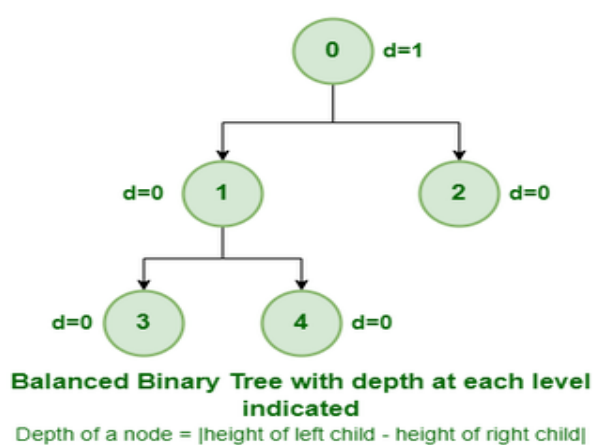


5. Balanced Binary Tree

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:

- difference between the left and the right subtree for any node is not more than one
- the left subtree is balanced
- the right subtree is balanced



Application of Balanced Binary Tree:

- AVL Trees
- Red Black Tree
- Balanced Binary Search Tree

5.1. Problem with unbalanced binary trees

An unbalanced binary tree is a binary tree where the heights of the left and right subtrees of any node differ significantly. In other words, the tree is lopsided or skewed, with more nodes on one side than the other. This imbalance can lead to various problems and inefficiencies. Here are some of the issues associated with unbalanced binary trees:

- **Increased search time:** In a balanced binary tree, the height is logarithmic in the number of nodes, ensuring efficient searching operations. However, in an unbalanced binary tree, the height can approach the number of nodes in the worst case, resulting in slower search times. As a result, operations like searching for a specific element or inserting a new element may take longer.
- **Degraded performance:** Unbalanced trees can cause performance degradation for various operations. For example, if the tree is heavily skewed to the left or right, traversing the tree to perform tasks such as sorting or finding the minimum/maximum element becomes less efficient. The skewed structure reduces the benefits of binary tree operations, rendering them similar to linear data structures.
- **Memory inefficiency:** Unbalanced binary trees can waste memory due to uneven distribution of nodes. A balanced binary tree ensures an even distribution of elements, with the height logarithmic to the number of nodes. However, in an unbalanced tree, nodes may be concentrated on one side, leading to inefficient memory utilization.
- **Unpredictable behavior:** The performance of operations on unbalanced binary trees can become unpredictable depending on the specific tree structure. Certain operations that perform well on balanced trees, such as self-balancing or rotation operations, may not effectively improve the tree's balance in unbalanced scenarios. As a result, the behavior of algorithms and operations can become less predictable and reliable.

To mitigate these issues, it is common to use self-balancing binary tree data structures, such as AVL trees or red-black trees. These data structures automatically balance themselves during insertions and deletions, ensuring that the tree remains balanced and efficient for various operations.

5.2. Balanced Binary Search Tree

A balanced binary search tree is a type of binary search tree (BST) that maintains a balanced structure to ensure efficient search, insertion, and deletion operations. In a balanced BST, the heights of the left and right subtrees of any node differ by at most one.

The balancing property of a balanced BST helps to prevent the tree from becoming heavily skewed or lopsided, which can occur in regular binary search trees. When a BST becomes unbalanced, the efficiency of search operations deteriorates, as the height of the tree increases, leading to longer search times.

There are various types of balanced binary search trees, with the most common ones being AVL trees, red-black trees, and B-trees. These trees maintain balance through different techniques and algorithms, but they all aim to keep the tree height logarithmic to the number of nodes.

The balancing algorithms employed in balanced BSTs involve performing rotations, restructuring, or other operations on the tree to maintain or restore balance during insertions and deletions. These operations ensure that the tree remains balanced and maintains its efficient search properties.

