DATA STRUCTURE

(II)

Contents

Chapter 1

Review Data Structure I

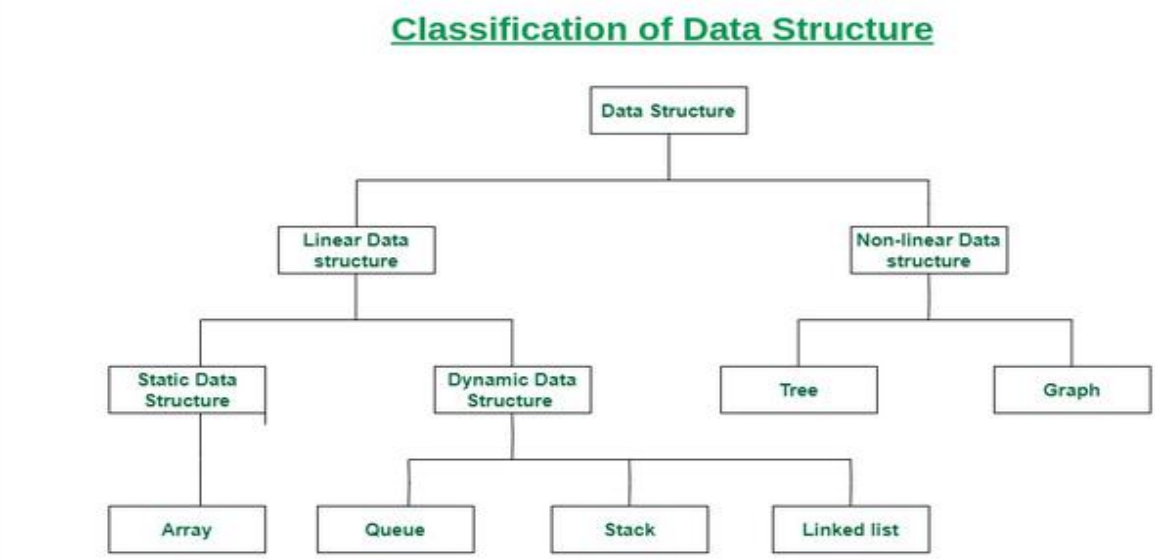
Introduction

Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. In this chapter, we will explore the most used data structures, including **arrays, linked lists, stacks, queues.**

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

Classification of Data Structure:

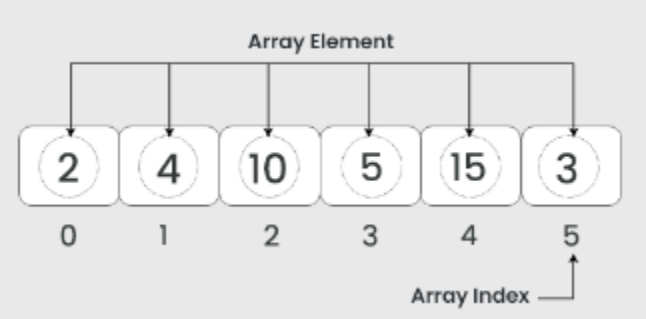


1. **Linear Data Structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
   Example: Array, Stack, Queue, Linked List, etc.

2. **Static Data Structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
   Example: array.
3. **Dynamic Data Structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
   Example: Queue, Stack, etc.
4. **Non-Linear Data Structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
   Examples: Trees and Graphs.

Array

An **array data structure** is a fundamental concept in computer science that stores a collection of elements in a contiguous block of memory. It allows for efficient access to elements using indices and is widely used in programming for organizing and manipulating data.



An **array** is a collection of items of the same variable type that are stored at contiguous memory locations. It's one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with **0** . Each element in an array is accessed through its index.

Arrays are a fundamental data structure in computer science. They are used in a wide variety of applications, including:

- Storing data for processing
- Implementing data structures such as stacks and queues
- Representing data in tables and matrices

- Creating dynamic data structures such as linked lists and trees

Types of Array

There are two main types of arrays:

- **One-dimensional arrays:** These arrays store a single row of elements.
- **Multidimensional arrays:** These arrays store multiple rows of elements.

Array Operations

- Common operations performed on arrays include:
  - **Traversal** : Visiting each element of an array in a specific order (e.g., sequential, reverse).
  - **Insertion** : Adding a new element to an array at a specific index.
  - **Deletion** : Removing an element from an array at a specific index.
  - **Searching** : Finding the index of an element in an array.

Applications of Array

- Arrays are used in a wide variety of applications, including:
  - Storing data for processing
  - Implementing data structures such as stacks and queues
  - Representing data in tables and matrices
  - Creating dynamic data structures such as linked lists and trees

Searching Algorithms

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications such as **databases, web search engines**, and more.

Searching is the fundamental process of **locating a specific element or item within a collection of data**. This collection of data can take various forms, such as arrays, lists, trees, or other structured representations. The primary objective of searching is to determine whether the desired element exists within the data, and if so, to identify its precise location or retrieve it. It plays an important role in various computational tasks and real-world applications, including information retrieval, data analysis, decision-making processes, and more.

Searching terminologies:

Target Element:

In searching, there is always a specific target element or item that you want to find within the data collection. This target could be a value, a record, a key, or any other data entity of interest.

Search Space:

The search space refers to the entire collection of data within which you are looking for the target element. Depending on the data structure used, the search space may vary in size and organization.

Complexity:

Searching can have different levels of complexity depending on the data structure and the algorithm used. The complexity is often measured in terms of time and space requirements.

Deterministic vs. Non-deterministic:

Some searching algorithms, like binary search, are deterministic, meaning they follow a clear, systematic approach. Others, such as linear search, are non-deterministic, as they may need to examine the entire search space in the worst case.

Importance of Searching in DSA:

- **Efficiency:** Efficient searching algorithms improve program performance.
- **Data Retrieval:** Quickly find and retrieve specific data from large datasets.
- **Database Systems:** Enables fast querying of databases.
- **Problem Solving:** Used in a wide range of problem-solving tasks.

Applications of Searching:

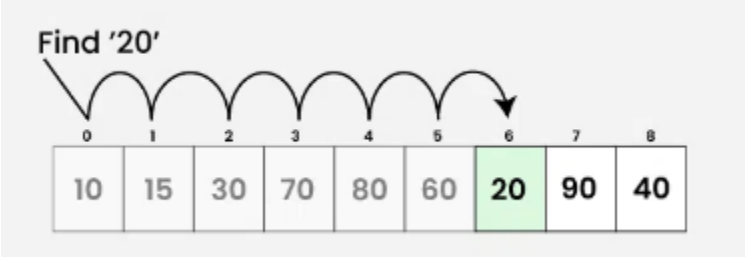
Searching algorithms have numerous applications across various fields. Here are some common applications:

- **Information Retrieval:** Search engines like Google, Bing, and Yahoo use sophisticated searching algorithms to retrieve relevant information from vast amounts of data on the web.

- **Database Systems:** Searching is fundamental in database systems for retrieving specific data records based on user queries, improving efficiency in data retrieval.
- **E-commerce:** Searching is crucial in e-commerce platforms for users to find products quickly based on their preferences, specifications, or keywords.
- **Networking:** In networking, searching algorithms are used for routing packets efficiently through networks, finding optimal paths, and managing network resources.
- **Artificial Intelligence:** Searching algorithms play a vital role in AI applications, such as problem-solving, game playing (e.g., chess), and decision-making processes
- **Pattern Recognition:** Searching algorithms are used in pattern matching tasks, such as image recognition, speech recognition, and handwriting recognition.

Linear Search Algorithm

The linear search algorithm is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found; otherwise, the search continues till the end of the dataset. In this chapter , we will learn about the basics of the linear search algorithm, its applications, advantages, disadvantages, and more to provide a deep understanding of linear search.



Linear search is a method for searching for an element in a collection of elements. In linear search, each element of the collection is visited one by one in a sequential fashion to find the desired element. Linear search is also known as **sequential search**.

Algorithm for Linear Search Algorithm:

The algorithm for linear search can be broken down into the following steps:

- **Start:** Begin at the first element of the collection of elements.
- **Compare:** Compare the current element with the desired element.
- **Found:** If the current element is equal to the desired element, return true or index to the current element.
- **Move:** Otherwise, move to the next element in the collection.
- **Repeat:** Repeat steps 2-4 until we have reached the end of collection.
- **Not found:** If the end of the collection is reached without finding the desired element, return that the desired element is not in the array.
- **How Does Linear Search Algorithm Work?**

In Linear Search Algorithm,

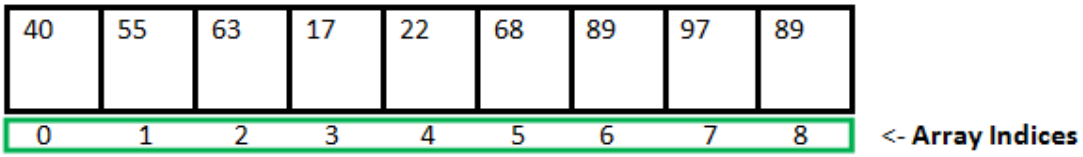
- Every element is considered as a potential match for the key and checked for the same.
- If any element is found equal to the key, the search is successful and the index of that element is returned.
- If no element is found equal to the key, the search yields “No match found”.

Static Data Structure vs Dynamic Data Structure

Data Structure is a way of storing and organizing data efficiently such that the required operations on them can be performed be efficient with respect to time as well as memory. Simply, Data Structure are used to reduce complexity (mostly the time complexity) of the code. Data structures can be two types : 1. Static Data Structure 2. Dynamic Data Structure

What is a Static Data structure?

In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.

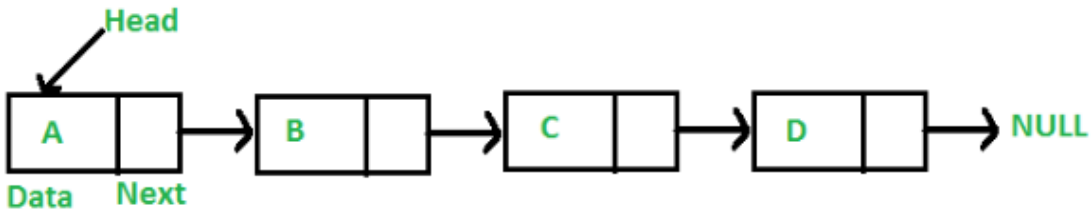


Array Length = 9
First Index = 0
Last Index = 8

Example of Static Data Structures: Array

Dynamic Data Structure:-

In Dynamic data structure the size of the structure is not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.



Example of Dynamic Data Structures: Linked List

Static Data Structure vs Dynamic Data Structure

- **Static data structures, such as arrays, have a fixed size and are allocated at compile-time. This means that their memory size cannot be changed during program execution. Index-based access to elements is fast and efficient since the address of the element is known.**
- **Dynamic data structures, on the other hand, have a variable size and are allocated at run-time. This means that their memory size can be changed during program execution. Memory can be dynamically allocated or deallocated during program execution. Due to this dynamic nature, accessing elements based on index may be slower as it may require memory allocation and deallocation**.

Aspect	Static Data Structure	Dynamic Data Structure
Memory allocation	Memory is allocated at compile-time	Memory is allocated at run-time
Size	Size is fixed and cannot be modified	Size can be modified during runtime
Memory utilization	Memory utilization may be inefficient	Memory utilization is efficient as memory can be reused
Access	Access time is faster as it is fixed	Access time may be slower due to indexing and pointer usage
Examples	Arrays, Stacks, Queues, Trees (with fixed size)	Lists, Trees (with variable size), Hash tables

Advantage of Static data structure :

- **Fast access time: Static data structures offer fast access time because memory is allocated at compile-time and the size is fixed, which makes accessing elements a simple indexing operation.**
- **Predictable memory usage:** Since the memory allocation is fixed at compile-time, the programmer can precisely predict how much memory will be used by the program, which is an important factor in memory-constrained environments.
- **Ease of implementation and optimization:** Static data structures may be easier to implement and optimize since the structure and size are fixed, and algorithms can be optimized for the specific data structure, which reduces cache misses and can increase the overall performance of the program.
- **Efficient memory management:** Static data structures allow for efficient memory allocation and management. Since the size of the data structure is fixed at compile-time, memory can be allocated and released efficiently, without the need for frequent reallocations or memory copies.

- **Simplified code:** Since static data structures have a fixed size, they can simplify code by removing the need for dynamic memory allocation and associated error checking.
- **Reduced overhead:** Static data structures typically have lower overhead than dynamic data structures, since they do not require extra bookkeeping to manage memory allocation and deallocation.

Advantage Of Dynamic Data Structure :

- **Flexibility:** Dynamic data structures can grow or shrink at runtime as needed, allowing them to adapt to changing data requirements. This flexibility makes them well-suited for situations where the size of the data is not known in advance or is likely to change over time.
- **Reduced memory waste:** Since dynamic data structures can resize themselves, they can help reduce memory waste. For example, if a dynamic array needs to grow, it can allocate additional memory on the heap rather than reserving a large fixed amount of memory that might not be used.
- **Improved performance for some operations:** Dynamic data structures can be more efficient than static data structures for certain operations. For example, inserting or deleting elements in the middle of a dynamic list can be faster than with a static array, since the remaining elements can be shifted over more efficiently.
- **Simplified code**: Dynamic data structures can simplify code by removing the need for manual memory management. Dynamic data structures can also reduce the complexity of code for data structures that need to be resized frequently.
- **Scalability: Dynamic data structures can be more scalable than static data structures, as they can adapt to changing data requirements as the data grows.**

Linked List

A linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque.

A linked list is a linear data structure that consists of a series of nodes connected by pointers (in C or C++) or references (in Java, Python and

JavaScript). Each node contains data and a pointer/reference to the next node in the list. Unlike arrays, linked lists allow for efficient insertion or removal of elements from any position in the list, as the nodes are not stored contiguously in memory.

Linked List vs. Array

Here's the comparison of Linked List vs Arrays

Linked List:

- **Data Structure: Non-contiguous**
- **Memory Allocation: Typically allocated one by one to individual elements**
- **Insertion/Deletion: Efficient**
- **Access: Sequential**

Array:

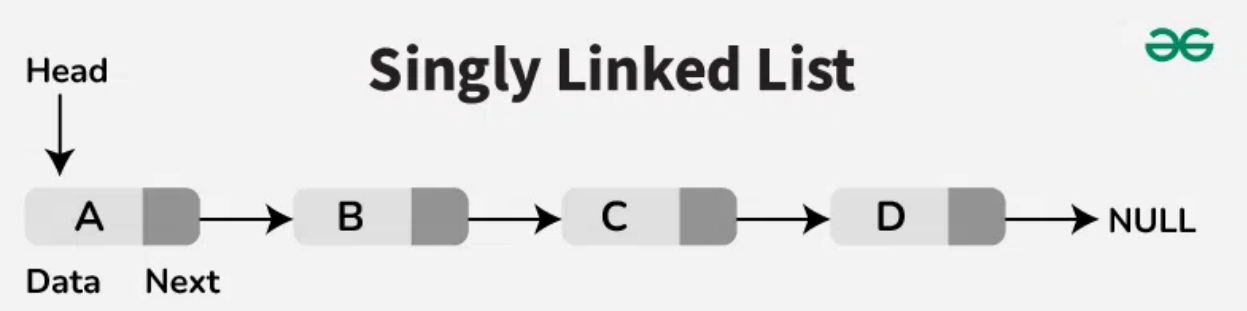
- **Data Structure: Contiguous**
- **Memory Allocation: Typically allocated to the whole array**
- **Insertion/Deletion: Inefficient**
- **Access: Random**

Types of Linked List :

(1) Single Linked List
(2) Doubly Linked List
(3) Circular Linked List
(4) Circular Double Linked List
(5) Header Linked List

Singly linked list :-

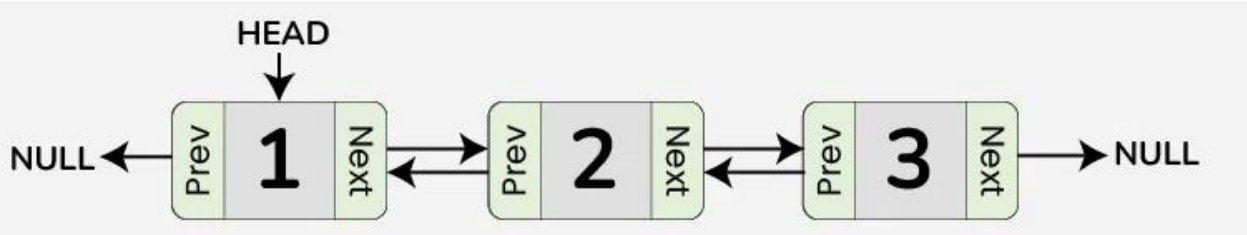
Singly linked list is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.



Double Linked List

A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.

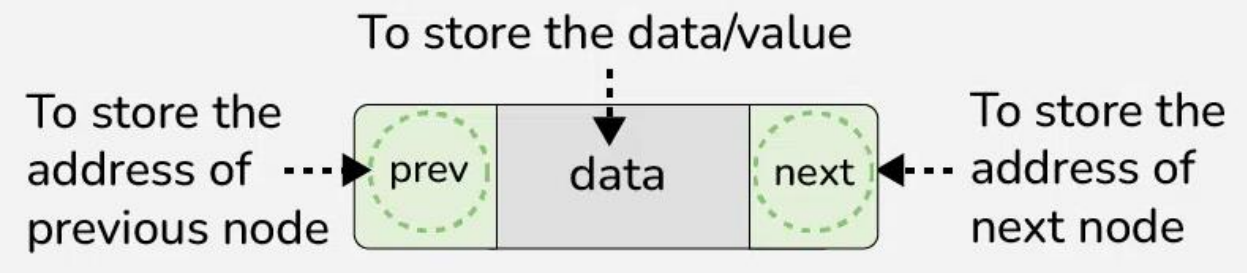
A doubly linked list is a data structure that consists of a set of nodes, each of which contains a value and two pointers, one pointing to the previous node in the list and one pointing to the next node in the list. This allows for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.



Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (next)
3. A pointer to the previous node (prev)



A circular linked list :-

A circular linked list is a data structure where the last node connects back to the first, forming a loop. This structure allows for continuous traversal without any interruptions. Circular linked lists are especially helpful for tasks like scheduling and managing playlists, this allowing for smooth navigation. In this chapter, we'll cover the basics of circular linked lists, how to work with them, their advantages and disadvantages, and their applications.

A circular linked list is a special type of linked list where all the nodes are connected to form a circle. Unlike a regular linked list, which ends with a node pointing to NULL, the last node in a circular linked list points back to the first node. This means that you can keep traversing the list without ever reaching a NULL value.

Types of Circular Linked Lists

We can create a circular linked list from both singly linked lists and doubly linked lists.

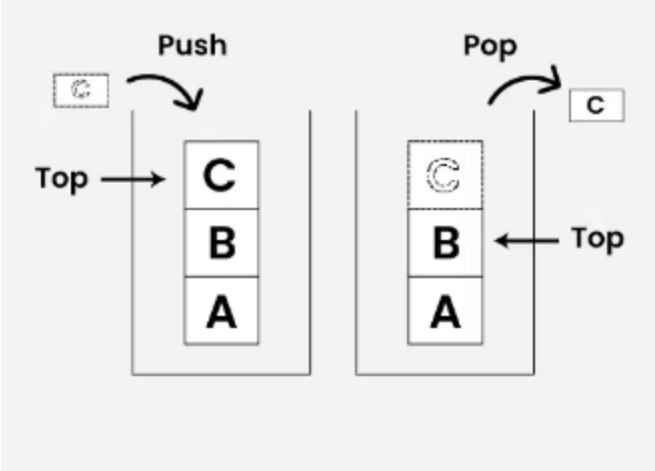
Circular Doubly Linked List:-

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.

Stack

Stack Data Structure is a linear data structure that follows LIFO (Last In First Out) Principle , so the last element inserted is the first to be popped out. In this chapter , we will cover all the basics of Stack, Operations on Stack, its implementation,

advantages, disadvantages which will help you solve all the problems based on Stack.



Stack is a linear data structure based on LIFO(Last In First Out) principle in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.

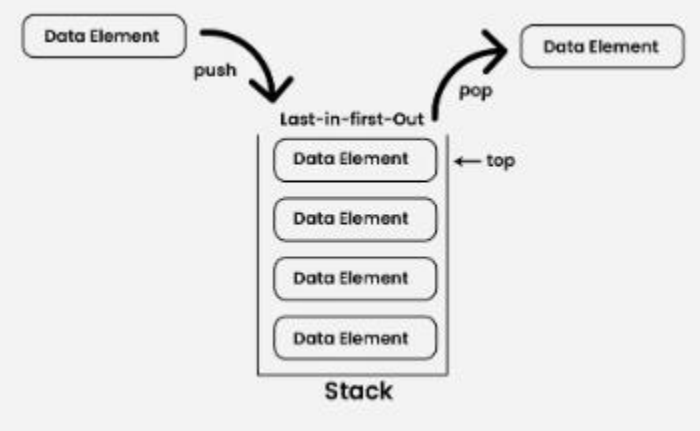
To implement the stack, it is required to maintain the pointer to the top of the stack , which is the last element to be inserted because we can access the elements only on the top of the stack.

LIFO(Last In First Out) Principle in Stack Data Structure:

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

Representation of Stack Data Structure:

Stack follows LIFO (Last In First Out) Principle so the element which is pushed last is popped first.



Types of Stack Data Structure:

- Fixed Size Stack : As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
- Dynamic Size Stack : A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Basic Operations on Stack Data Structure:

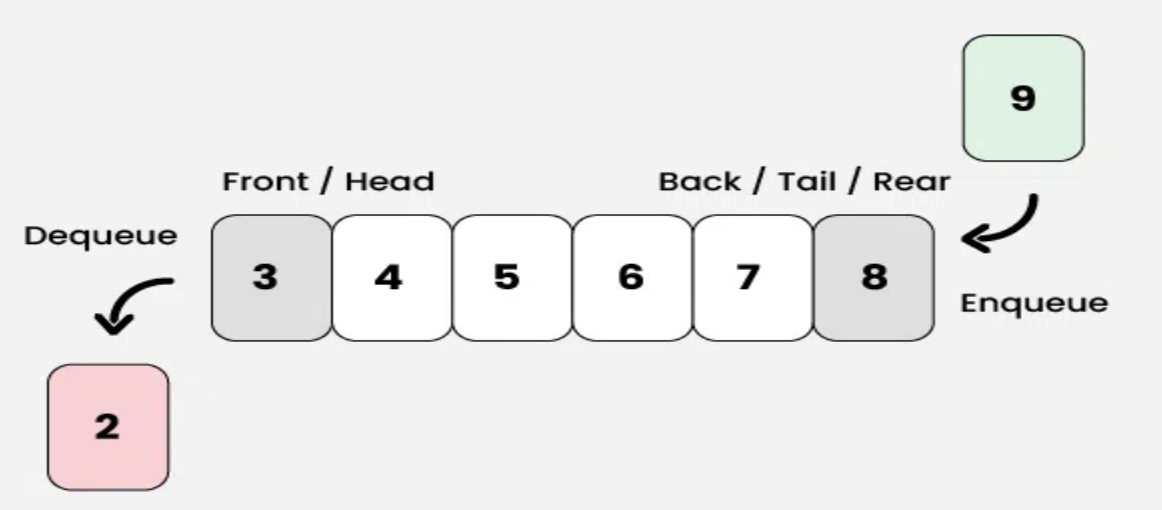
In order to make manipulations in a stack, there are certain operations provided to us.