The benefits of using balanced binary search trees include:

- **Efficient search:** The balanced structure of the tree ensures that the height is logarithmic to the number of nodes, resulting in efficient search operations. This property allows for quick retrieval of elements based on their keys, making balanced BSTs suitable for applications that require fast search times.
- **Balanced insertions and deletions:** Balanced BSTs perform operations such as rotations or restructuring to maintain or restore balance after insertions or deletions. This ensures that the tree remains balanced, preventing degradation of search performance over time.
- **Predictable performance:** Balanced BSTs provide predictable and consistent performance characteristics. The worst-case time complexity for search, insertion, and deletion operations in a balanced BST is $O(\log n)$, where n is the number of nodes. This predictability is crucial for time-critical applications and algorithms.
- **Memory efficiency:** Balanced BSTs offer efficient memory utilization. Unlike some other data structures, such as hash tables, they do not require a fixed amount of memory and can dynamically adjust their structure as elements are inserted or deleted. This flexibility makes them suitable for situations where memory resources are limited or dynamically changing.
- **Versatility:** Balanced BSTs can be used in various applications that involve ordered data or key-value pairs. They are commonly employed in database systems, language compilers, symbol tables, file systems, and other scenarios that require efficient storage, retrieval, and manipulation of data.

Overall, balanced binary search trees combine the advantages of binary search trees with the benefits of maintaining a balanced structure. They provide efficient search operations, predictable performance, and effective memory utilization, making them a valuable data structure in many applications.

6. AVL tree

6.1. Definition and Need of AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

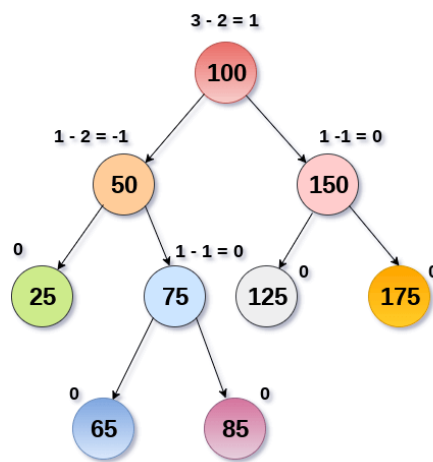
AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Need of AVL Tree

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is $O(h)$. However, it can be extended to $O(n)$ if the BST becomes skewed (i.e. worst case). By limiting this height to $\log n$, AVL tree imposes an upper bound on each operation to be $O(\log n)$ where n is the number of nodes.

6.2. Construction of AVL tree: Insertion, Deletion on AVL tree and Rotation Operations

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

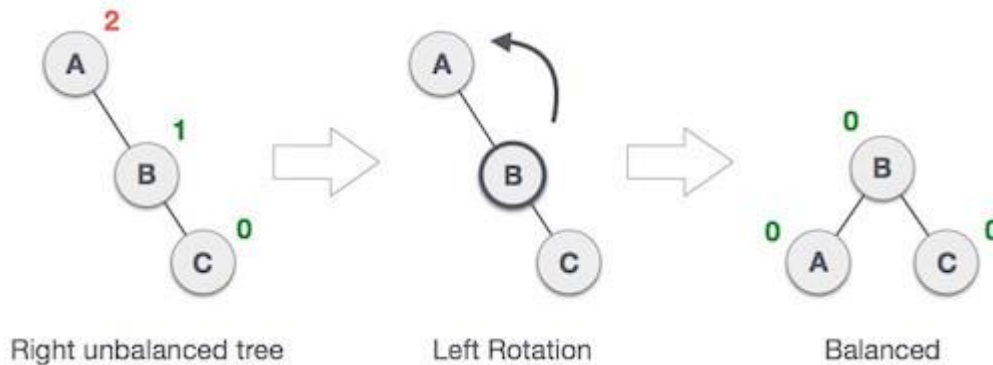
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