- push() to insert an element into the stack
- pop() to remove an element from the stack
- top() Returns the top element of the stack.
- isEmpty() returns true if stack is empty else false.
- isFull() returns true if the stack is full else false.

Queue

Queue Data Structure is a linear data structure that follows FIFO (First In First Out) Principle, so the first element inserted is the first to be popped out. We will

cover all the basics of Queue, Operations on Queue, its implementation, advantages, disadvantages which will help you solve all the problems based on Queue.

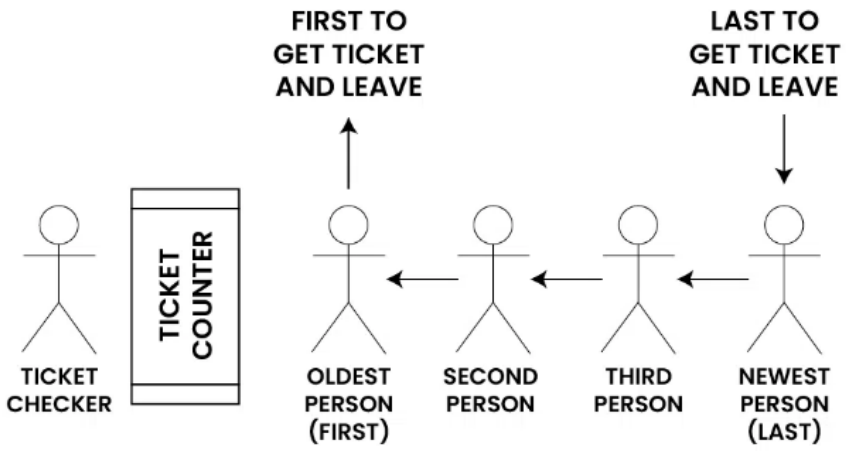


Queue Data Structure is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end (back of the queue), and all deletions from the list are made at the other end(front of the queue). The element which is first pushed into the order, the delete operation is first performed on that.

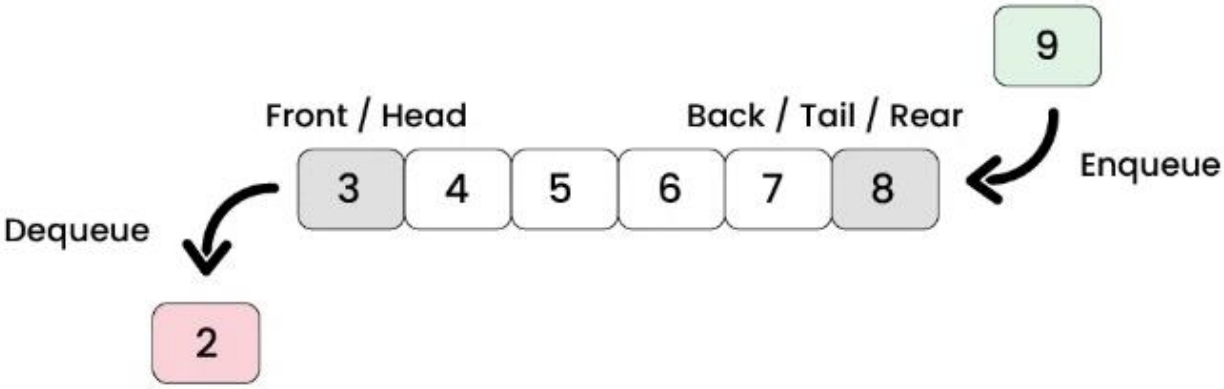
FIFO Principle of Queue Data Structure:-

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue(sometimes, head of the queue). Similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue.



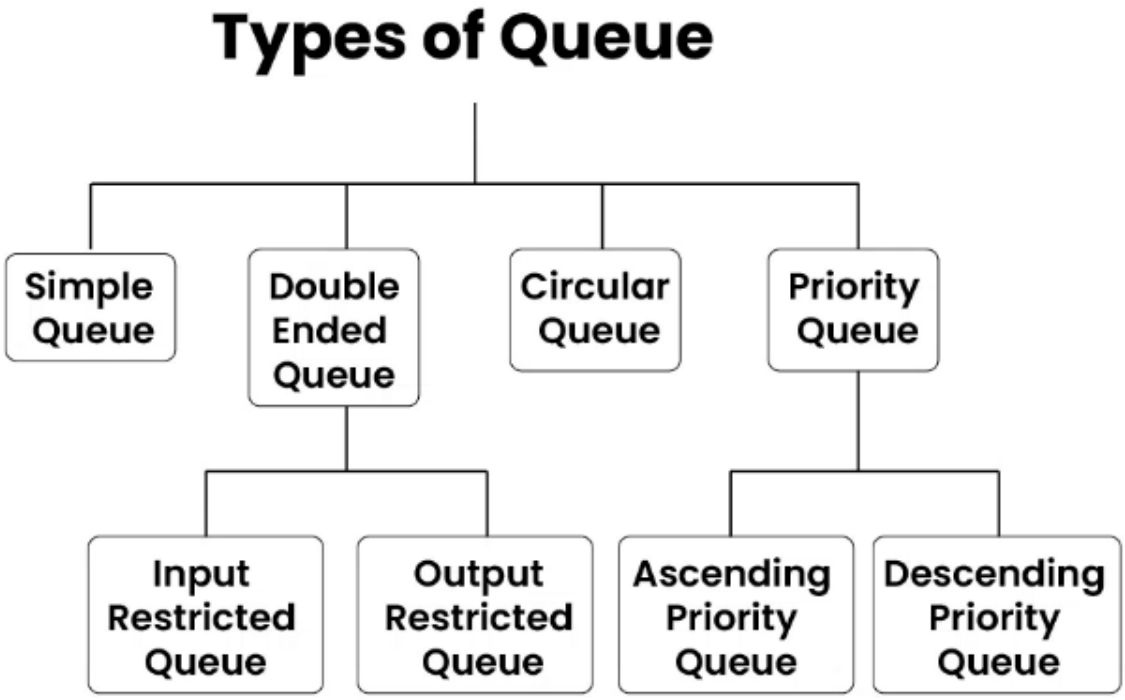
Representation of Queue Data Structure:

The image below shows how we represent Queue Data Structure:



Types of Queue Data Structure

Queue data structure can be classified into 4 types:



There are different types of queues:

1. **Simple Queue:** Simple Queue simply follows FIFO Structure. We can only insert the element at the back and remove the element from the front of the queue.
2. **Double-Ended Queue (Dequeue):** In a double-ended queue the insertion and deletion operations, both can be performed from both ends. They are of two types:
    - Input Restricted Queue: This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
    - Output Restricted Queue: This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
3. **Circular Queue:** This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order.
4. **Priority Queue:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them. They are of two types:

- Ascending Priority Queue**:** In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first**.**
- Descending Priority Queue**:** In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority is popped first.

Basic Operations of Queue Data Structure :

Some of the basic operations for Queue in Data Structure are:

1. Enqueue: Adds (or stores) an element to the end of the queue..
2. Dequeue: Removal of elements from the queue.
3. Peek or front: Acquires the data element available at the front node of the queue without deleting it.
4. rear: This operation returns the element at the rear end without removing it.
5. isFull: Validates if the queue is full.
6. isEmpty: Checks if the queue is empty.

Chapter 2

Tree

Introduction

In this chapter, we look at a simple data structure for which the average running time of most operations is O(logN). We will discuss their use in other, more general applications. In this chapter, we will :-

- See how trees are used to implement the file system of several popular operating systems.
- See how trees can be used to evaluate arithmetic expressions.
- Show how to use trees to support searching operations in O(logN) average time and how to refine these ideas to obtain O(logN) worst-case bounds.
- We will also see how to implement these operations when the data are stored on a disk. Discuss and use the set and map classes.

Definition of Tree:

- A tree can be defined in several ways. One natural way to define a tree is recursively.
- A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node, r, called the root, and zero or more nonempty (sub)trees T_1, T_2, ... , T_k, each of whose roots are connected by a directed edge from r.
- The root of each subtree is said to be a child of r, and r is the parent of each subtree root. Figure 1.1 shows a typical tree using the recursive definition.

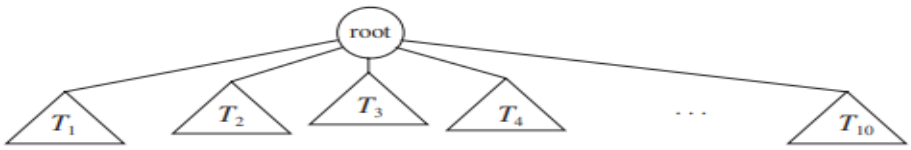
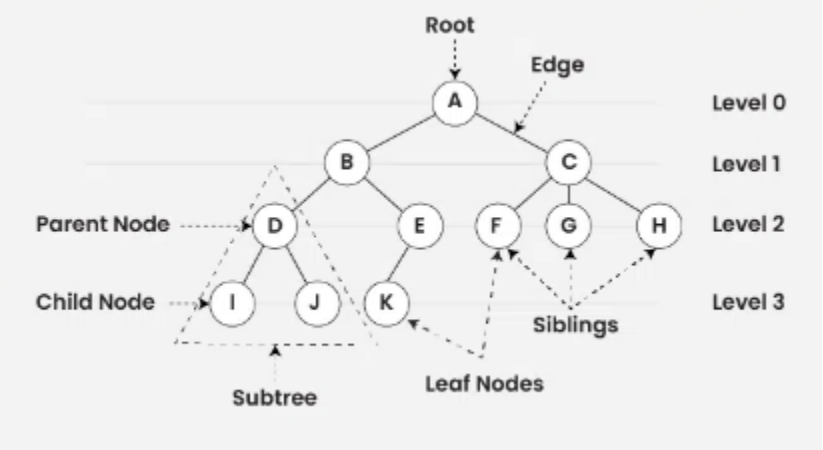


Figure 1.1 Generic Tree

Tree data structure is a specialized data structure to store data in hierarchical manner. It is used to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected.



Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the **root**, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

The data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**

- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

Example :

- From the recursive definition, we find that a tree is a collection of N nodes, one of which is the root, and N − 1 edges. That there are N − 1 edges follows from the fact that each edge connects some node to its parent, and every node except the root has one parent (see Fig. 1.2).

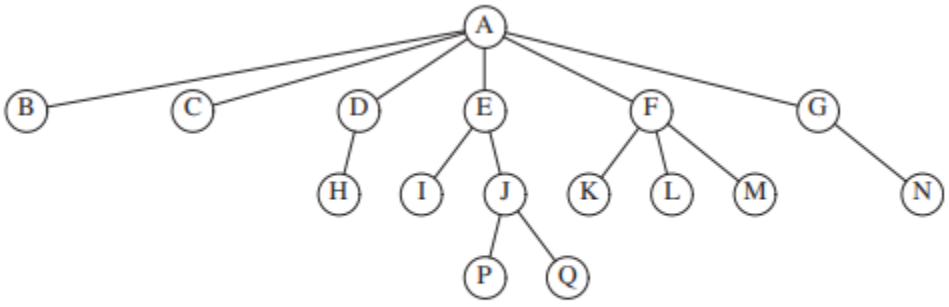


Figure 1.2 A tree

- In the tree of Figure 1.2, the root is A. Node F has A as a parent and K, L, and M as children.
- Each node may have an arbitrary number of children, possibly zero. Nodes with no children are known as **leaves**;
- The leaves in the tree above are B, C, H, I, P, Q, K, L, M, and N.
- Nodes with the same parent are **siblings**; thus, K, L, and M are all siblings.
- Grandparent and grandchild relations can be defined in a similar manner.

Representation of Tree Data Structure:

A tree consists of a root node, and zero or more subtrees T1, T2, … , Tk such that there is an edge from the root node of the tree to the root node of each subtree. Subtree of a node X consists of all the nodes which have node X as the ancestor node.

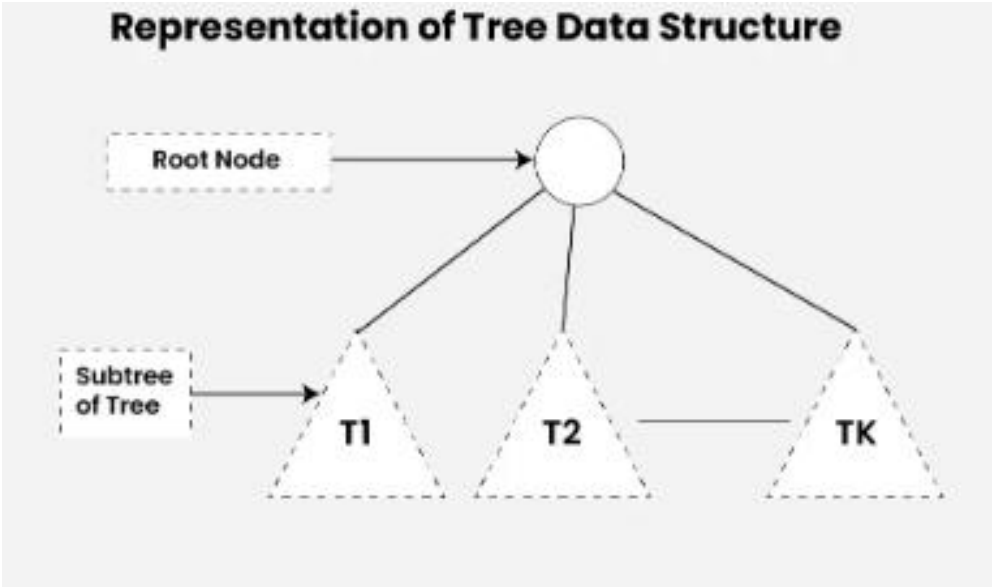


Figure 2.1 Representation Tree Data Structure

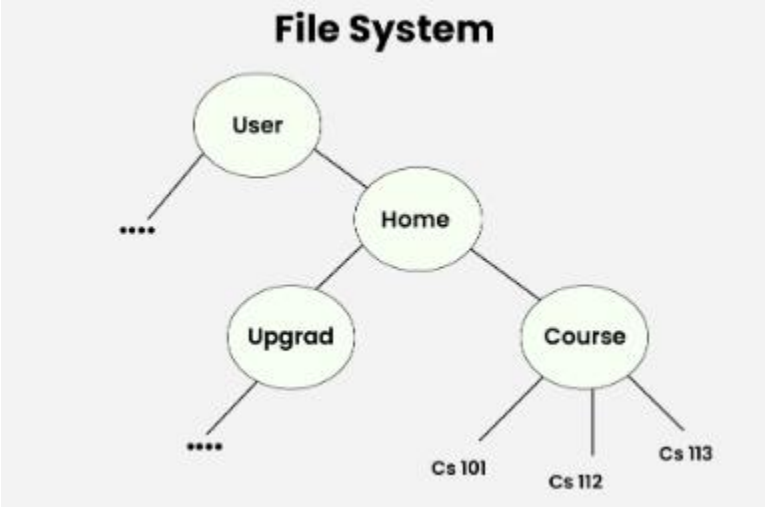
Representation of a Node in Tree Data Structure:

A tree can be represented using a collection of nodes. Each of the nodes can be represented with the help of class or structs. Below is the representation of Node in C++ language:

```
class Node {
public:
        int data;
        Node* first_child;
        Node* second_child;
        Node* third_child;
        .
        .
        .
        Node* nth_child;
};
```

Importance for Tree Data Structure:

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

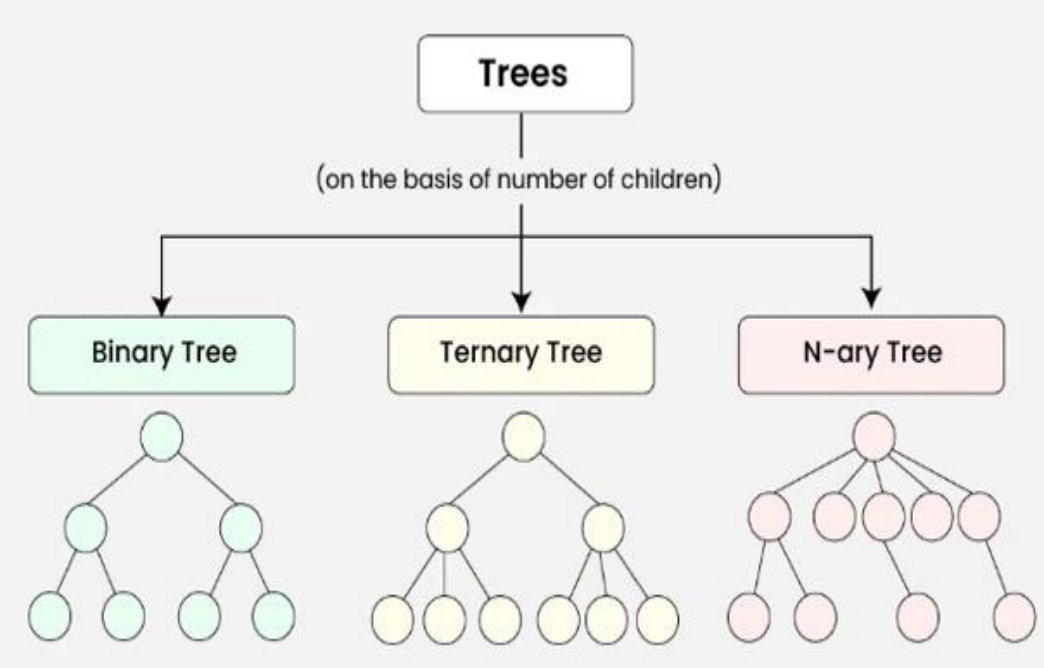


2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).

3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).

4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Types of Tree Data Structures:



Tree data structure can be classified into three types based upon the number of children each node of the tree can have. The types are:

- **Binary tree:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
- **Ternary Tree:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

- **Basic Operations Of Tree Data Structure:**
- **Create** – create a tree in the data structure.
- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Traversal**:
  - ✓ **Depth-First-Search Traversal**
  - ✓ **Breadth-First-Search Traversal**

Implementation of Tree Data Structure:

```
// C++ program to demonstrate some of the above
// terminologies
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to add an edge between vertices x and y
void addEdge(int x, int y, vector<vector<int> >& adj)
{
   adj[x].push_back(y);
   adj[y].push_back(x);
}

// Function to print the parent of each node
void printParents(int node, vector<vector<int> >& adj,
                  int parent)
{
   // current node is Root, thus, has no parent
   if (parent == 0)
      cout << node << "->Root" << endl;
   else
      cout << node << "->" << parent << endl;

   // Using DFS
   for (auto cur : adj[node])
      if (cur != parent)
         printParents(cur, adj, node);
}

// Function to print the children of each node
void printChildren(int Root, vector<vector<int> >& adj)
{
   // Queue for the BFS
   queue<int> q;

   // pushing the root
   q.push(Root);

   // visit array to keep track of nodes that have been
   // visited
   int vis[adj.size()] = { 0 };

   // BFS
   while (!q.empty()) {
      int node = q.front();
      q.pop();
      vis[node] = 1;
      cout << node << "-> ";
      for (auto cur : adj[node])
         if (vis[cur] == 0) {
            cout << cur << " ";
            q.push(cur);
```

```
        }
      cout << endl;
   }
}

// Function to print the leaf nodes
void printLeafNodes(int Root, vector<vector<int> >& adj)
{
   // Leaf nodes have only one edge and are not the root
   for (int i = 1; i < adj.size(); i++)
      if (adj[i].size() == 1 && i != Root)
         cout << i << " ";
   cout << endl;
}

// Function to print the degrees of each node
void printDegrees(int Root, vector<vector<int> >& adj)
{
   for (int i = 1; i < adj.size(); i++) {
      cout << i << ": ";

      // Root has no parent, thus, its degree is equal to
      // the edges it is connected to
      if (i == Root)
         cout << adj[i].size() << endl;
      else
         cout << adj[i].size() - 1 << endl;
   }
}
// Driver code
int main()
{
   // Number of nodes
   int N = 7, Root = 1;
   // Adjacency list to store the tree
   vector<vector<int> > adj(N + 1, vector<int>());
   // Creating the tree
   addEdge(1, 2, adj);
   addEdge(1, 3, adj);
   addEdge(1, 4, adj);
   addEdge(2, 5, adj);
   addEdge(2, 6, adj);
   addEdge(4, 7, adj);

   // Printing the parents of each node
   cout << "The parents of each node are:" << endl;
```

```
    printParents(Root, adj, 0);

    // Printing the children of each node
    cout << "The children of each node are:" << endl;
    printChildren(Root, adj);

    // Printing the leaf nodes in the tree
    cout << "The leaf nodes of the tree are:" << endl;
    printLeafNodes(Root, adj);

    // Printing the degrees of each node
    cout << "The degrees of each node are:" << endl;
    printDegrees(Root, adj);

    return 0;
}
```

Properties of Tree Data Structure:

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in

Some Terminology

- ✓ **A path: A sequence of edges**
  from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, ... , n_k such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The length of this path is the number of edges on the path, namely, $k - 1$. There is a path of length zero from every node to itself. Notice that in a tree there is exactly one path from the root to each node**.**

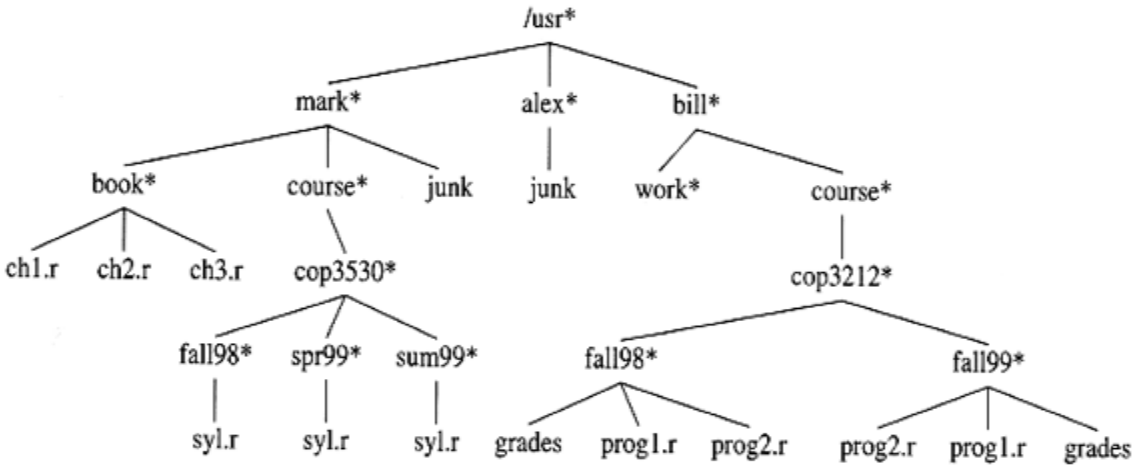
- ✓ ***Length of a path:*** **number of edges on the path**
- ✓ **A Depth : length of the unique path from the root to that node**

  For any node n_i, the depth of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0. The height of n_i is the length of the longest path from n_i to a leaf. Thus all leaves are at height 0. The height of a tree is equal to the height of the root. For the tree in Figure 1.2, E is at depth 1 and height 2; F is at depth 1 and height 1; the height of the tree is 3. The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree. If there is a path from n1 to n2, then n1 is an ancestor of n2 and n2 is a descendant of n1. If n1 = n2, then n1 is a proper ancestor of n2 and n2 is a proper descendant of n1.
- ✓ ***Height*** **of a node:** length of the longest path from that node to a leaf all leaves are at height 0
- ✓ The height of a tree = the height of the root
  = the depth of the deepest leaf
- ✓ ***Ancestor*** **and** ***descendant***
  - If there is a path from n1 to n2
  - n1 is an ancestor of n2, n2 is a descendant of n1
  - *Proper ancestor* and *proper descendant*

Applications of Tree Data Structure:-

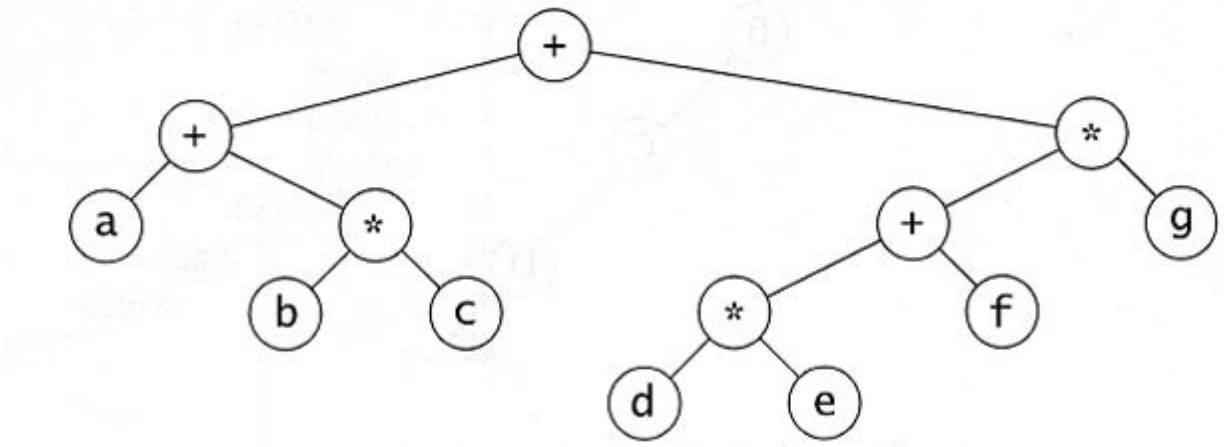
- **File System:** This allows for efficient navigation and organization of files.