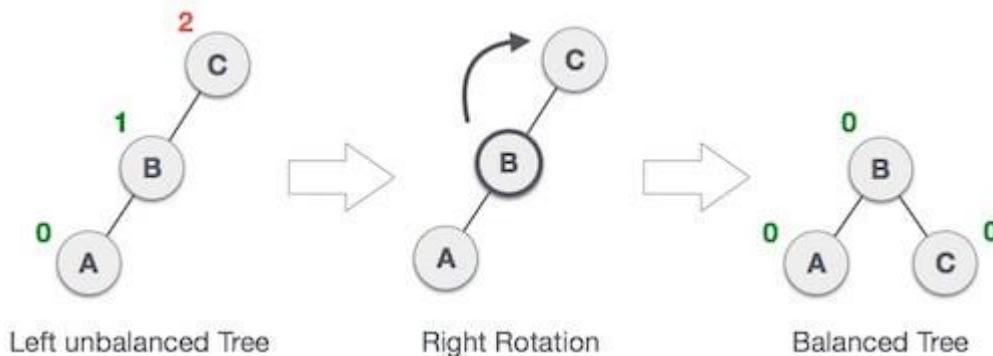
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

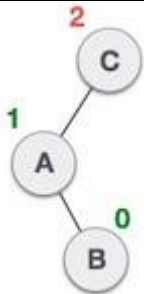
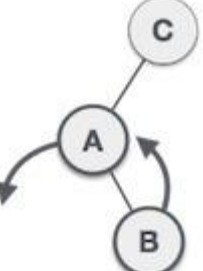
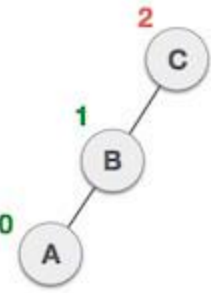
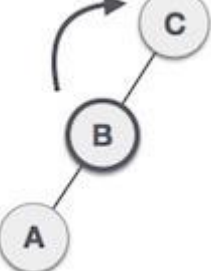
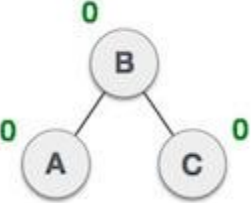


In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

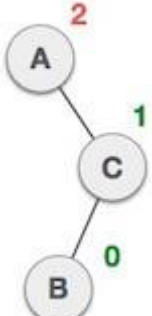
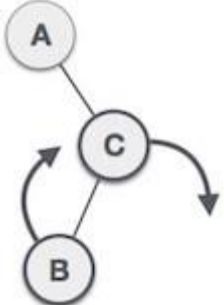
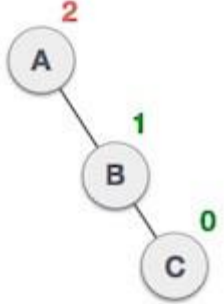
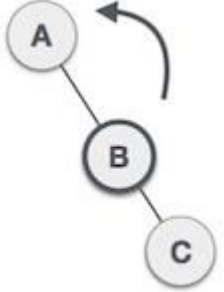
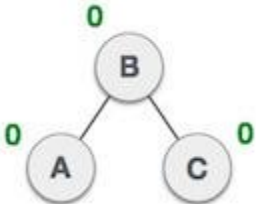
Let us understand each and every step very clearly:

State	Action
	<p>A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C</p>
	<p>As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.</p>
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C</p>
	<p>Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.</p>

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

Unit 4: Tree

State	Action
	<p>A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A</p>
	<p>As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.</p>
	<p>After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.</p>
	<p>Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.</p>