  Example:-Unix Directory



- **Data Compression**: Huffman coding is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
- **Compiler Design:** In compiler design, a syntax tree is used to represent the structure of a program.
- **Database Indexing**: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.
- **Expression Tree:**

Expression tree for (a + b * c) + ((d * e + f) * g)



- ✓ Leaves are operands (constants or variables)
- ✓ The internal nodes contain operators
- ✓ Will not be a binary tree if some operators are not binary

Advantages of Tree Data Structure:

- Tree offer **Efficient Searching** Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
- Trees provide a hierarchical representation of data, making it **easy to organize and navigate** large amounts of information.
- The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

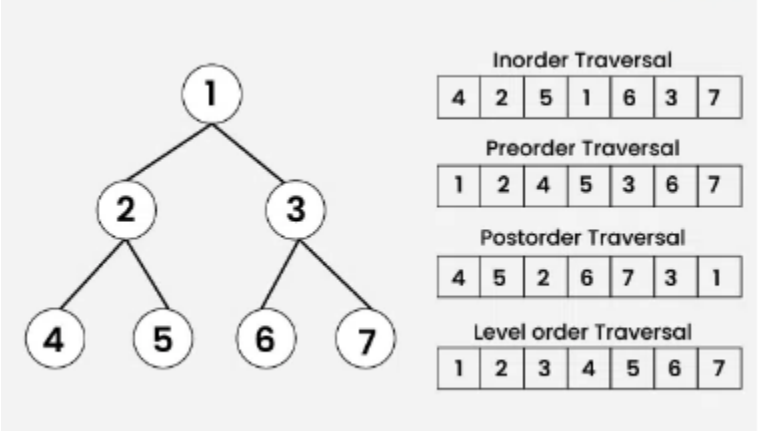
Disadvantages of Tree Data Structure:-

- Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to inefficient search times.
- Trees demand more memory space requirements than some other data structures like arrays and linked lists, especially if the tree is very large.
- The implementation and manipulation of trees can be complex and require a good understanding of the algorithms.

Tree Traversal techniques

Tree Traversal :**Used to print out the data in a tree in a certain order**

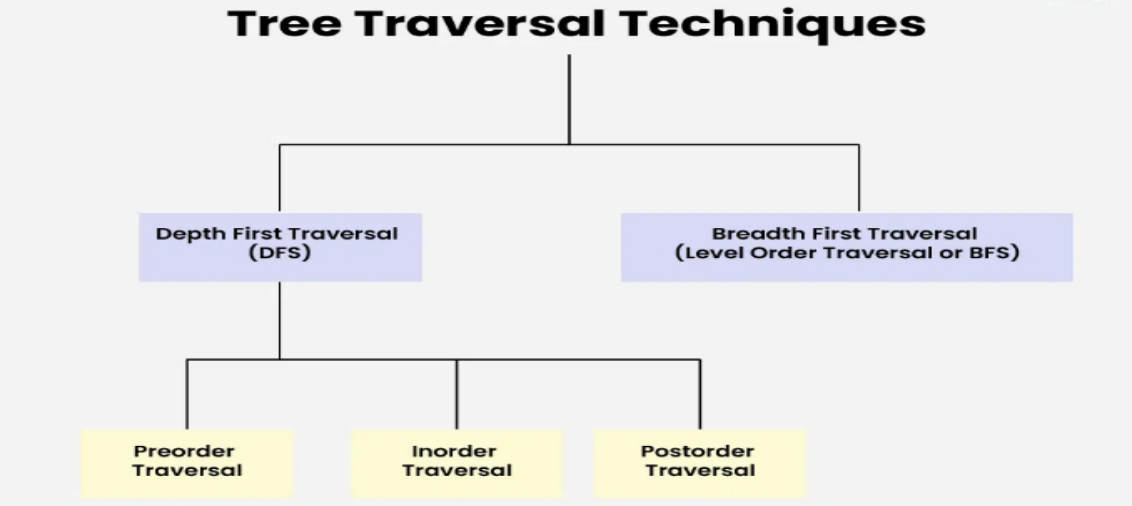
Tree Traversal techniques include various ways to visit all the nodes of the tree. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. In this chapter , we will discuss about all the tree traversal techniques along with their uses.



Tree Traversal Meaning:

Tree Traversal refers to the process of visiting or accessing each node of the tree exactly once in a certain order. Tree traversal algorithms help us to visit and process all the nodes of the tree. Since tree is not a linear data structure, there are multiple nodes which we can visit after visiting a certain node. There are multiple tree traversal techniques which decide the order in which the nodes of the tree are to be visited.

Tree Traversal Techniques:

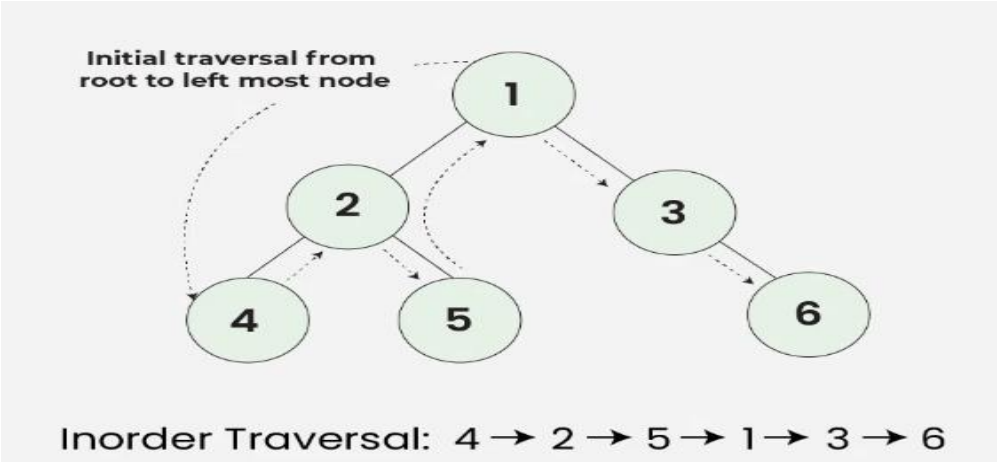


A Tree Data Structure can be traversed in following ways:

- Depth First Search or DFS
  - Inorder Traversal
  - Preorder Traversal
  - Postorder Traversal
- Level Order Traversal or Breadth First Search or BFS

Inorder traversal:-

Inorder traversal visits the node in the order: **Left -> Root -> Right**



Algorithm for Inorder Traversal:

- ***Traverse the left subtree, i.e., call Inorder(left->subtree)***
- ***Visit the root.***
- ***Traverse the right subtree, i.e., call Inorder(right->subtree)***

Pseudocode for Inorder

```
Algorithm Inorder(x)
Input: x is the root of a subtree.
1.   if x ≠ NULL
2.      then Inorder(left(x));
3.           output key(x);
4.           Inorder(right(x));
```

Uses of Inorder Traversal:

- **In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.**
- To **get nodes** of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.
- Inorder traversal can be used to evaluate arithmetic expressions stored in expression trees.

The implementation of **Inorder Traversal using C++:-**

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

struct Node {
   int data;
   Node* left;
   Node* right;
   Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

// Function to perform inorder traversal
void inorderTraversal(Node* root) {

   // Empty Tree
   if (root == nullptr)
      return;
```

```
        // Recur on the left subtree
        inorderTraversal(root->left);

        // Visit the current node
        cout << root->data << " ";

        // Recur on the right subtree
        inorderTraversal(root->right);
    }

    int main() {
        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);
        inorderTraversal(root);
        return 0;
}
```

Output

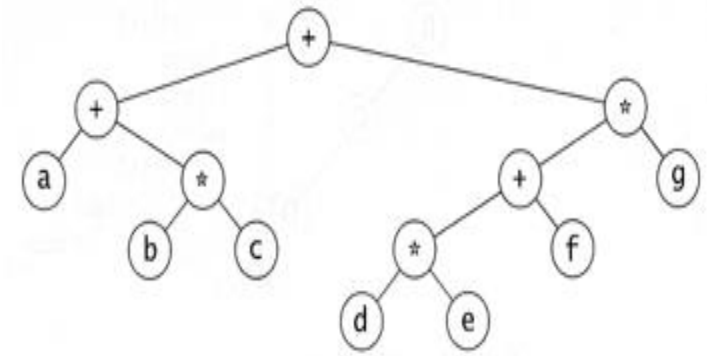
```
4 2 5 1 3
```

Time Complexity: O(N)

Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

Example :

- Inorder traversal
  - left, node, right
  - infix expression
    - a+b*c+d*e+f*g

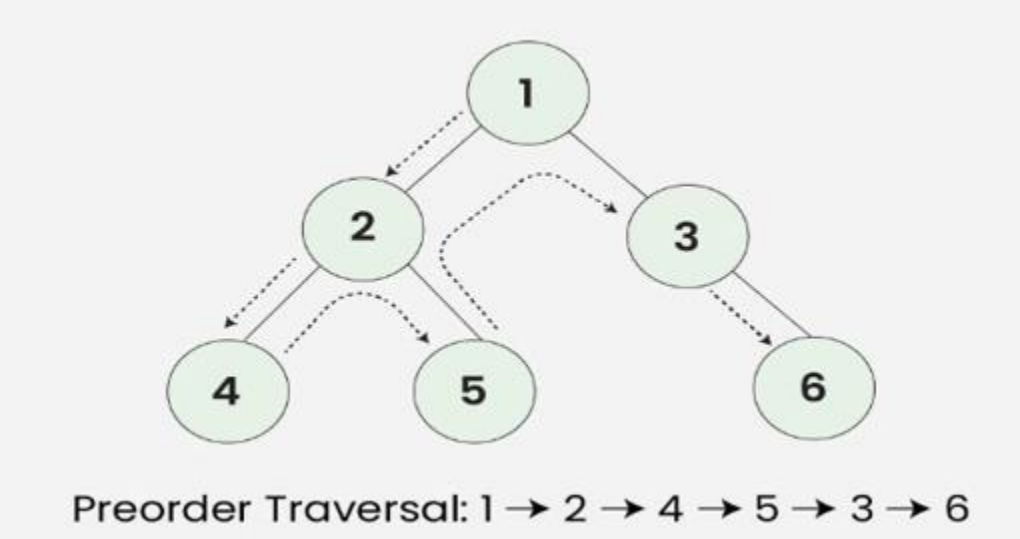


Expression tree for (a + b * c) + ((d * e + f) * g)

Preorder Traversal:-

- Print the data at the root
- Recursively print out all data in the leftmost subtree
- …
- Recursively print out all data in the rightmost subtree

Preorder traversal visits the node in the order: **Root -> Left -> Right**



Algorithm for Preorder Traversal:

Preorder(tree)

- **Visit the root.**
- **Traverse the left subtree, i.e., call Preorder(left->subtree)**
- **Traverse the right subtree, i.e., call Preorder(right->subtree)**

Pseudo code for preorder:

```
Algorithm Preorder(x)
Input: x is the root of a subtree.
1.  if x ≠ NULL
2.     then output key(x);
3.          Preorder(left(x));
4.          Preorder(right(x));
```

Uses of Preorder Traversal:

- Preorder traversal is used to create a copy of the tree.
- Preorder traversal is also used to get prefix expressions on an expression tree.

C++ Code for Preorder Traversal:

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

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int x) {
        data = x;
        left = right = nullptr;
    }
};

// Function to perform preorder traversal
void preorderTraversal(Node* root) {
    // Base case
    if (root == nullptr)
        return;

    // Visit the current node
    cout << root->data << " ";

    // Recur on the left subtree
    preorderTraversal(root->left);

    // Recur on the right subtree
    preorderTraversal(root->right);
}

int main() {
    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);
    preorderTraversal(root);
    return 0;
}
```

Output :

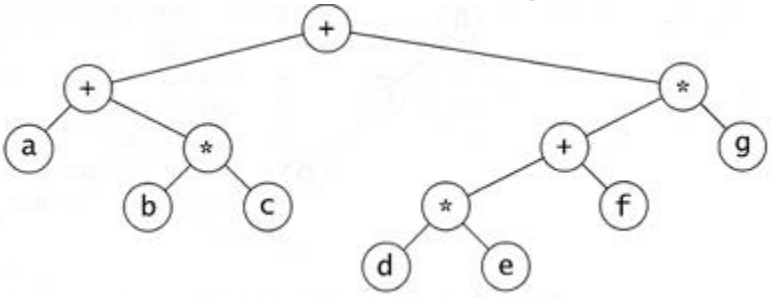
```
1 2 4 5 3
```

Time Complexity: O(N)
Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.
Example :

- Preorder traversal
    - node, left, right

- prefix expression

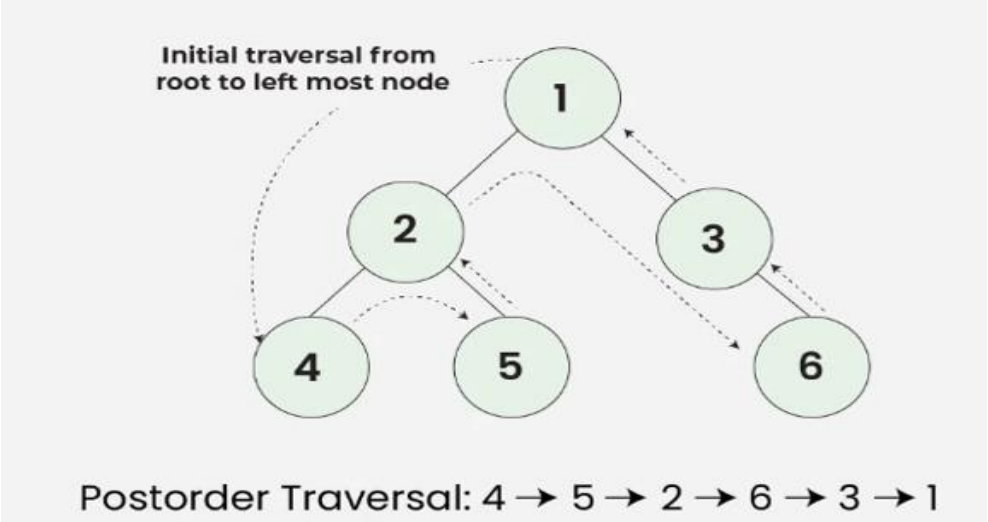
++a*bc*+*defg



Expression tree for (a + b * c) + ((d * e + f) * g)

Postorder traversal

Postorder traversal visits the node in the order: **Left -> Right -> Root**



Algorithm for Postorder Traversal:

Algorithm Postorder(tree)

- *Traverse the left subtree, i.e., call Postorder(left->subtree)*
- *Traverse the right subtree, i.e., call Postorder(right->subtree)*
- *Visit the root*

Pseudocode for PostOrder:

Algorithm *Postorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Postorder*(left(x));
3. *Postorder*(right(x));
4. output key(x);

Uses of Postorder Traversal:

- Postorder traversal is used to delete the tree.
- Postorder traversal is also useful to get the postfix expression of an expression tree.
- Postorder traversal can help in garbage collection algorithms, particularly in systems where manual memory management is used.

C++ Code for Postorder Traversal:

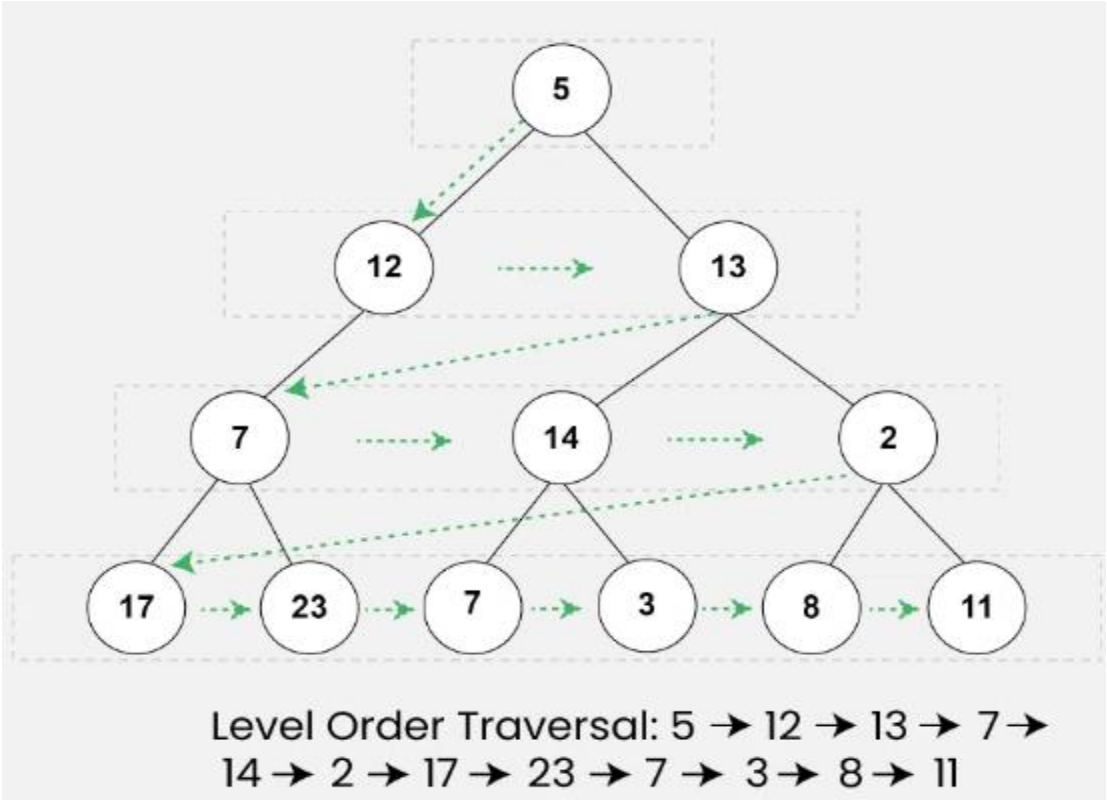
```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int x) {
        data = x;
        left = right = nullptr;
    }
};

// Function to perform postorder traversal
void postorderTraversal(Node* node) {

    // Base case
    if (node == nullptr)
        return;

    // Recur on the left subtree
    postorderTraversal(node->left);

    // Recur on the right subtree
    postorderTraversal(node->right);

    // Visit the current node
    cout << node->data << " ";
}

int main() {
    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);
    postorderTraversal(root);
    return 0;
}
```

```
4 5 2 3 1
```

Level Order Traversal

Level Order Traversal visits all nodes present in the same level completely before visiting the next level.



Algorithm for Level Order Traversal:

LevelOrder(tree)

- *Create an empty queue Q*
- *Enqueue the root node of the tree to Q*
- *Loop while Q is not empty*
    - *Dequeue a node from Q and visit it*
    - *Enqueue the left child of the dequeued node if it exists*
    - *Enqueue the right child of the dequeued node if it exists*

Uses of Level Order:

- Level Order Traversal is mainly used as Breadth First Search to search or process nodes level-by-level.
- Level Order traversal is also used for Tree Serialization and Deserialization .

C++ Code for Level Order Traversal:

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

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int x) {
      data = x;
      left = right = nullptr;
    }
};

// Prints level order traversal
void levelOrderTraversal(Node* root) {
    if (!root) return;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* curr = q.front();
        q.pop();
        cout << curr->data << " ";
        if (curr->left) q.push(curr->left);
        if (curr->right) q.push(curr->right);
    }
}

int main() {
    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->right = new Node(6);
    levelOrderTraversal(root);
    return 0;
}
```

Output

```
1 2 3 4 5 6
```

Other Tree Traversals:

1. Boundary Traversal
2. Diagonal Traversal

Boundary Traversal of a Tree includes:

- left boundary (nodes on left excluding leaf nodes)
- leaves (consist of only the leaf nodes)
- right boundary (nodes on right excluding leaf nodes)

Algorithm for Boundary Traversal:

BoundaryTraversal(tree)

- *If root is not null:*
  - *Print root's data*
  - *PrintLeftBoundary(root->left) // Print the left boundary nodes*
  - *PrintLeafNodes(root->left) // Print the leaf nodes of left subtree*
  - *PrintLeafNodes(root->right) // Print the leaf nodes of right subtree*
  - *PrintRightBoundary(root->right) // Print the right boundary nodes*

Uses of Boundary Traversal:

- Boundary traversal helps visualize the outer structure of a binary tree, providing insights into its shape and boundaries.
- Boundary traversal provides a way to access and modify these nodes, enabling operations such as pruning or repositioning of boundary nodes.

Diagonal Traversal

In the Diagonal Traversal of a Tree, all the nodes in a single diagonal will be printed one by one.

Algorithm for Diagonal Traversal:

Algorithm for Diagonal Traversal:
If root is not null:

- *Create an empty map*
- *DiagonalTraversalUtil(root, 0, M) // Call helper function with initial diagonal level 0*
- *For each key-value pair (diagonalLevel, nodes) in M:*

- *For each node in nodes:*
    - *Print node's data*

DiagonalTraversalUtil(node, diagonalLevel, M):

- *If node is null:*
- *Return*
- *If diagonalLevel is not present in M:*
    - *Create a new list in M for diagonalLevel*
- *Append node's data to the list at M[diagonalLevel]*
- *DiagonalTraversalUtil(node->left, diagonalLevel + 1, M) // Traverse left child with increased diagonal level*
- *DiagonalTraversalUtil(node->right, diagonalLevel, M) // Traverse right child with same diagonal level*

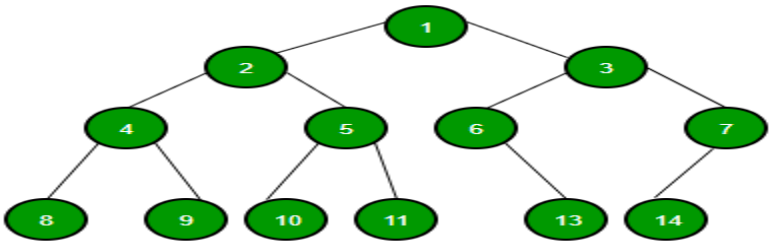
Uses of Diagonal Traversal:

- Diagonal traversal helps in visualizing the hierarchical structure of binary trees, particularly in tree-based data structures like binary search trees (BSTs) and heap trees.
- Diagonal traversal can be utilized to calculate path sums along diagonals in a binary tree.

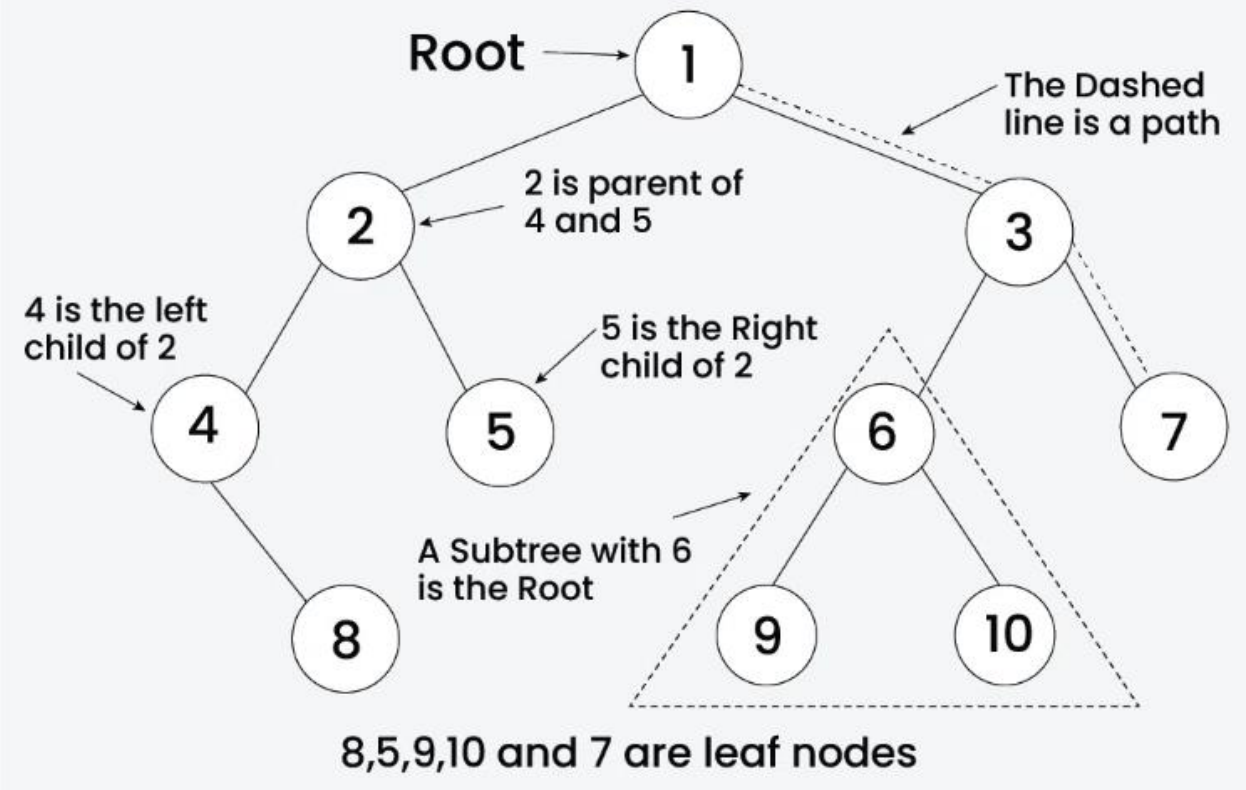
Types of Tree

Binary Tree Data Structure

A Binary Tree Data Structure is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used in computer science for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal.



Binary Tree is a non-linear data structure where each node has at most two children. In this, chapter we will cover all the basics of Binary Tree, Operations on Binary Tree, its implementation, advantages, disadvantages which will help you solve all the problems based on Binary Tree.

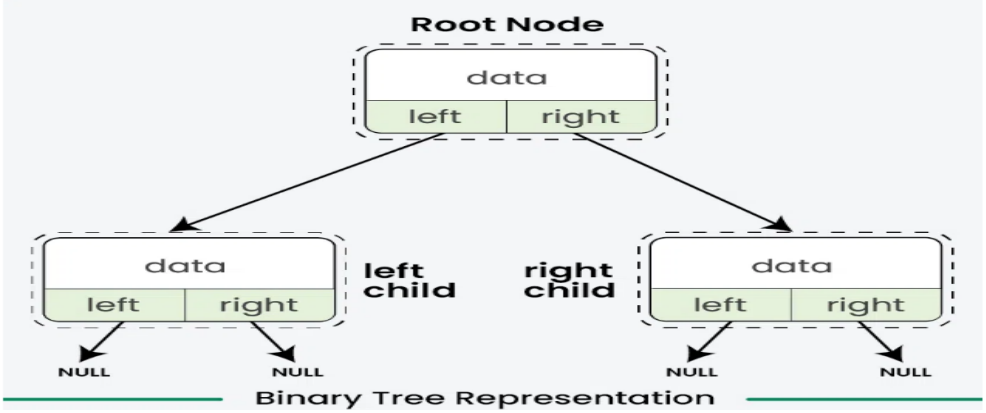


Binary tree is a tree data structure(non-linear) in which each node can have at most two children which are referred to as the left child and the right child.
The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.

Representation of Binary Tree

Each node in a Binary Tree has three parts:

- Data
- Pointer to the left child
- Pointer to the right child

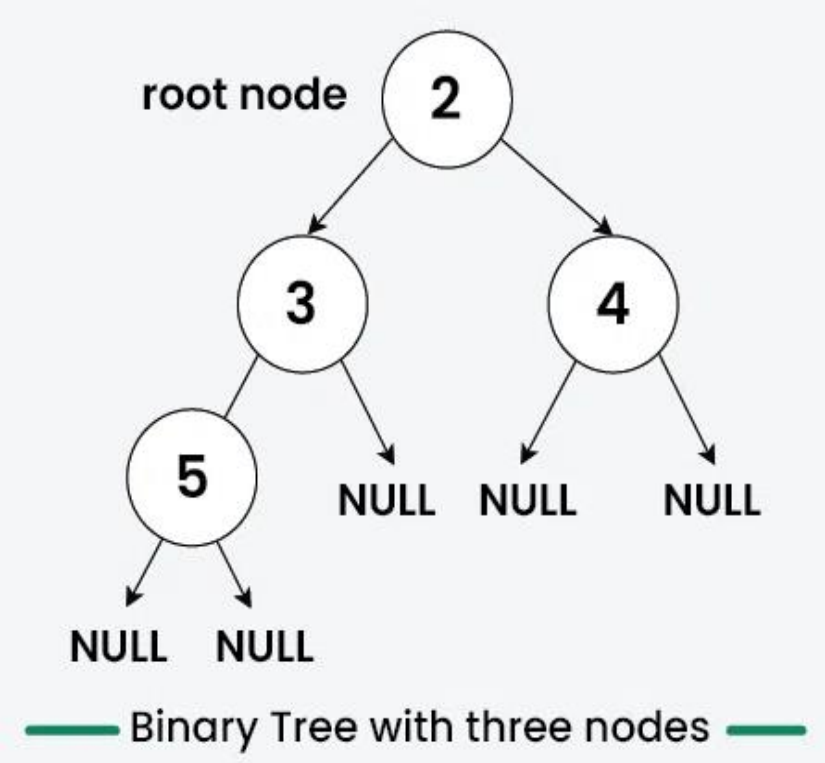


Create/Declare a Node of Circular Linked List
Syntax to declare a Node of Binary Tree in C++ language :

```
// Use any below method to implement Nodes of binary tree

// 1: Using struct
struct Node {
    int data;
    Node* left, * right;

    Node(int key) {
        data = key;
        left = nullptr;
        right = nullptr;
    }
};

// 2: Using class
class Node {
public:
    int data;
    Node* left, * right;

    Node(int key) {
        data = key;
        left = nullptr;
        right = nullptr;
    }
};
```

Example for Creating a Binary Tree

Here's an example of creating a Binary Tree with four nodes (2, 3, 4, 5)




```
#include <iostream>
using namespace std;
struct Node{
   int data;
   Node *left, *right;
   Node(int d){
      data = d;
      left = NULL;
      right = NULL;
   }
};
int main(){
   // Initilize and allocate memory for tree nodes
   Node* firstNode = new Node(2);
   Node* secondNode = new Node(3);
   Node* thirdNode = new Node(4);
   Node* fourthNode = new Node(5);
   // Connect binary tree nodes
   firstNode->left = secondNode;
   firstNode->right = thirdNode;
   secondNode->left = fourthNode;
   return 0;
}
```

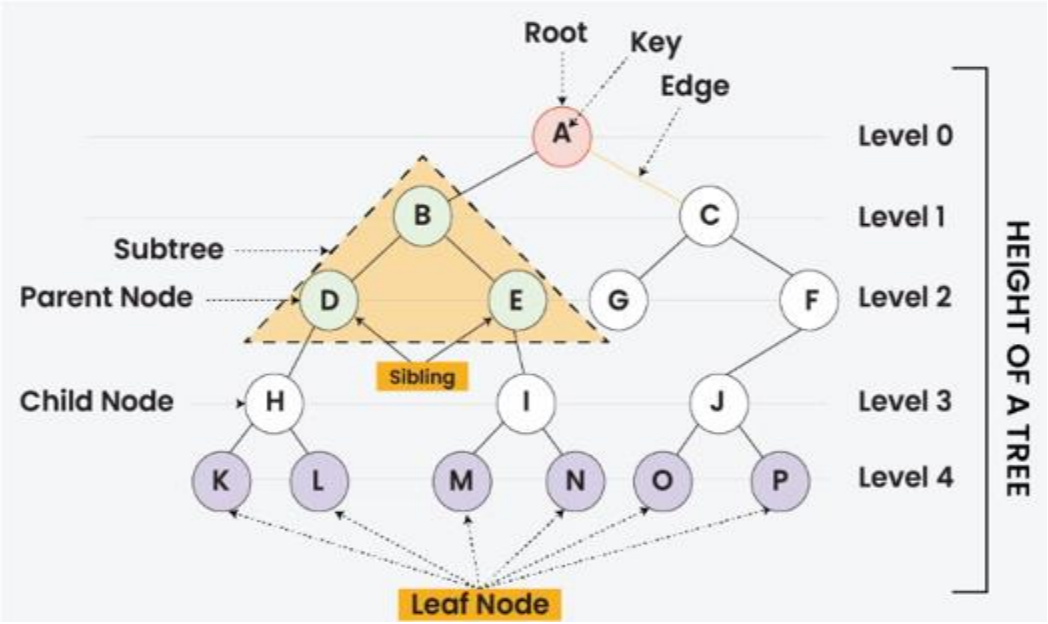
In the above code, we have created four tree nodes firstNode, secondNode, thirdNode and fourthNode having values 2, 3, 4 and 5 respectively.

- **After creating three nodes, we have connected these node to form the tree structure like mentioned in above image.**
- **Connect the secondNode to the left of firstNode by firstNode->left = secondNode**
- **Connect the thirdNode to the right of firstNode by firstNode->right = thirdNode**
- **Connect the fourthNode to the left of secondNode by secondNode->left = fourthNode**

Terminologies in Binary Tree

- **Nodes: The fundamental part of a binary tree, where each node contains data and link to two child nodes.**
- **Root**: The topmost node in a tree is known as the root node. It has no parent and serves as the starting point for all nodes in the tree.
- **Parent Node**: A node that has one or more child nodes. In a binary tree, each node can have at most two children.
- **Child Node**: A node that is a descendant of another node (its parent).
- **Leaf Node**: A node that does not have any children.
- **Internal Node**: A node that has at least one child. This includes all nodes except the **root** and the **leaf** nodes.
- **Depth of a Binary Tree**: The number of edges from a specific node to the root node. The depth of the **root** node is zero.
- **Height of a Binary Tree**: The number of nodes from the deepest leaf node to the root node.

The diagram below shows all these terms in a binary tree.



Properties of Binary Tree

- **The maximum number of nodes at level L of a binary tree is 2^L**
- **The maximum number of nodes in a binary tree of height H is $2^H - 1$**
- **Total number of leaf nodes in a binary tree = total number of nodes with 2 children + 1**
- **In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $Log_2(N+1)$**
- **A Binary Tree with L leaves has at least | Log2L |+ 1 levels**

Types of Binary Tree

Binary Tree can be classified into multiples types based on multiple factors:

- **On the basis of Number of Children**
  - **Full Binary Tree**
  - **Degenerate Binary Tree**
  - **Skewed Binary Trees**
- **On the basis of Completion of Levels**
  - **Complete Binary Tree**
  - **Perfect Binary Tree**
  - **Balanced Binary Tree**
- **On the basis of Node Values:**

- **Binary Search Tree**
- **AVL Tree**
- **Red Black Tree**
- **B Tree**
- **B+ Tree**
- **Segment Tree**

Operations On Binary Tree

Following is a list of common operations that can be performed on a binary tree:

1. Traversal in Binary Tree

Traversal in Binary Tree involves visiting all the nodes of the binary tree. Tree Traversal algorithms can be classified broadly into two categories, DFS and BFS:

Depth-First Search (DFS) algorithms: DFS explores as far down a branch as possible before backtracking. It is implemented using recursion. The main traversal methods in DFS for binary trees are:

- Preorder Traversal (current-left-right): Visits the node first, then left subtree, then right subtree.
- Inorder Traversal (left-current-right): Visits left subtree, then the node, then the right subtree.
- Postorder Traversal (left-right-current): Visits left subtree, then right subtree, then the node.

Breadth-First Search (BFS) algorithms: BFS explores all nodes at the present

depth before moving on to nodes at the next depth level. It is typically implemented using a queue. BFS in a binary tree is commonly referred to as Level Order Traversal.

Below is the implementation of traversals algorithm in binary tree:

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* left, * right;

    Node(int d) {
        data = d;
        left = nullptr;
        right = nullptr;
    }
};

// In-order DFS: Left, Root, Right
void inOrderDFS(Node* node) {
    if (node == nullptr) return;

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

// Pre-order DFS: Root, Left, Right
void preOrderDFS(Node* node) {
    if (node == nullptr) return;

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

// Post-order DFS: Left, Right, Root
void postOrderDFS(Node* node) {
    if (node == nullptr) return;

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

void BFS(Node* root) {

    if (root == nullptr) return;
    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* node = q.front();
```

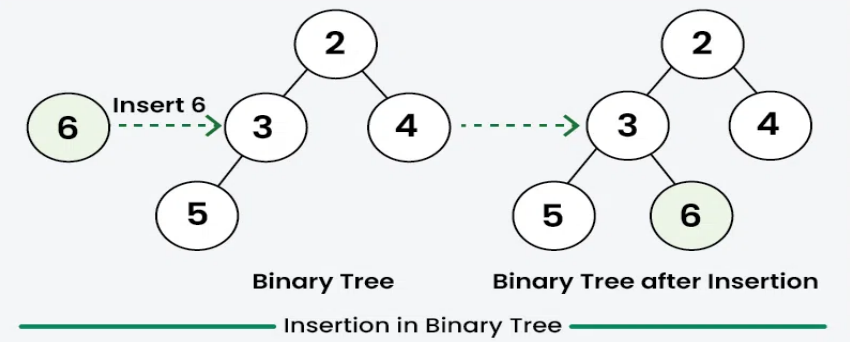
```
        q.pop();
        cout << node->data << " ";
        if (node->left != nullptr) q.push(node->left);
        if (node->right != nullptr) q.push(node->right);

    }
}

int main() {
    Node* root = new Node(2);
    root->left = new Node(3);
    root->right = new Node(4);
    root->left->left = new Node(5);

    cout << "In-order DFS: ";
    inOrderDFS(root);

    cout << "\nPre-order DFS: ";
    preOrderDFS(root);

    cout << "\nPost-order DFS: ";
    postOrderDFS(root);

    cout << "\nLevel order: ";
    BFS(root);

    return 0;
}
```

Output:

```
In-order DFS: 5 3 2 4
Pre-order DFS: 2 3 5 4
Post-order DFS: 5 3 4 2
Level order: 2 3 4 5
```

2. Insertion in Binary Tree

Inserting elements means add a new node into the binary tree. As we know that there is no such ordering of elements in the binary tree, So we do not have to worry about the ordering of node in the binary tree. We would first creates a root node in case of empty tree. Then subsequent insertions involve iteratively searching for an empty place at each level of the tree. When an empty left or right child is found then new node is inserted there. By convention, insertion always starts with the left child node.




```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* left, * right;
    Node(int k) {
        data = k;
        left = right = nullptr;
    }
};

// Function to insert a new node in the binary tree
Node* insert(Node* root, int key) {
    // If the tree is empty, create the root node
    if (root == nullptr) {
        root = new Node(key);
        return root;
    }
    // Create a queue for level order traversal
    queue<Node*> q;
    q.push(root);

    // Do level order traversal until we find an empty place
    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();

        // If left child is empty, insert the new node here
        if (temp->left == nullptr) {
            temp->left = new Node(key);
            break;
        } else {
            q.push(temp->left);
        }
        // If right child is empty, insert the new node here
        if (temp->right == nullptr) {
            temp->right = new Node(key);
            break;
        } else {
```

```
            q.push(temp->right);
        }
    }
    return root;
}

void inorder(Node* root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main() {
    Node* root = new Node(2);
    root->left = new Node(3);
    root->right = new Node(4) ;
    root->left->left = new Node(5);

    cout << "Inorder traversal before insertion: ";
    inorder(root);
    cout << endl;

    int key = 6;
    root = insert(root, key);

    cout << "Inorder traversal after insertion: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

Output

```
Inorder traversal before insertion: 5 3 2 4
Inorder traversal after insertion: 5 3 6 2 4
```

3. Searching in Binary Tree

Searching for a value in a binary tree means looking through the tree to find a node that has that value. Since binary trees do not have a specific order like binary search trees, we typically use any traversal method to search. The most common methods are **depth-first search (DFS)** and **breadth-first search (BFS)**. In **DFS**, we start from the **root** and explore the depth nodes first. In BFS, we explore all the nodes at the present depth level before moving on to the nodes at the next level. We continue this process until we either find the node with the desired value or reach the end of the tree. If the tree is empty or the value isn't

found after exploring all possibilities, we conclude that the value does not exist in the tree.

Here is the implementation of searching in a binary tree using Depth-First Search (DFS)

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

struct Node{
   int data;
   Node *left, *right;
   Node(int k){
      data = k;
      left = right = nullptr;
   }
};

// Function to search for a value in the binary tree using DFS
bool searchDFS(Node *root, int value){
   // Base case: If the tree is empty or we've reached a leaf node
   if (root == nullptr) return false;

   // If the node's data is equal to the value we are searching for
   if (root->data == value) return true;

   // Recursively search in the left and right subtrees
   bool left_res = searchDFS(root->left, value);
   bool right_res = searchDFS(root->right, value);

   return left_res || right_res;
}

int main()
{
   Node *root = new Node(2);
   root->left = new Node(3);
   root->right = new Node(4);
   root->left->left = new Node(5);
   root->left->right = new Node(6);

   int value = 6;
   if (searchDFS(root, value))
      cout << value << " is found in the binary tree" << endl;
   else
      cout << value << " is not found in the binary tree" << endl;

   return 0;
}
```

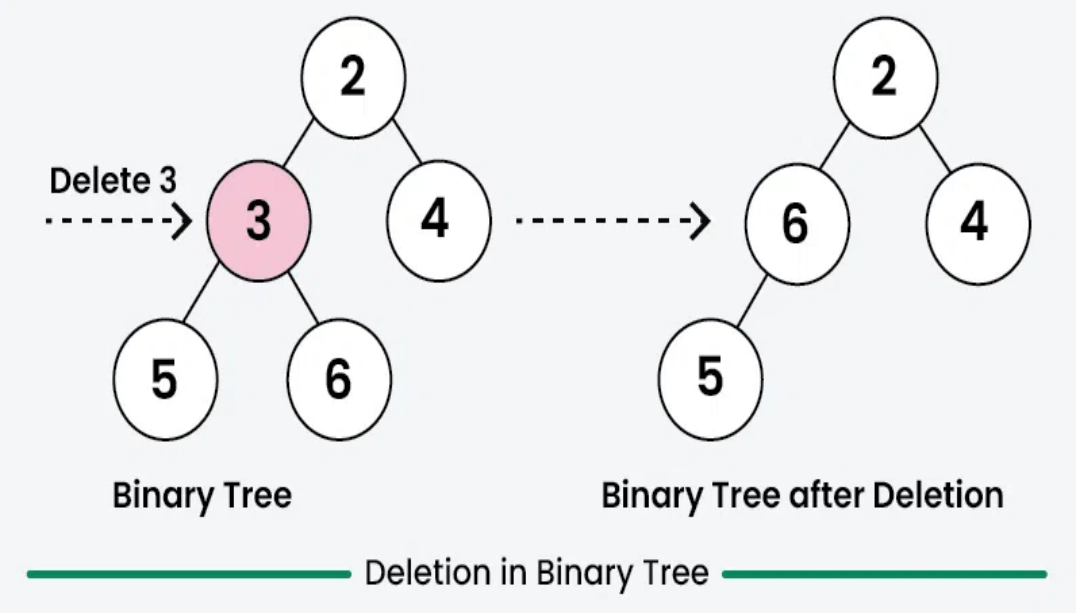
Output

```
6 is found in the binary tree
```

4. Deletion in Binary Tree

Deleting a node from a binary tree means removing a specific node while keeping the tree's structure. First, we need to find the node that want to delete by traversing through the tree using any traversal method. Then replace the node's value with the value of the last node in the tree (found by traversing to the rightmost leaf), and then delete that last node. This way, the tree structure won't be effected. And remember to check for special cases, like trying to delete from an empty tree, to avoid any issues.