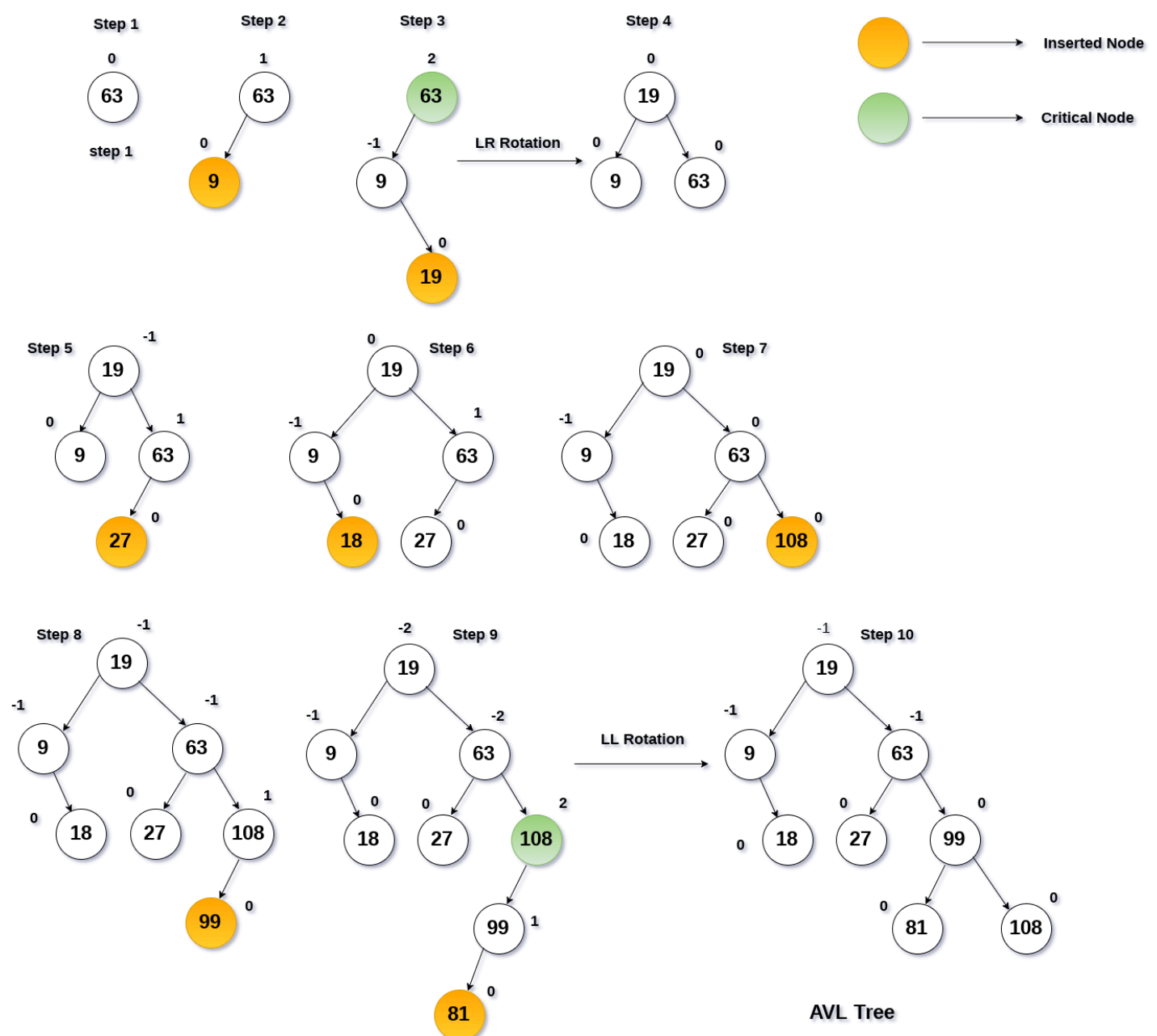
Construction of AVL Tree:

63, 9, 19, 27, 18, 108, 99, 81

The process of constructing an AVL tree from the given set of elements is shown in the following figure.

At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node.

All the elements are inserted in order to maintain the order of binary search tree.