Note: There is no specific rule of deletion but we always make sure that during deletion the binary tree proper should be preserved.




```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* left, * right;
    Node(int k) {
        data = k;
        left = right = nullptr;
    }
};

// Function to delete a node from the binary tree
Node* deleteNode(Node* root, int val) {
    if (root == nullptr) return nullptr;
```

```
// Use a queue to perform BFS
queue<Node*> q;
q.push(root);
Node* target = nullptr;

// Find the target node
while (!q.empty()) {
   Node* curr = q.front();
   q.pop();

   // Check for current node is the target node to delete
   if (curr->data == val) {
      target = curr;
      break;
   }
   // Add children to the queue
   if (curr->left) q.push(curr->left);
   if (curr->right) q.push(curr->right);
}
// If target node is not found, return the original tree
if (target == nullptr) return root;

// Find the deepest rightmost node and its parent
pair<Node*, Node*> last = {nullptr, nullptr};
queue<pair<Node*, Node*>> q1;
q1.push({root, nullptr});

while (!q1.empty()) {
   auto curr = q1.front();
   q1.pop();

   // Update the last
   last = curr;

   if (curr.first->left)
     q1.push({curr.first->left, curr.first});
   if (curr.first->right)
     q1.push({curr.first->right, curr.first});
}

Node* lastNode = last.first;
Node* lastParent = last.second;

// Replace target's value with the last node's value
target->data = lastNode->data;

// Remove the last node
if (lastParent) {
   if (lastParent->left==lastNode)lastParent->left = nullptr;
   else lastParent->right = nullptr;
   delete lastNode;
```

```
            } else {
                // If the last node was the root
                delete lastNode;
                return nullptr;
            }
            return root;
        }

        void inOrder(Node* root) {
            if (root == nullptr) return;
            inOrder(root->left);
            cout << root->data << " ";
            inOrder(root->right);
        }

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

            cout << "Original tree (in-order): ";
            inOrder(root);

            int valToDel = 3;
            root = deleteNode(root, valToDel);

            cout <<"\nTree after deleting " << valToDel << " (in-order): ";
            inOrder(root);
            cout << endl;

            return 0;
}
```

```
Original tree (in-order): 5 3 6 2 4
Tree after deleting 3 (in-order): 5 6 2 4
```

Complexity Analysis of Binary Tree Operations

Here's the complexity analysis for specific binary tree operations:

Operation	Time Complexity	Auxiliary Space
In-Order Traversal	O(n)	O(n)
Pre-Order Traversal	O(n)	O(n)
Post-Order Traversal	O(n)	O(n)
Insertion (Unbalanced)	O(n)	O(n)
Searching (Unbalanced)	O(n)	O(n)
Deletion (Unbalanced)	O(n)	O(n)

Advantages of Binary Tree

- **Efficient Search**: Binary Search Trees (a variation of Binary Tree) are efficient when searching for a specific element, as each node has at most two child nodes when compared to linked list and arrays
- **Memory Efficient**: Binary trees require lesser memory as compared to other tree data structures, therefore they are memory-efficient.
- Binary trees are relatively easy to implement and understand as each node has at most two children, left child and right child.

Disadvantages of Binary Tree

- **Limited structure:** Binary trees are limited to two child nodes per node, which can limit their usefulness in certain applications. For example, if a tree requires more than two child nodes per node, a different tree structure may be more suitable.
- **Unbalanced trees:** Unbalanced binary trees, where one subtree is significantly larger than the other, can lead to inefficient search operations. This can occur if the tree is not properly balanced or if data is inserted in a non-random order.

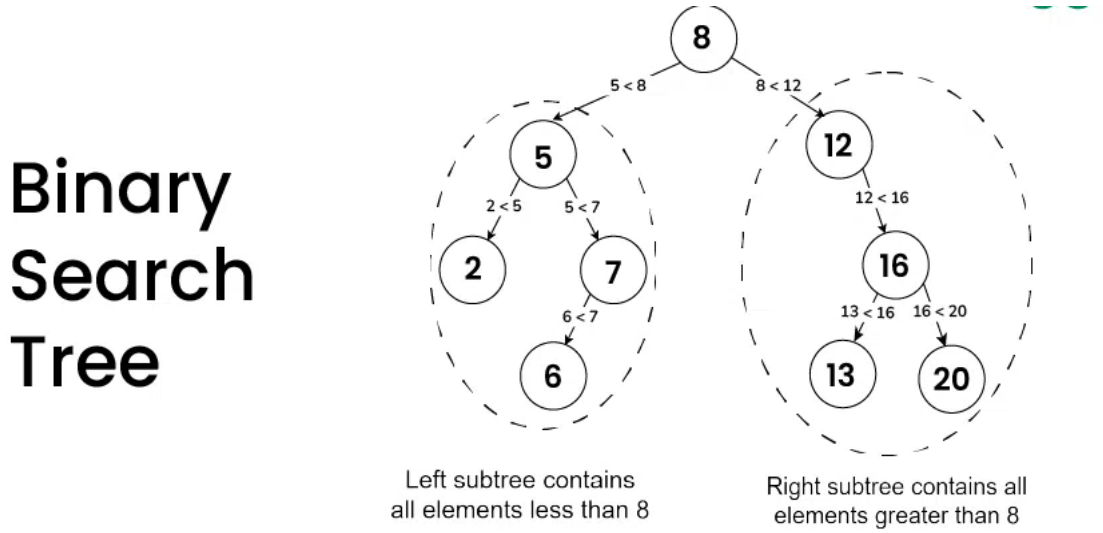
- **Space inefficiency:** Binary trees can be space inefficient when compared to other data structures. This is because each node requires two child pointers, which can be a significant amount of memory overhead for large trees.
- **Slow performance in worst-case scenarios:** In the worst-case scenario, a binary tree can become degenerate or skewed, meaning that each node has only one child. In this case, search operations can degrade to O(n) time complexity, where n is the number of nodes in the tree.

Applications of Binary Tree

- **Binary Tree can be used to represent hierarchical data.**
- Huffman Coding trees are used in **data compression algorithms**.
- Priority Queue is another application of binary tree that is used for searching maximum or minimum in O(1) time complexity.
- Useful for indexing segmented at the database is useful in storing cache in the system,
- Binary trees can be used to implement decision trees, a type of machine learning algorithm used for classification and regression analysis.

Binary Search Tree

A Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner. Each node in a Binary Search Tree has at most two children, a left child and a right child, with the left child containing values less than the parent node and the right child containing values greater than the parent node. This hierarchical structure allows for efficient searching, insertion, and deletion operations on the data stored in the tree.



Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

Properties of Binary Search Tree:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes(BST may have duplicate values with different handling approaches).

Basic Operations on Binary Search Tree:

1. Searching a node in BST:

Searching in BST means to locate a specific node in the data structure. In Binary search tree, searching a node is easy because of its a specific order. The steps of searching a node in Binary Search tree are listed as follows –

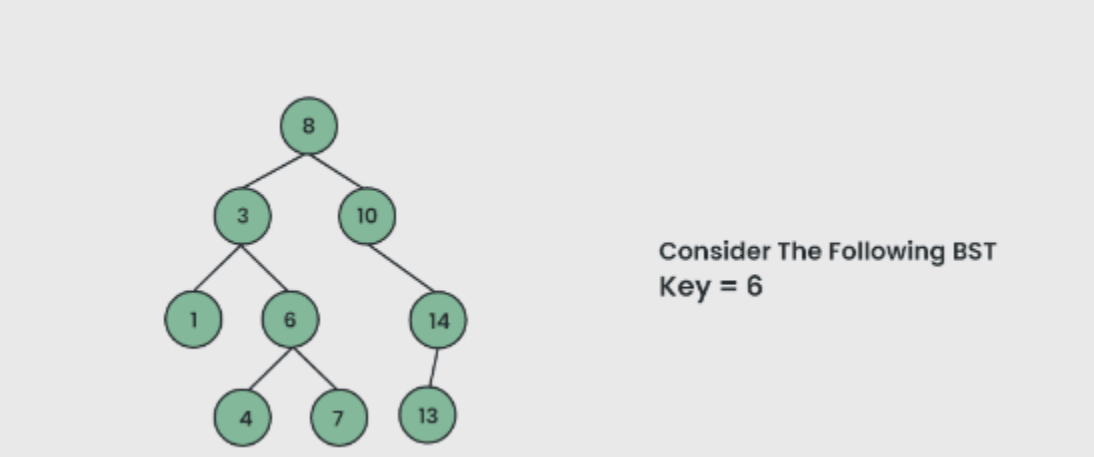
1. First, compare the element to be searched with the root element of the tree.
    - If root is matched with the target element, then return the node's location.

- 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.
- If it is larger than the root element, then move to the right subtree.

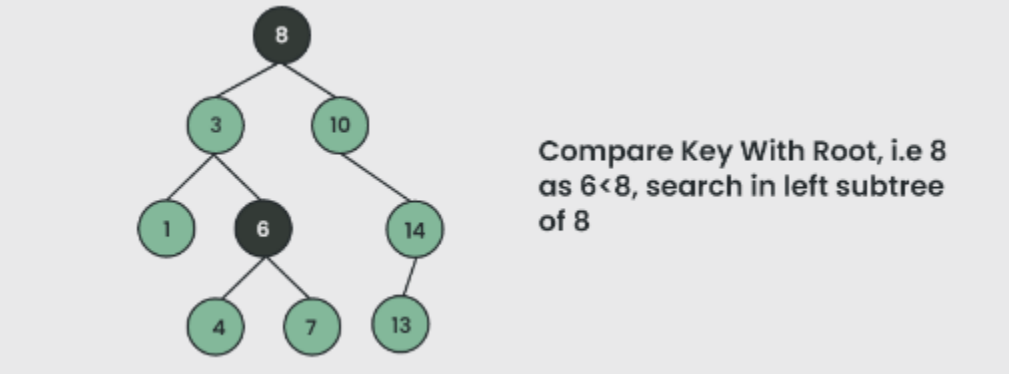
2. Repeat the above procedure recursively until the match is found.
3. If the element is not found or not present in the tree, then return NULL.

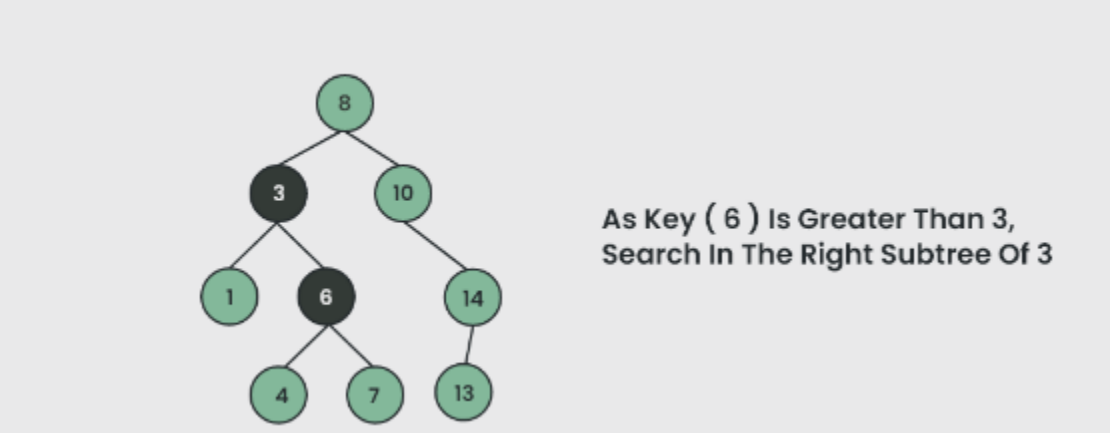
Now, let's understand the searching in binary tree using an example:

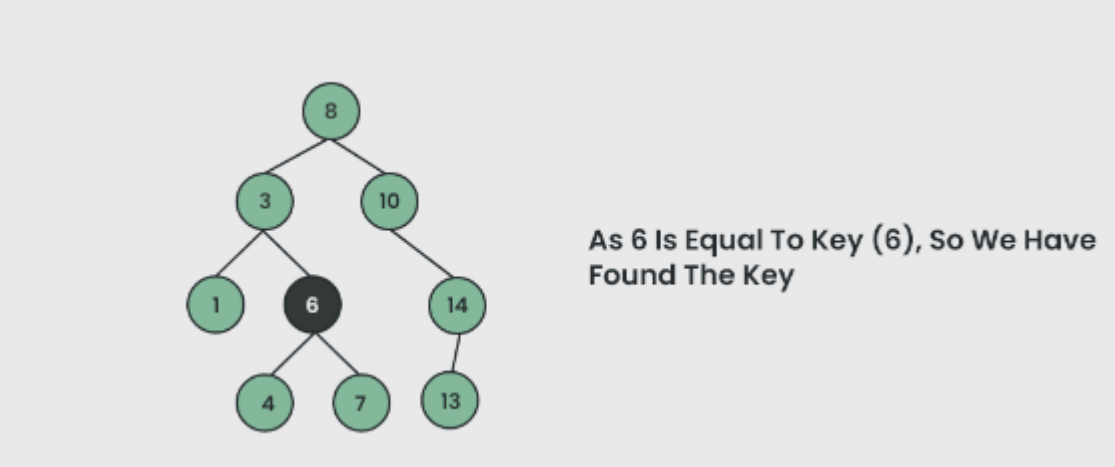
Below is given a BST and we have to search for element 6.



Searching In BST







Below is the implementation of searching in BST using C++ :

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

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int item) {
        key = item;
        left = right = NULL;
    }
};

// function to search a key in a BST
Node* search(Node* root, int key) {

    // Base Cases: root is null or key
    // is present at root
    if (root == NULL || root->key == key)
```

```
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

// Driver Code
int main() {

    // Creating a hard coded tree for keeping
    // the length of the code small. We need
    // to make sure that BST properties are
    // maintained if we try some other cases.
    Node* root = new Node(50);
    root->left = new Node(30);
    root->right = new Node(70);
    root->left->left = new Node(20);
    root->left->right = new Node(40);
    root->right->left = new Node(60);
    root->right->right = new Node(80);

    (search(root, 19) != NULL)? cout << "Found\n":
                        cout << "Not Found\n";


    (search(root, 80) != NULL)? cout << "Found\n":
                        cout << "Not Found\n";

    return 0;
}
```

Output

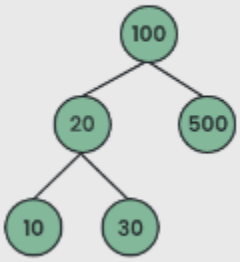
```
Not Found
Found
```

Time Complexity : O(h) where h is height of BST

2. **Insert a node into a BST:**

A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

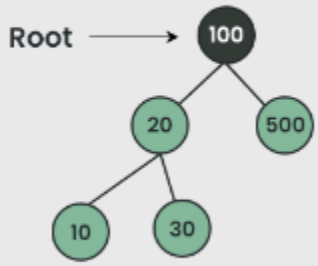
Consider The Following BST



X = 40 (The Node To Be Inserted)

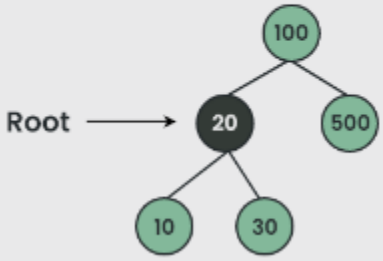
Insertion In BST

STEP 1 : Comparing X with Root Node

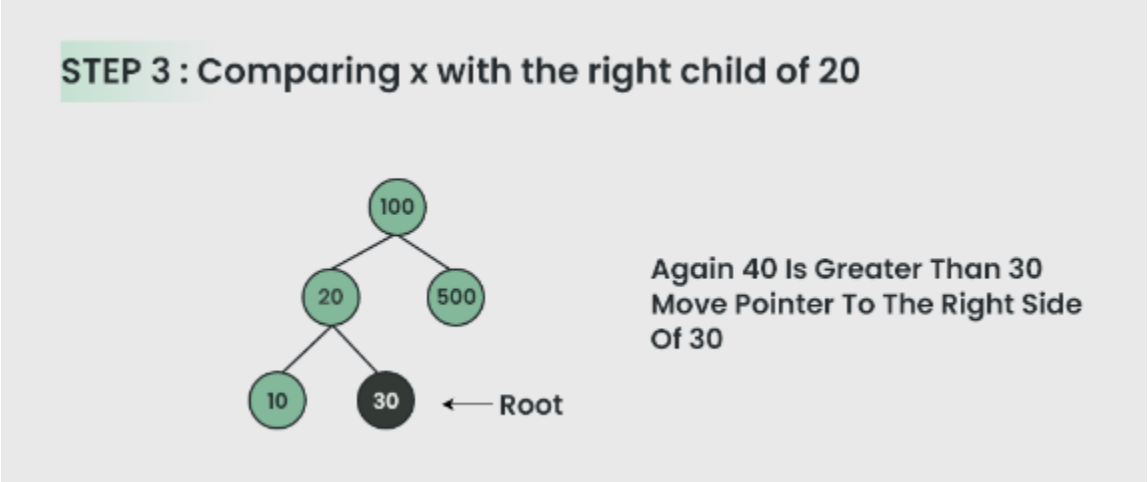


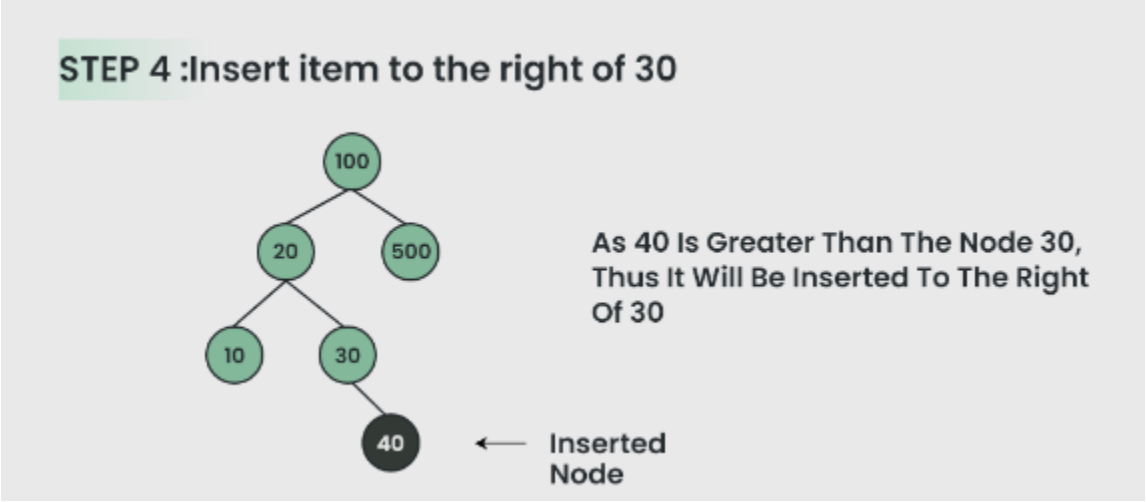
Since 100 Is Greater Than 40.
Move Pointer To The Left Child (20)

STEP 2 : Comparing X with left child of root node



Since 20 Is Less Than 40, Move
Pointer To The Right Child (30)





Below is the implementation of the Insertion of a single node in Binary Search Tree:

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

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int item) {
        key = item;
        left = NULL;
        right = NULL;
    }
};

// A utility function to insert a new node with
// the given key
Node* insert(Node* node, int key) {
```

```
    // If the tree is empty, return a new node
    if (node == NULL)
        return new Node(key);

    // If the key is already present in the tree,
    // return the node
    if (node->key == key)
        return node;

    // Otherwise, recur down the tree/ If the key
    // to be inserted is greater than the node's key,
    // insert it in the right subtree
    if (node->key < key)
        node->right = insert(node->right, key);

    // If the key to be inserted is smaller than
    // the node's key,insert it in the left subtree
    else
        node->left = insert(node->left, key);

    // Return the (unchanged) node pointer
    return node;
}

// A utility function to do inorder tree traversal
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

// Driver program to test the above functions
int main() {
    // Creating the following BST
    //      50
    //     /  \
    //    30   70
    //   / \   / \
    //  20 40 60 80

    Node* root = new Node(50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Print inorder traversal of the BST
```

```
        inorder(root);

        return 0;
}
```

Output:

```
    20 30 40 50 60 70 80
```

Time Complexity : O(h) where h is height of BST

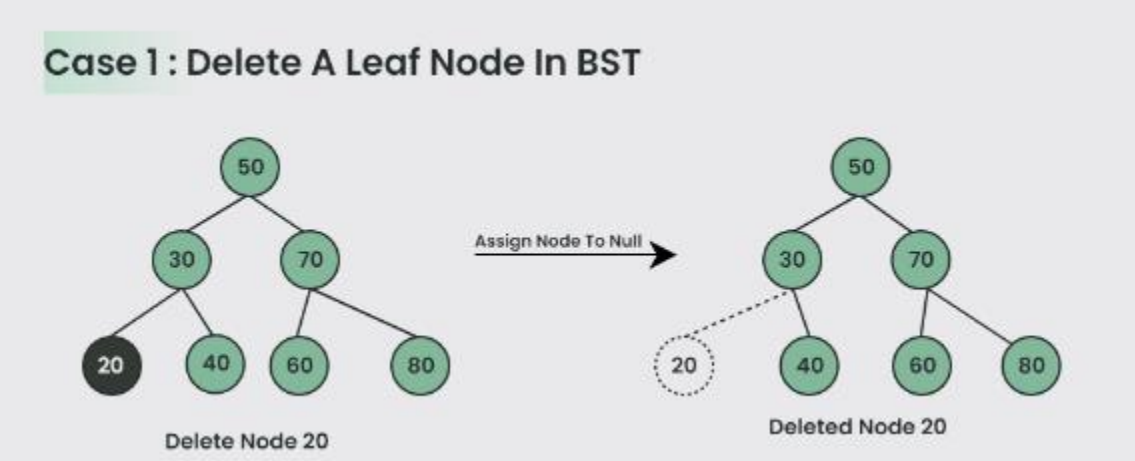
3- Delete a Node of BST:

It is used to delete a node with specific key from the BST and return the new BST.

Different scenarios for deleting the node:

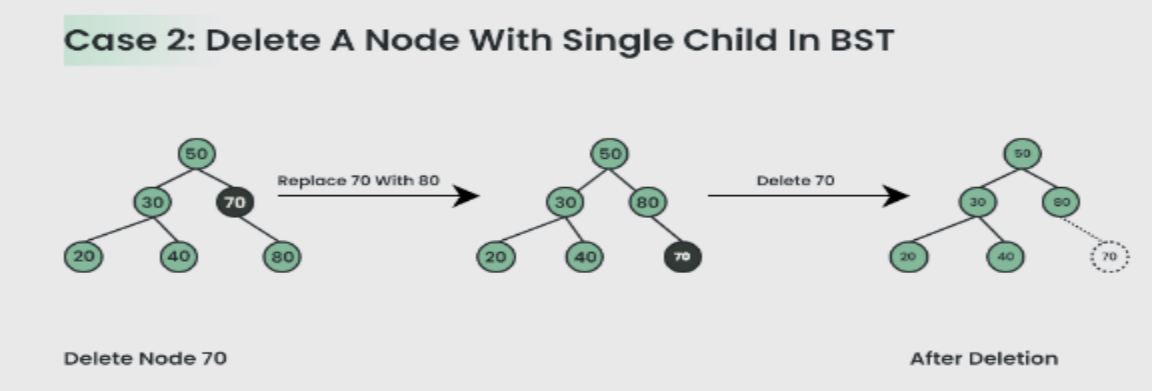
Node to be deleted is the leaf node :

Its simple you can just null it out.



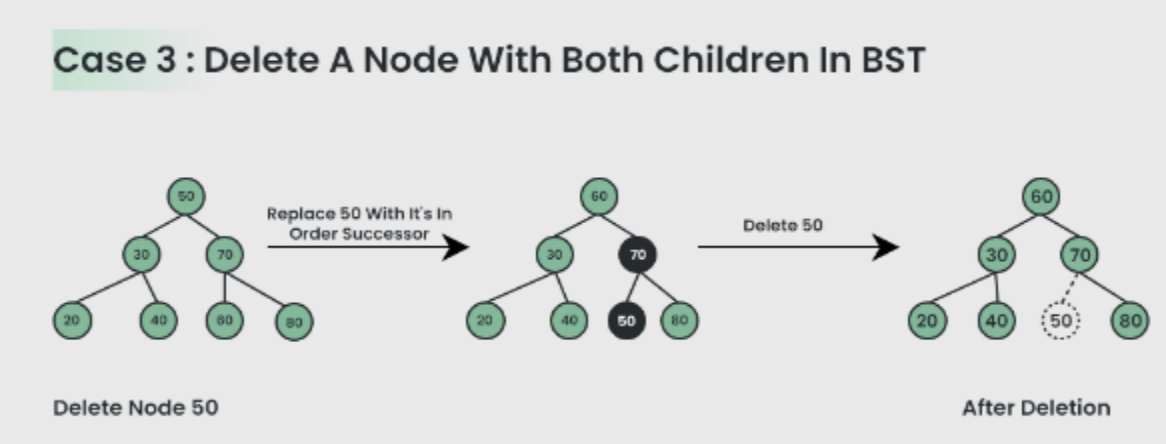
Node to be deleted has one child :

You can just replace the node with the child node.



Node to be deleted has two children :

Here we have to **delete the node is such a way, that the resulting tree follows the properties of a BST.** The trick is to find the inorder successor of the node. Copy contents of the inorder successor to the node, and delete the inorder successor.



Take Care of following things while deleting a node of a BST:

1. Need to figure out what will be the replacement of the node to be deleted.
2. Want minimal disruption to the existing tree structure
3. Can take the replacement node from the deleted nodes left or right subtree.
4. If taking if from the left subtree, we have to take the largest value in the left subtree.
5. If taking if from the right subtree, we have to take the smallest value in the right subtree.

Below is the implementation of the deletion in BST:

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k)
    {
        key = k;
        left = right = NULL;
    }
};
```

```
// Note that it is not a generic inorder
// successor function. It mainly works
// when right child is not empty which is
// the case wwe need in BST delete
Node* getSuccessor(Node* curr)
{
    curr = curr->right;
    while (curr != NULL && curr->left != NULL)
        curr = curr->left;
    return curr;
}

// This function deletes a given key x from
// the give BST and returns modified root of
// the BST (if it is modified)
Node* delNode(Node* root, int x)
{

    // Base case
    if (root == NULL)
        return root;

    // If key to be searched is in a subtree
    if (root->key > x)
        root->left = delNode(root->left, x);
    else if (root->key < x)
        root->right = delNode(root->right, x);

    // If root matches with the given key
    else {

        // Cases when root has 0 children
        // or only right child
        if (root->left == NULL) {
            Node* temp = root->right;
            delete root;
            return temp;
        }

        // When root has only left child
        if (root->right == NULL) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // When both children are present
        Node* succ = getSuccessor(root);
        root->key = succ->key;
        root->right = delNode(root->right, succ->key);
```

```
    }
    return root;
}

// Utility function to do inorder
// traversal
void inorder(Node* root)
{
    if (root != NULL) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

// Driver code
int main()
{
    Node* root = new Node(10);
    root->left = new Node(5);
    root->right = new Node(15);
    root->right->left = new Node(12);
    root->right->right = new Node(18);
    int x = 15;

    root = delNode(root, x);
    inorder(root);
    return 0;
}
```

Output

```
5 10 12 18
```

Time Complexity : O(h) where h is height of BST

4- Traversal (Inorder traversal of BST) :

- In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. We visit the left child first, then the root, and then the right child.

Below is the implementation of how to do inorder traversal of a Binary Search Tree:

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    Node* left;
    Node* right;
```

```
    Node(int k)
    {
        key = k;
        left = right = NULL;
    }
};

// Utility function to do inorder
// traversal
void inorder(Node* root)
{
    if (root != NULL) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

// Driver code
int main()
{
    Node* root = new Node(10);
    root->left = new Node(5);
    root->right = new Node(15);
    root->right->left = new Node(12);
    root->right->right = new Node(18);
    inorder(root);
    return 0;
}
```

Output

```
5 10 12 15 18
```

Time Complexity: O(N), where N is the number of nodes of the BST

Applications of BST:

- **Self-balancing binary search tree:** Self-balancing data structures such as AVL tree and Red-black tree are the most useful variations of BSTs. In these variations, we maintain the height as O(Log n) so that all operations are bounded by O(Log n). TreeSet and TreeMap in Java (or set and map in C++) are library implementations of self balancing BSTs.
- **Sorted Stream of Data** : If we wish to maintain a sorted stream of data where we wish to have operations like insert, search, delete and traversal in sorted order, BST is the most suitable data structure for this case.

- **Doubly Ended Priority Queues:** With Self Balancing BSTs, we can extract both maximum and minimum in O(Log n) time, so when we need a data structure with both operations supported efficiently, we use self balancing BSTs.

Advantages:

- **Fast search:** Searching for a specific value in a BST has an average time complexity of O(log n), where n is the number of nodes in the tree. This is much faster than searching for an element in an array or linked list, which have a time complexity of O(n) in the worst case.

- **In-order traversal:** BSTs can be traversed in-order, which visits the left subtree, the root, and the right subtree. This can be used to sort a dataset.

Disadvantages:

- **Skewed trees:** If a tree becomes skewed, the time complexity of search, insertion, and deletion operations will be O(n) instead of O(log n), which can make the tree inefficient.

- **Additional time required:** Self-balancing trees require additional time to maintain balance during insertion and deletion operations.

- **Efficiency:** For only search, insert and / or delete operations only hashing is always preferred over BSts. However if we need to maintain sorted data along with these operations, we use BST.

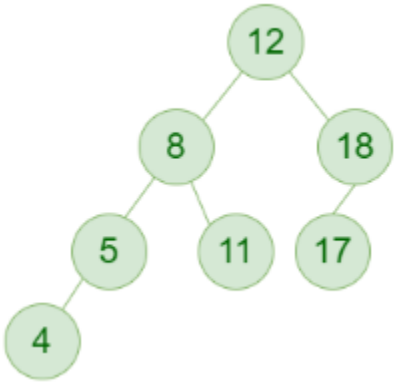
AVL TREE

*An **AVL tree** defined as a self-balancing **Binary Search Tree** (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.*

The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node.

The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.

Example of AVL Trees:



AVL Tree

The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

Operations on an AVL Tree:

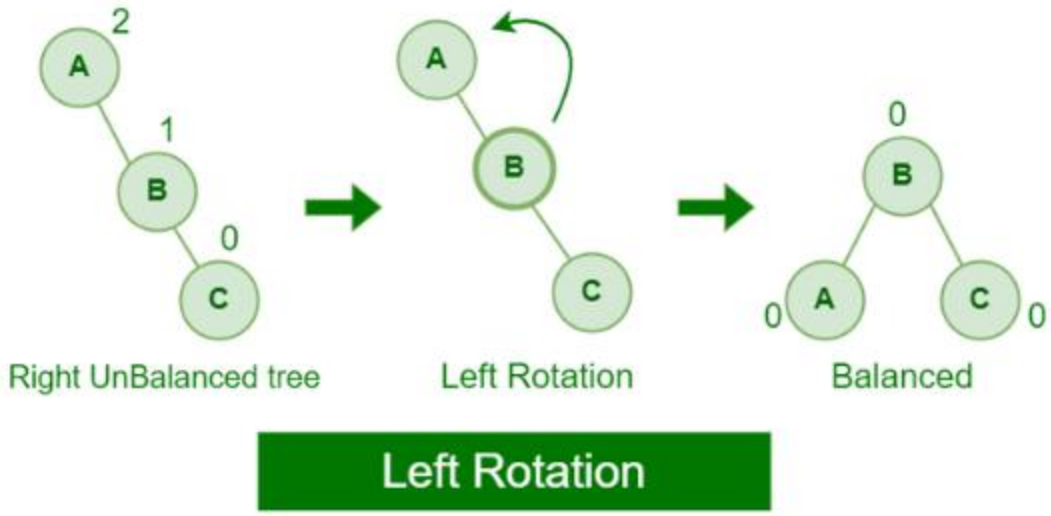
- Insertion
- Deletion
- Searching [It is similar to performing a search in BST]

Rotating the subtrees in an AVL Tree:

An AVL tree may rotate in one of the following four ways to keep itself balanced:

Left Rotation:

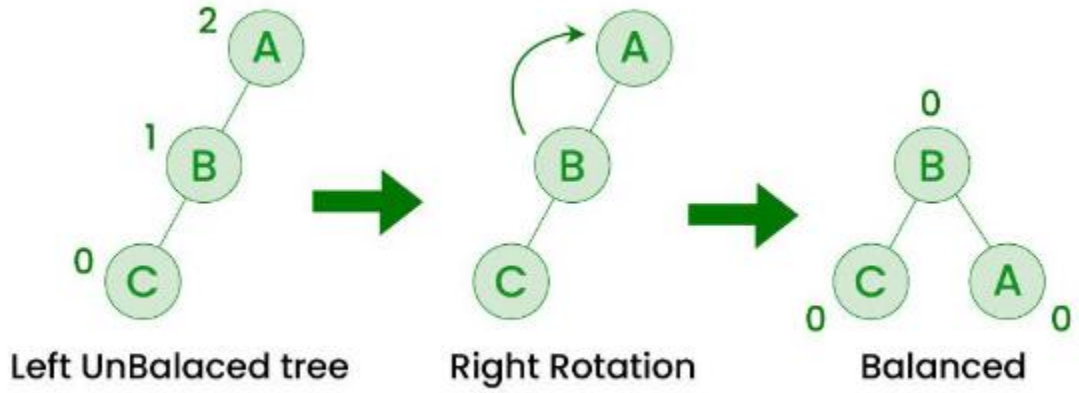
When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



A left Rotate in AVL TREE

Right Rotation:

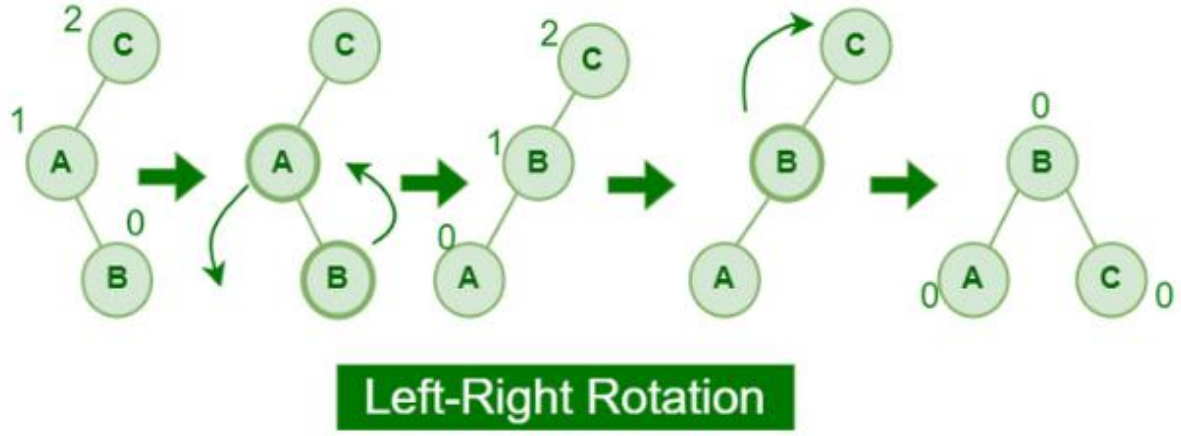
If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



A Right Rotate in AVL TREE

Left-Right Rotation:

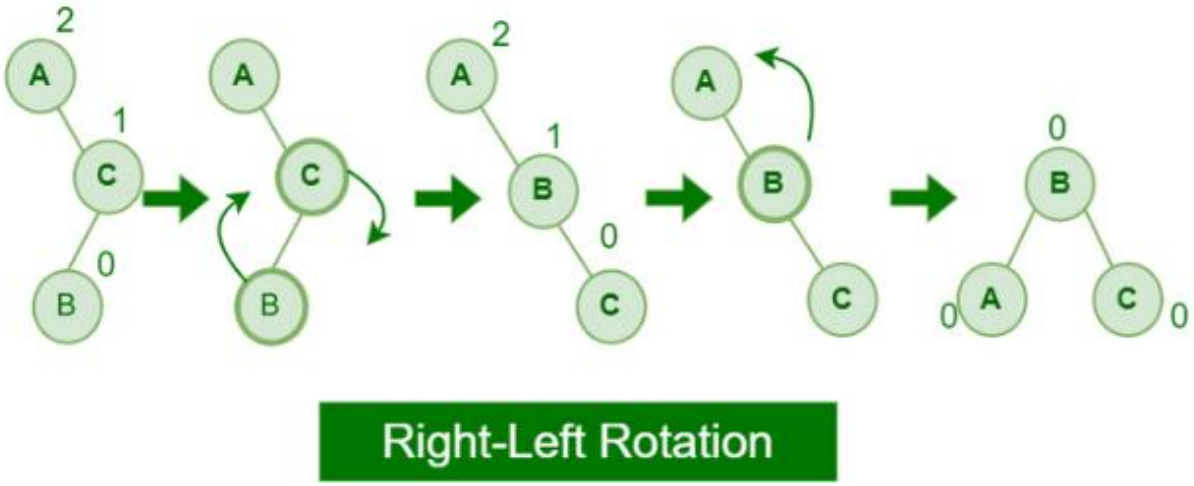
A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



A Left-Right Rotate in AVL TREE

Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



A Right-Left Rotate in AVL TREE

Advantages of AVL Tree:

1. AVL trees can self-balance themselves and therefore provides time complexity as O(Log n) for search, insert and delete.
2. It is a BST only (with balancing), so items can be traversed in sorted order.

3. Since the balancing rules are strict compared to Red Black Tree, AVL trees in general have relatively less height and hence the search is faster.
4. AVL tree is relatively less complex to understand and implement compared to Red Black Trees.

Disadvantages of AVL Tree:

1. It is difficult to implement compared to normal BST and easier compared to Red Black
2. Less used compared to Red-Black trees.
3. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.

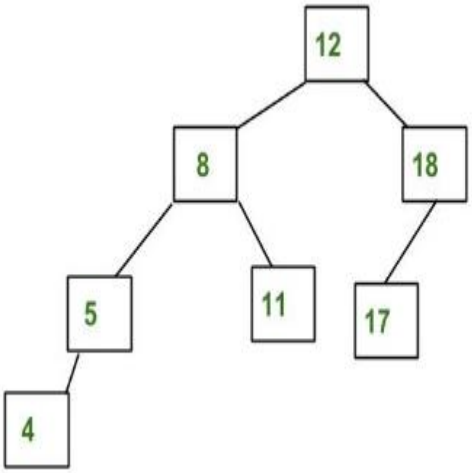
Applications of AVL Tree:

1. AVL Tree is used as a first example self balancing BST in teaching DSA as it is easier to understand and implement compared to Red Black
2. Applications, where insertions and deletions are less common but frequent data lookups along with other operations of BST like sorted traversal, floor, ceil, min and max.
3. Red Black tree is more commonly implemented in language libraries like map in C++, set in C++, TreeMap in Java and TreeSet in Java.
4. AVL Trees can be used in a real time environment where predictable and consistent performance is required.

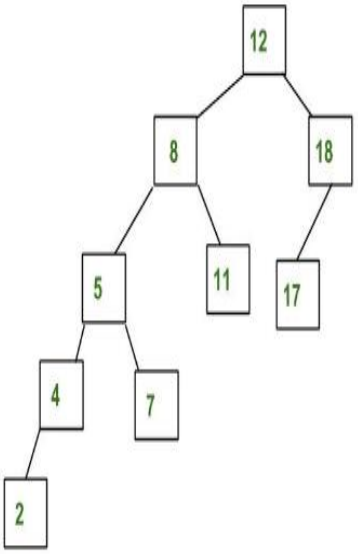
Why AVL Tree?

*Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take **O(h)** time where **h** is the height of the BST. The cost of these operations may become **O(n)** for a **skewed Binary tree**. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.*