7. B Tree: Definition, Need and Application

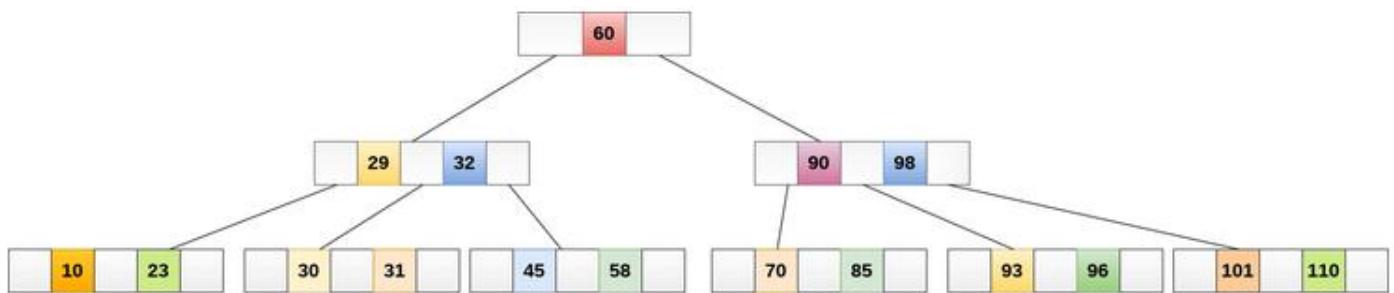
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have $m/2$ number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Needs of B-Trees

B-trees are specifically designed to address the needs of efficiently storing and retrieving large amounts of data in a disk-based storage system. The main needs and advantages of B-trees include:

1. **Effective Disk I/O:** B-trees are optimized for minimizing disk I/O operations. By storing multiple keys and associated values in each node, B-trees reduce the number of disk reads required to access or modify data. This makes them ideal for scenarios where frequent disk accesses are necessary, such as in databases and file systems.
2. **Balanced Structure:** B-trees maintain a balanced structure by ensuring that all leaf nodes are at the same level. This balance is achieved through split and merge operations, which ensure that the tree remains relatively shallow and provides efficient search, insertion, and deletion.

operations. The balanced structure helps maintain consistent performance regardless of the size of the dataset.

3. **Support for Large Datasets:** B-trees can handle large datasets efficiently. As the number of keys and associated values increases, B-trees continue to provide fast access times since the height of the tree remains relatively small due to the balancing property. This scalability makes B-trees suitable for managing and searching large databases or file systems.
4. **Sorted Data Storage:** B-trees maintain data in a sorted order within the tree structure. This enables efficient searching, range queries, and other operations that require data to be accessed in a sorted manner. B-trees leverage binary search techniques to quickly locate the desired data, resulting in improved query performance.
5. **Concurrent Access:** B-trees can be designed to support concurrent access from multiple threads or processes. Techniques such as locking or multi-version concurrency control can be implemented to ensure consistency and avoid conflicts when multiple operations are performed simultaneously. This makes B-trees well-suited for concurrent environments like databases with concurrent read and write operations.
6. **Disk Space Utilization:** B-trees efficiently utilize disk space by ensuring that each node is optimally filled with keys and values. Unlike linear structures, B-trees accommodate a variable number of keys per node and dynamically adjust the tree structure to maintain balance. This efficient utilization of disk space is crucial when dealing with large datasets, as it minimizes wasted space and optimizes storage capacity.

Overall, B-trees provide an effective and efficient solution for organizing and accessing data in disk-based storage systems. Their balanced structure, support for large datasets, optimized disk I/O, and other advantages make them a fundamental data structure in many database systems, file systems, and other applications dealing with large-scale data management.

Application of B tree

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

/*Need of a B Tree

he need for B-tree emerged as the demand for faster access to physical storage media such as a hard disk grew. With a bigger capacity, backup storage devices were slower. There was a demand for data structures that reduced the number of disc accesses.

A binary search tree, an AVL tree, a red-black tree, and other data structures can only store one key in each node. When you need to store a significant number of keys, the height of such trees grows quite vast, and the time it takes to access them grows.

B-tree, on the other hand, can store multiple keys inside a single node and has several child nodes. It considerably reduces the height, allowing for speedier disk access. */