Insertion in an AVL Tree



AVL Tree



NOT AVL Tree

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

```
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left
side) or x (on the right side)

     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - - >      T1   y
  / \       < - - - - - - - -          / \
 T1  T2     Left Rotation            T2  T3

Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

Steps to follow for insertion:

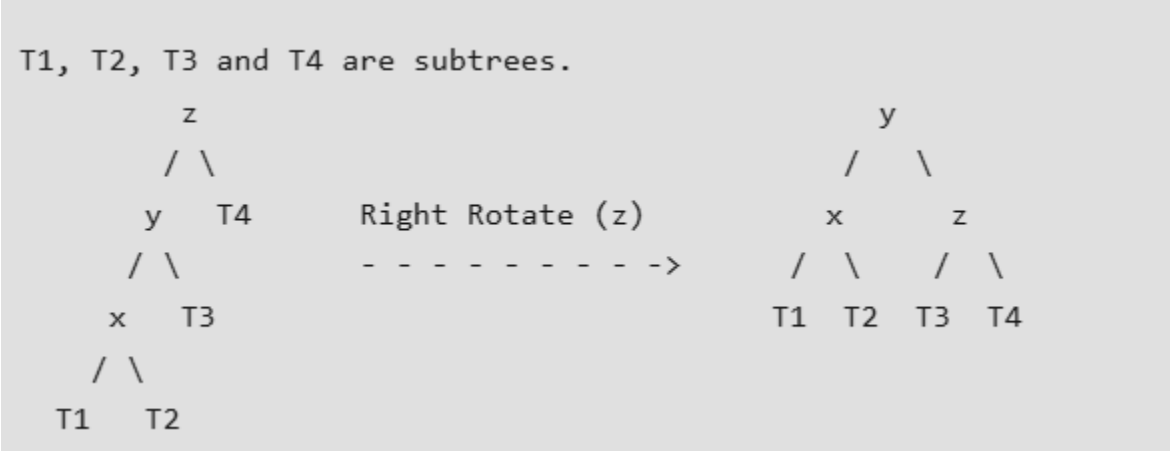
Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
    - y is the left child of z and x is the left child of y (Left Left Case)
    - y is the left child of z and x is the right child of y (Left Right Case)
    - y is the right child of z and x is the right child of y (Right Right Case)
    - y is the right child of z and x is the left child of y (Right Left Case)

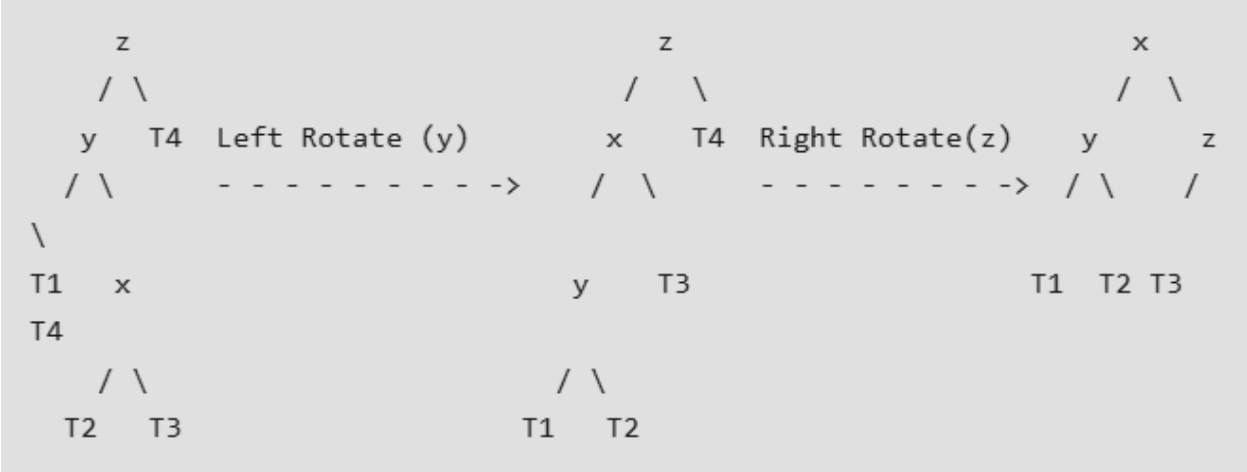
*Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to **re-balance** the subtree rooted with **z** and the complete tree becomes balanced as the height of the subtree (After*

appropriate rotations) rooted with ***z*** *becomes the same as it was before insertion.*

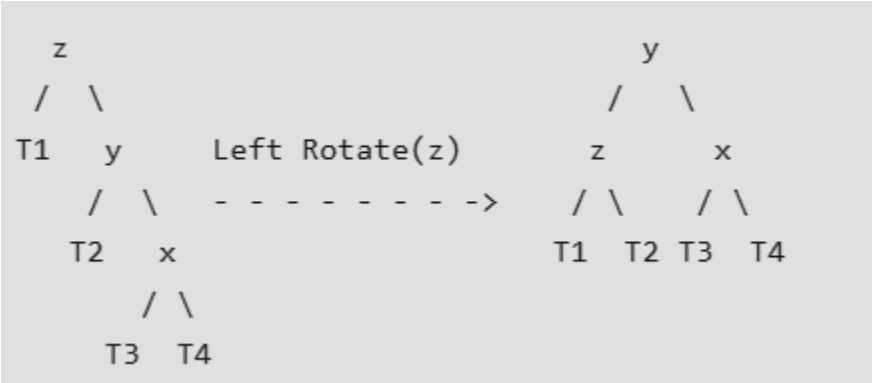
1. Left Left Case



2. Left Right Case



3. Right Right Case



4. Right Left Case

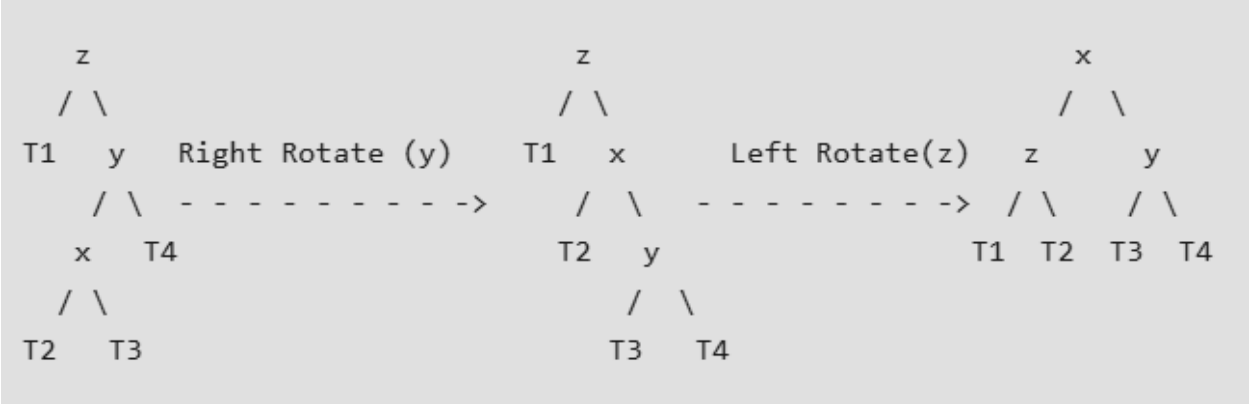
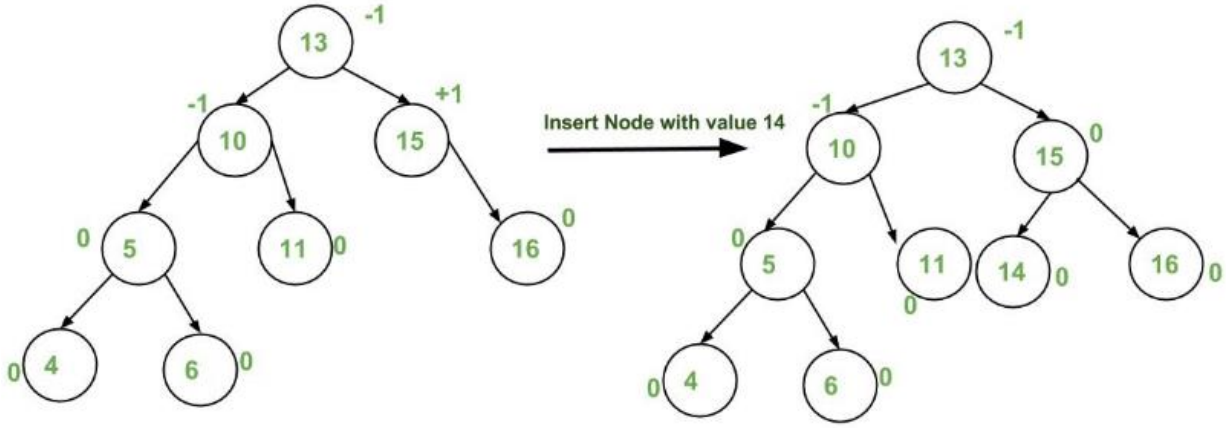
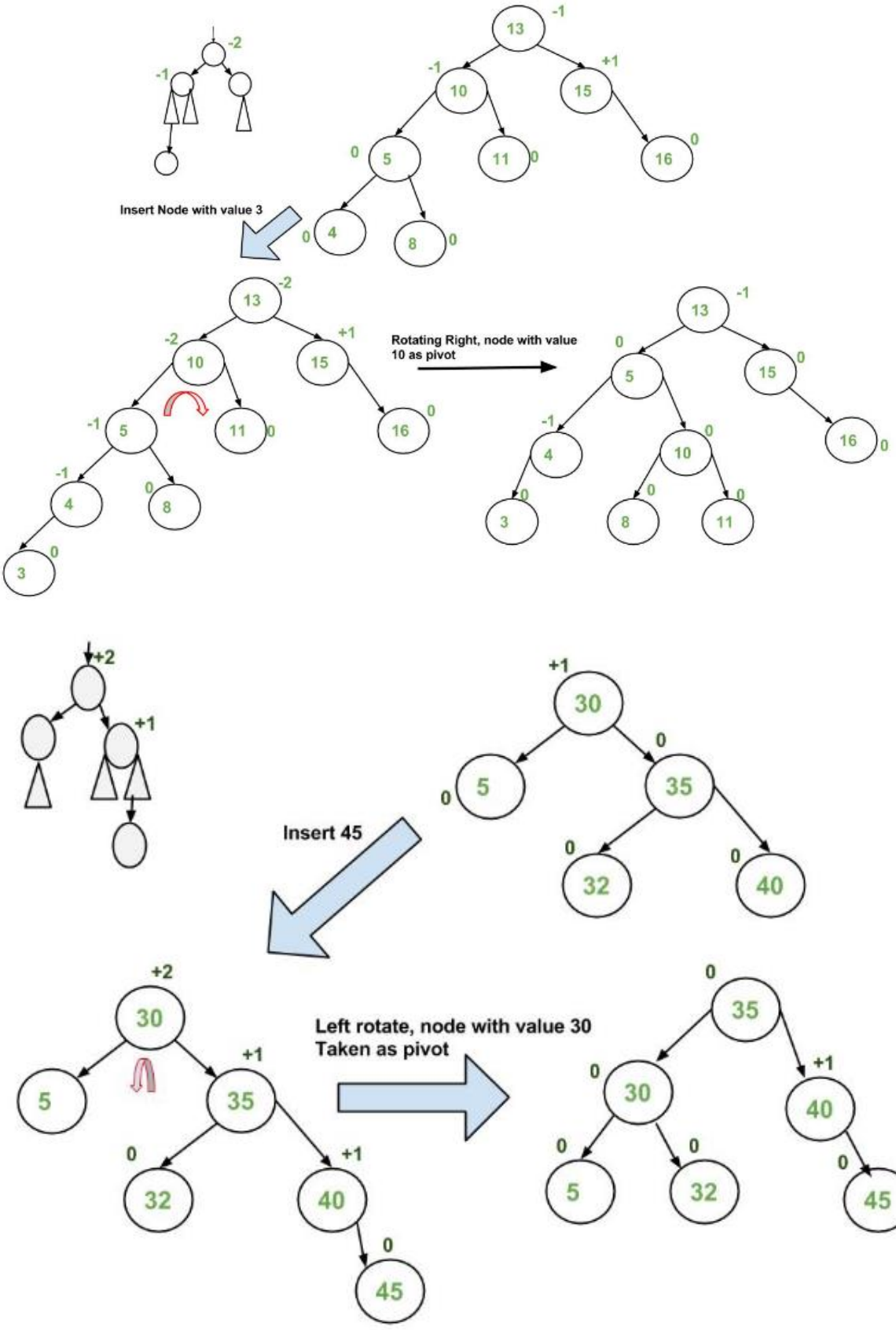
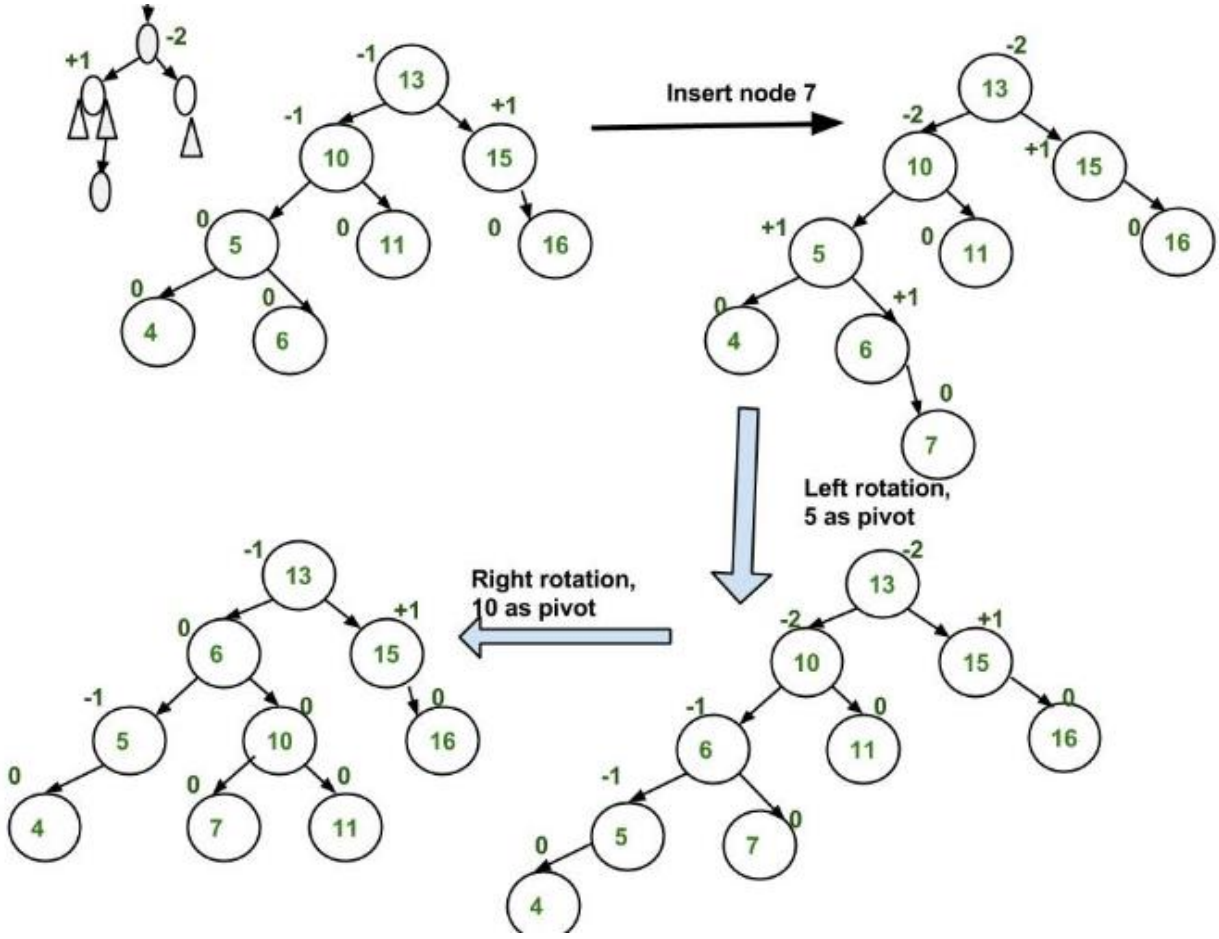
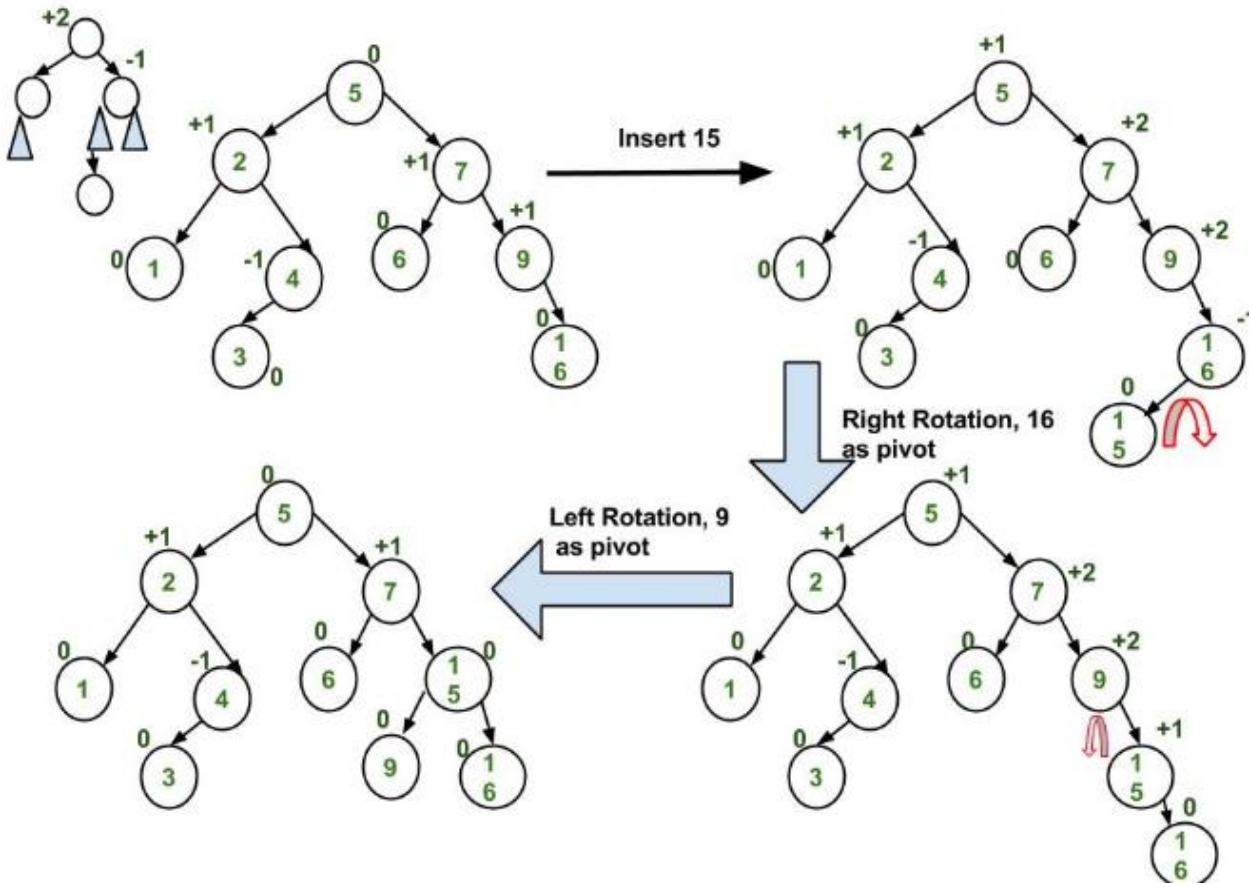


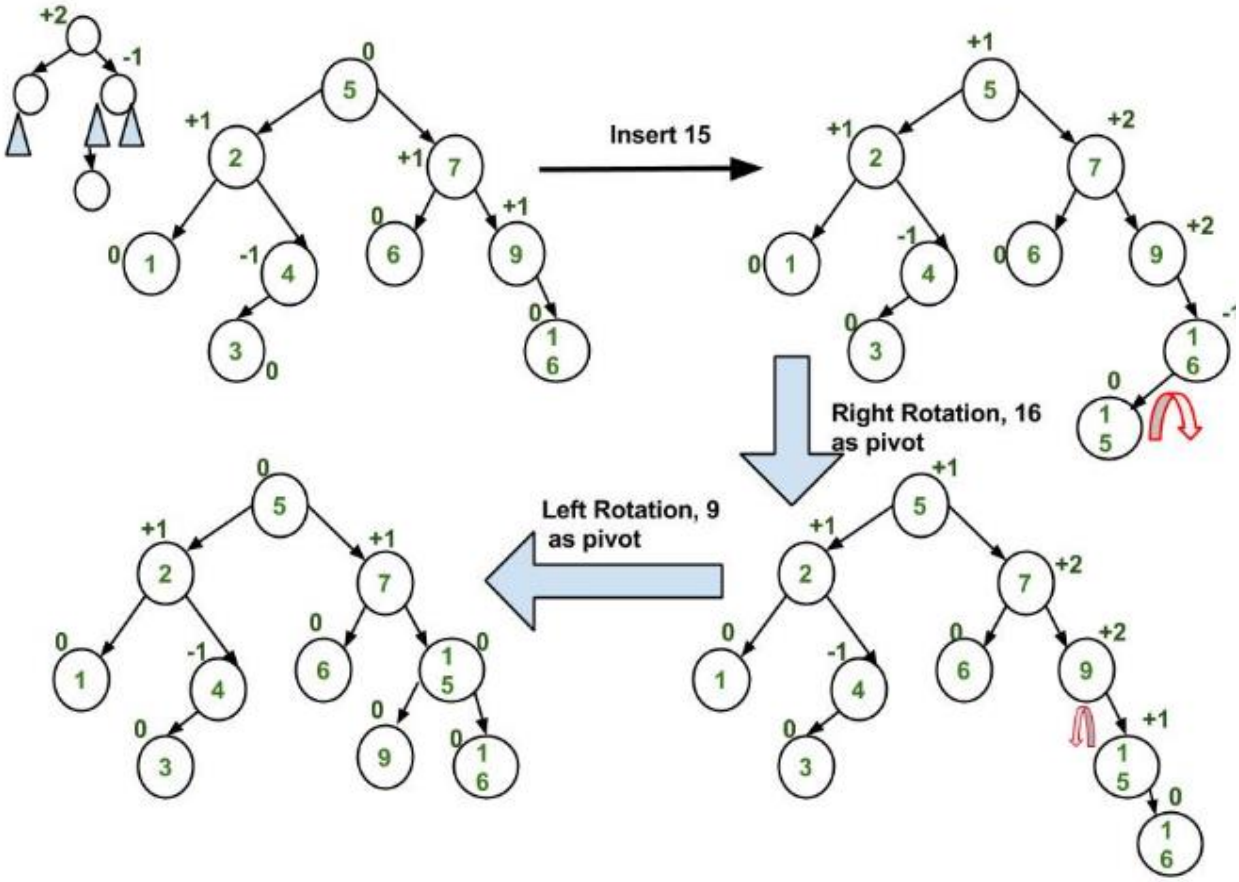
Illustration of Insertion at AVL Tree




Insert Node with value 3
Rotating Right, node with value 10 as pivot
Insert 45
Left rotate, node with value 30 Taken as pivot


Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot


Insert 15
Right Rotation, 16
as pivot
Left Rotation, 9
as pivot



Approach

The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

Follow the steps mentioned below to implement the idea:

- Perform the normal BST insertion.
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor **(left subtree height – right subtree height)** of the current node.

- If the balance factor is greater than **1,** then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root.

Below is the implementation of the above approach:

```
// C++ program to insert a node in AVL tree
#include <bits/stdc++.h>
using namespace std;

// An AVL tree node
struct Node {
    int key;
    Node *left;
    Node *right;
    int height;

    Node(int k) {
        key = k;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

// A utility function to
// get the height of the tree
int height(Node *N) {
    if (N == nullptr)
        return 0;
    return N->height;
}

// A utility function to right
```

```
// rotate subtree rooted with y
Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = 1 + max(height(y->left),
                        height(y->right));
    x->height = 1 + max(height(x->left),
                        height(x->right));

    // Return new root
    return x;
}

// A utility function to left rotate
// subtree rooted with x
Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = 1 + max(height(x->left),
                        height(x->right));
    y->height = 1 + max(height(y->left),
                        height(y->right));

    // Return new root
    return y;
}

// Get balance factor of node N
```

```
int getBalance(Node *N) {
    if (N == nullptr)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in
// the subtree rooted with node
Node* insert(Node* node, int key) {

    // Perform the normal BST insertion
    if (node == nullptr)
        return new Node(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    // Update height of this ancestor node
    node->height = 1 + max(height(node->left),
                           height(node->right));

    // Get the balance factor of this ancestor node
    int balance = getBalance(node);

    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
```

```
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the (unchanged) node pointer
    return node;
}

// A utility function to print
// preorder traversal of the tree
void preOrder(Node *root) {
    if (root != nullptr) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main() {
    Node *root = nullptr;

    // Constructing tree given in the above figure
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
              30
             /  \
```

```
        20     40
       /  \     \
      10   25    50
   */
   cout << "Preorder traversal : \n";
   preOrder(root);

   return 0;
}
```

Output

```
Preorder traversal :
30 20 10 25 40 50
```

Complexity Analysis

Time Complexity: O(log(n)), For Insertion
Auxiliary Space: O(Log n) for recursion call stack as we have written a recursive method to insert

The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of the AVL insert remains the same as the BST insert which is O(h) where h is the height of the tree. Since the AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

Deletion in an AVL Tree

We have discussed AVL insertion in the previous section . In this section, we will follow a similar approach for deletion.

Steps to follow for deletion.
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1. Left Rotation

2. Right Rotation

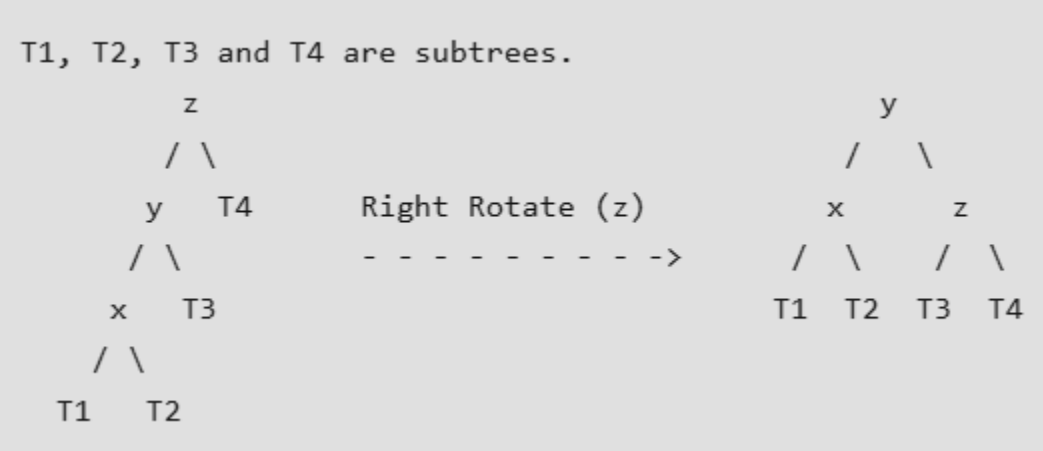
```
T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)
               y                               x
              / \     Right Rotation          /  \
             x   T3   - - - - - - - >        T1   y
            / \       < - - - - - - -            / \
           T1  T2     Left Rotation            T2  T3
Keys in both of the above trees follow the following order
      keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

Let w be the node to be deleted

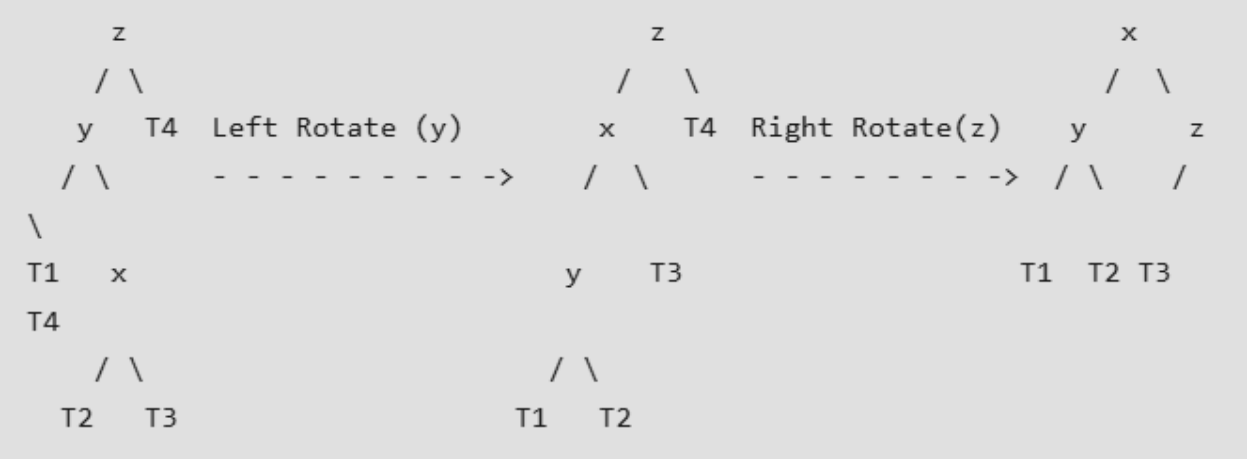
1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
    1. y is left child of z and x is left child of y (Left Left Case)
    2. y is left child of z and x is right child of y (Left Right Case)
    3. y is right child of z and x is right child of y (Right Right Case)
    4. y is right child of z and x is left child of y (Right Left Case)

    Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well

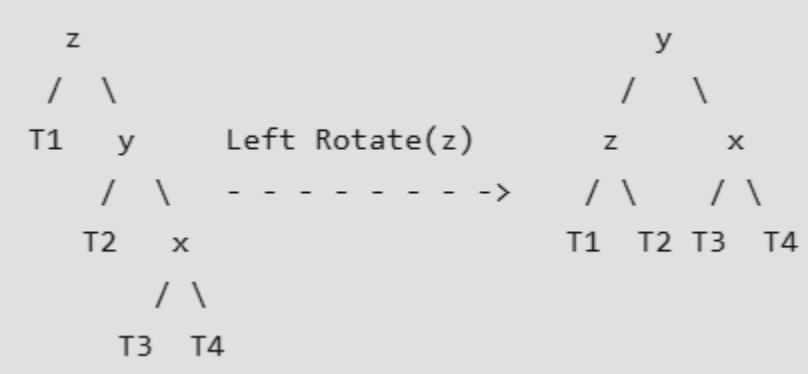
a) Left Left Case



b) Left Right Case



c) Right Right Case

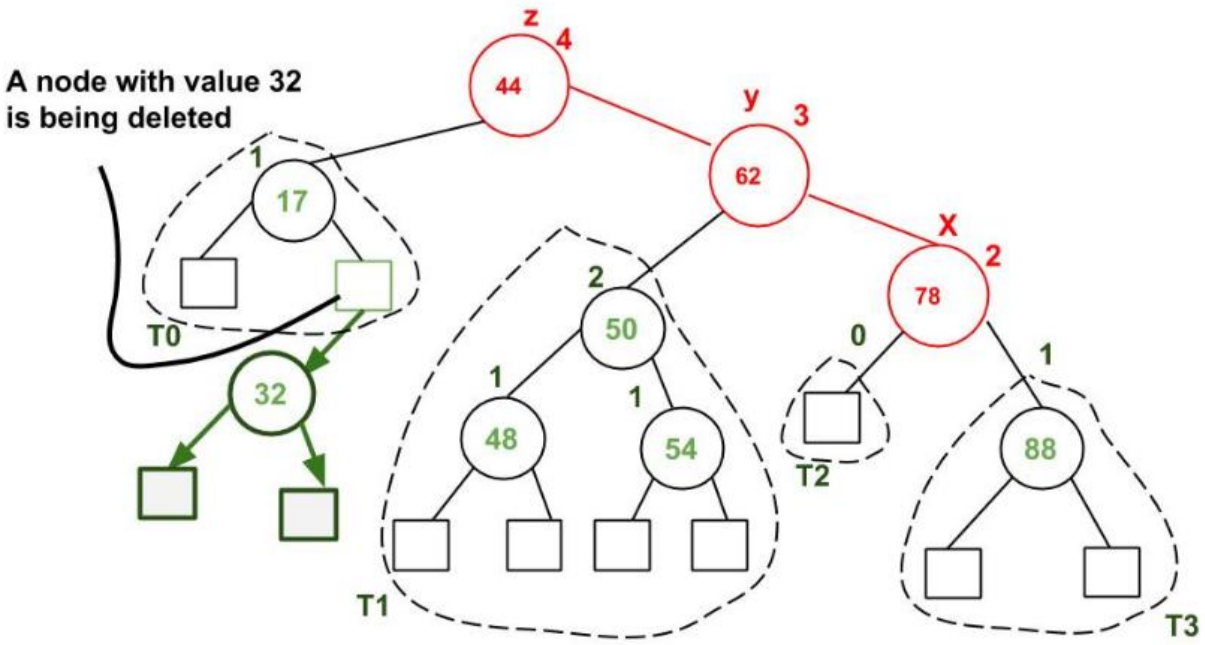


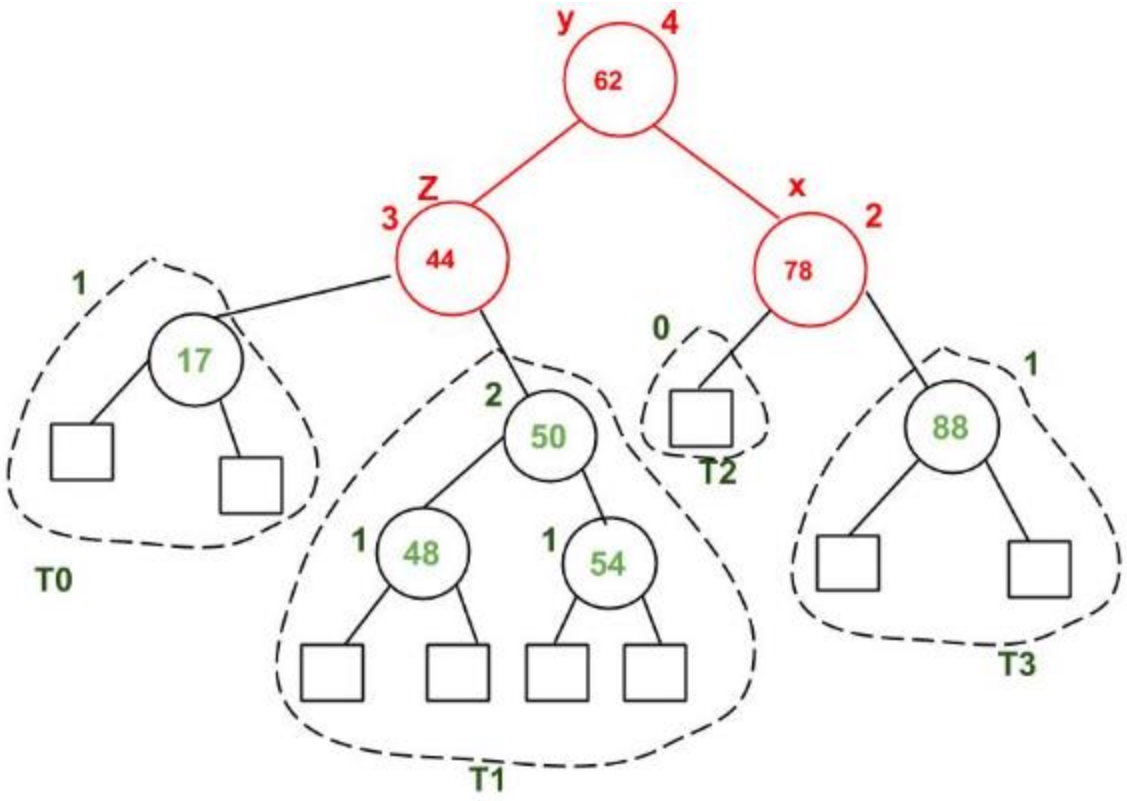
d) Right Left Case

```
   z                                z                              x
  / \                              / \                            /  \
T1   y   Right Rotate (y)       T1   x      Left Rotate(z)      z      y
    / \  - - - - - - - - - ->      /  \   - - - - - - - - ->  / \    / \
   x   T4                        T2   y                     T1  T2  T3  T4
  / \                                /  \
T2   T3                            T3   T4
```

Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

Example of AVL Tree Deletion





A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

Following is the implementation for AVL Tree Deletion. The following implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer (or reference) to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

1. Perform the normal BST deletion.
2. The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3. Get the balance factor (left subtree height – right subtree height) of the current node.

4. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

```
#include <bits/stdc++.h>
using namespace std;

// An AVL tree node
class Node {
public:
   int key;
   Node *left;
   Node *right;
   int height;

   Node(int k) {
      key = k;
      left = nullptr;
      right = nullptr;
      height = 1;
   }
};

// A utility function to get the height
// of the tree
int height(Node *N) {
   if (N == nullptr)
      return 0;
   return N->height;
}
```

```
// A utility function to right rotate
// subtree rooted with y
Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = height(y->left) +
            height(y->right) + 1;
    x->height = height(x->left) +
            height(x->right) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate
// subtree rooted with x
Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = height(x->left) +
            height(x->right) + 1;
    y->height = height(y->left) +
            height(y->right) + 1;

    // Return new root
    return y;
}
```

```
// Get Balance factor of node N
int getBalance(Node *N) {
    if (N == nullptr)
        return 0;
    return height(N->left) -
           height(N->right);
}

Node* insert(Node* node, int key) {
    // 1. Perform the normal BST rotation
    if (node == nullptr)
        return new Node(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    // 2. Update height of this ancestor node
    node->height = height(node->left) +
                   height(node->right) + 1;

    // 3. Get the balance factor of this
    // ancestor node to check whether this
    // node became unbalanced
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
```

```
    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // return the (unchanged) node pointer
    return node;
}

// Given a non-empty binary search tree,
// return the node with minimum key value
// found in that tree. Note that the entire
// tree does not need to be searched.
Node * minValueNode(Node* node) {
    Node* current = node;

    // loop down to find the leftmost leaf
    while (current->left != nullptr)
        current = current->left;

    return current;
}

// Recursive function to delete a node with
// given key from subtree with given root.
// It returns root of the modified subtree.
Node* deleteNode(Node* root, int key) {
    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == nullptr)
        return root;

    // If the key to be deleted is smaller
```

```
    // than the root's key, then it lies in
    // left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater
    // than the root's key, then it lies in
    // right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then
    // this is the node to be deleted
    else {
        // node with only one child or no child
        if ((root->left == nullptr) ||
            (root->right == nullptr)) {
            Node *temp = root->left ?
                        root->left : root->right;

            // No child case
            if (temp == nullptr) {
                temp = root;
                root = nullptr;
            } else // One child case
                *root = *temp; // Copy the contents of
                            // the non-empty child
            free(temp);
        } else {
            // node with two children: Get the
            // inorder successor (smallest in
            // the right subtree)
            Node* temp = minValueNode(root->right);

            // Copy the inorder successor's
            // data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
```

```
    }
}

// If the tree had only one node then return
if (root == nullptr)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = height(root->left) +
               height(root->right) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS
// NODE (to check whether this node
// became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then
// there are 4 cases

// Left Left Case
if (balance > 1 &&
    getBalance(root->left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 &&
    getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 &&
    getBalance(root->right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 &&
    getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
```

```
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder
// traversal of the tree.
void preOrder(Node *root) {
    if (root != nullptr) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main() {
    Node *root = nullptr;

    // Constructing tree given in the
    // above figure
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    cout << "Preorder traversal of the "
         "constructed AVL tree is \n";
    preOrder(root);

    root = deleteNode(root, 10);

    cout << "\nPreorder traversal after"
         " deletion of 10 \n";
```

```
    preOrder(root);

    return 0;
}
```

Output

```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11
```

Time Complexity:

The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Log n). **Auxiliary Space**: O(Log n) for recursion call stack as we have written a recursive method to delete.

B-Tree

The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search

and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

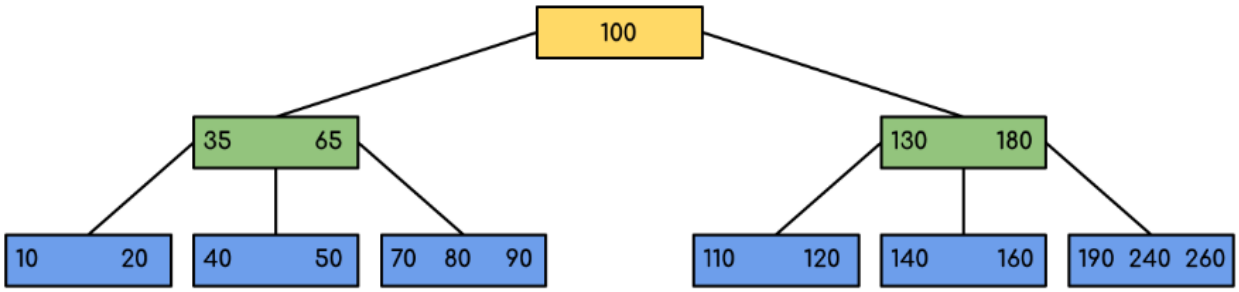
Note: "n" is the total number of elements in the B-tree

Properties of B-Tree:

- All leaves are at the same level.
- B-Tree is defined by the term minimum degree '**t**'. The value of '**t**' depends upon disk block size.
- Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
- All nodes (including root) may contain at most (**2*t – 1**) keys.
- Number of children of a node is equal to the number of keys in it plus **1**.
- All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O(log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.

Example : B- Tree

- Following is an example of a B-Tree of minimum order 5
  Note: that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf have no empty sub-tree and have keys one less than the number of their children.

Interesting Facts about B-Trees:

- *The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is:*
  $h_{min} = \lceil \log_m(n+1) \rceil - 1$
- *The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have is:*
  $h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor$
  and
  $t = \lceil \frac{m}{2} \rceil$

Traversal in B-Tree:

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children and keys. In the end, recursively print the rightmost child.

Search Operation in B-Tree:

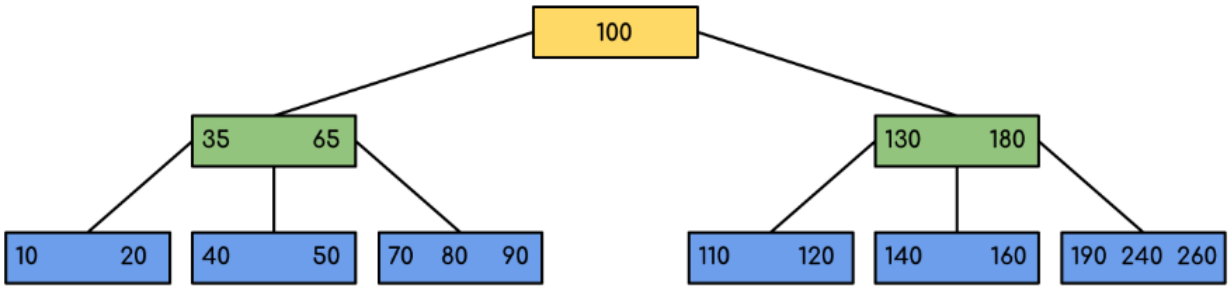
Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

- Start from the root and recursively traverse down.
- For every visited non-leaf node,
    - If the node has the key, we simply return the node.
    - Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.
- Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.
- Algorithm for Searching an Element in a B-Tree:-

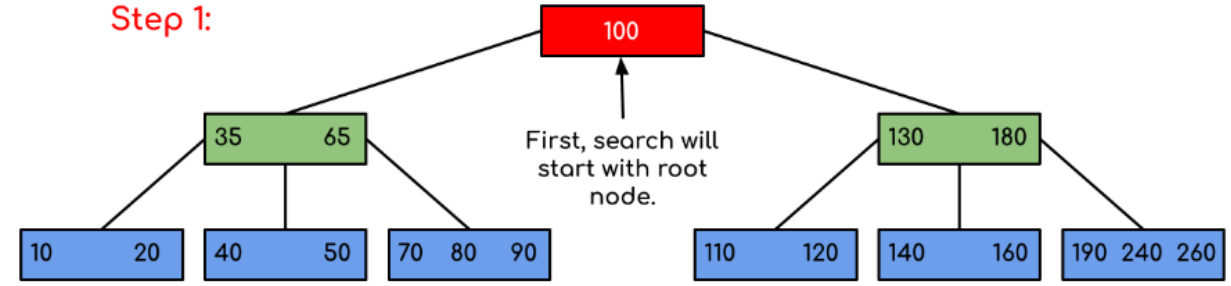
```
struct Node {
    int n;
    int key[MAX_KEYS];
    Node* child[MAX_CHILDREN];
    bool leaf;
};

Node* BtreeSearch(Node* x, int k) {
    int i = 0;
    while (i < x->n && k > x->key[i]) {
            i++;
    }
    if (i < x->n && k == x->key[i]) {
            return x;
    }
    if (x->leaf) {
            return nullptr;
    }
    return BtreeSearch(x->child[i], k);
}
```

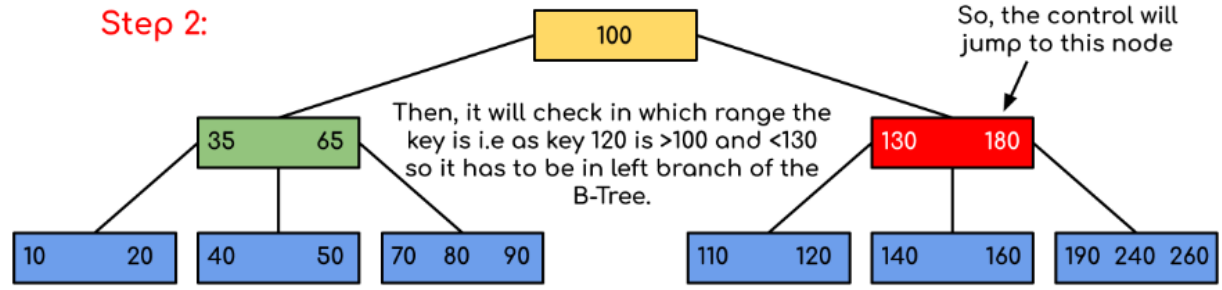
Example:- Search in B- Tree

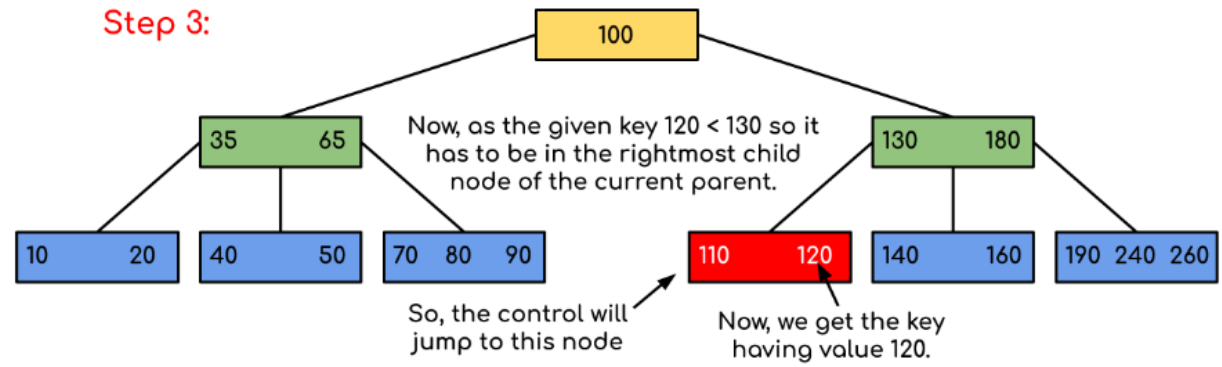
Input: *Search 120 in the given B-Tree.*



Solution







In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as 90 < 100 so it'll go to the left subtree automatically, and therefore the control flow will go similarly as shown within the above example.

Below is the implementation of the above approach:

```
// C++ implementation of search() and traverse() methods
#include <iostream>
using namespace std;

// A BTree node
class BTreeNode {
        int* keys; // An array of keys
        int t; // Minimum degree (defines the range for number
                // of keys)
        BTreeNode** C; // An array of child pointers
        int n; // Current number of keys
        bool leaf; // Is true when node is leaf. Otherwise false
public:
        BTreeNode(int _t, bool _leaf); // Constructor

        // A function to traverse all nodes in a subtree rooted
        // with this node
        void traverse();

        // A function to search a key in the subtree rooted with
        // this node.
        BTreeNode*
        search(int k); // returns NULL if k is not present.

        // Make the BTree friend of this so that we can access
        // private members of this class in BTree functions
        friend class BTree;
};
```

```
// A BTree
class BTree {
	BTreeNode* root; // Pointer to root node
	int t; // Minimum degree
public:
	// Constructor (Initializes tree as empty)
	BTree(int _t)
	{
		root = NULL;
		t = _t;
	}

	// function to traverse the tree
	void traverse()
	{
		if (root != NULL)
			root->traverse();
	}

	// function to search a key in this tree
	BTreeNode* search(int k)
	{
		return (root == NULL) ? NULL : root->search(k);
	}
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
	// Copy the given minimum degree and leaf property
	t = _t;
	leaf = _leaf;

	// Allocate memory for maximum number of possible keys
	// and child pointers
	keys = new int[2 * t - 1];
	C = new BTreeNode*[2 * t];

	// Initialize the number of keys as 0
```

```
        n = 0;
}

// Function to traverse all nodes in a subtree rooted with
// this node
void BTreeNode::traverse()
{
        // There are n keys and n+1 children, traverse through n
        // keys and first n children
        int i;
        for (i = 0; i < n; i++) {
                // If this is not leaf, then before printing key[i],
                // traverse the subtree rooted with child C[i].
                if (leaf == false)
                        C[i]->traverse();
                cout << " " << keys[i];
        }

        // Print the subtree rooted with last child
        if (leaf == false)
                C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode* BTreeNode::search(int k)
{
        // Find the first key greater than or equal to k
        int i = 0;
        while (i < n && k > keys[i])
                i++;

        // If the found key is equal to k, return this node
        if (keys[i] == k)
                return this;

        // If the key is not found here and this is a leaf node
        if (leaf == true)
                return NULL;

        // Go to the appropriate child
```

```
        return C[i]->search(k);
}
```

Note: The above code doesn't contain the driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree, second is to define by order. We have followed the minimum degree convention and will be following the same in coming posts on B-Tree. The variable names used in the above program are also kept the same

Applications of B-Trees:

- 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.

Advantages of B-Trees:

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

Disadvantages of B-Trees:

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.

- Slow in comparison to other data structures.

Insert Operation in B-Tree

Delete Operation in B-Tree

B+ Tree

B + Tree is a variation of the B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. In a B+ tree structure of a leaf node differs from the structure of internal nodes. The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record). The leaf nodes of the B+ tree are linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree are used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

Features of B+ Trees

- **Balanced:** B+ Trees are self-balancing, which means that as data is added or removed from the tree, it automatically adjusts itself to maintain a balanced structure. This ensures that the search time remains relatively constant, regardless of the size of the tree.
- **Multi-level:** B+ Trees are multi-level data structures, with a root node at the top and one or more levels of internal nodes below it. The leaf nodes at the bottom level contain the actual data.
- **Ordered:** B+ Trees maintain the order of the keys in the tree, which makes it easy to perform range queries and other operations that require sorted data.
- **Fan-out:** B+ Trees have a high fan-out, which means that each node can have many child nodes. This reduces the height of the tree and increases the efficiency of searching and indexing operations.
- **Cache-friendly:** B+ Trees are designed to be cache-friendly, which means that they can take advantage of the caching mechanisms in modern computer architectures to improve performance.

- **Disk-oriented:** B+ Trees are often used for disk-based storage systems because they are efficient at storing and retrieving data from disk.

Why Use B+ Tree?

- B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.
- B+ Trees are a good choice for database systems and applications needing quick data retrieval because of their balanced structure, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

Difference Between B+ Tree and B Tree

Some differences between B+ Tree and B Tree are stated below.

Parameters	B+ Tree	B Tree
Structure	Separate leaf nodes for data storage and internal nodes for indexing	Nodes store both keys and data values
Leaf Nodes	Leaf nodes form a linked list for efficient range-based queries	Leaf nodes do not form a linked list
Order	Higher order (more keys)	Lower order (fewer keys)
Key Duplication	Typically allows key duplication in leaf nodes	Usually does not allow key duplication

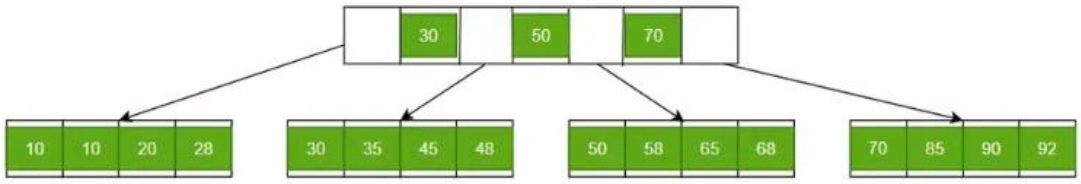
Disk Access	Better disk access due to sequential reads in a linked list structure	More disk I/O due to non-sequential reads in internal nodes
Applications	Database systems, file systems, where range queries are common	In-memory data structures, databases, general-purpose use
Performance	Better performance for range queries and bulk data retrieval	Balanced performance for search, insert, and delete operations
Memory Usage	Requires more memory for internal nodes	Requires less memory as keys and values are stored in the same node

Implementation of B+ Tree

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of the B-tree used for indexing, however, is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record. B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of the leaf nodes of a B+ tree is quite different from the structure of the internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record. From the above discussion, it is apparent that a B+ tree, unlike a B-tree, has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes.

Structure of B+ Trees



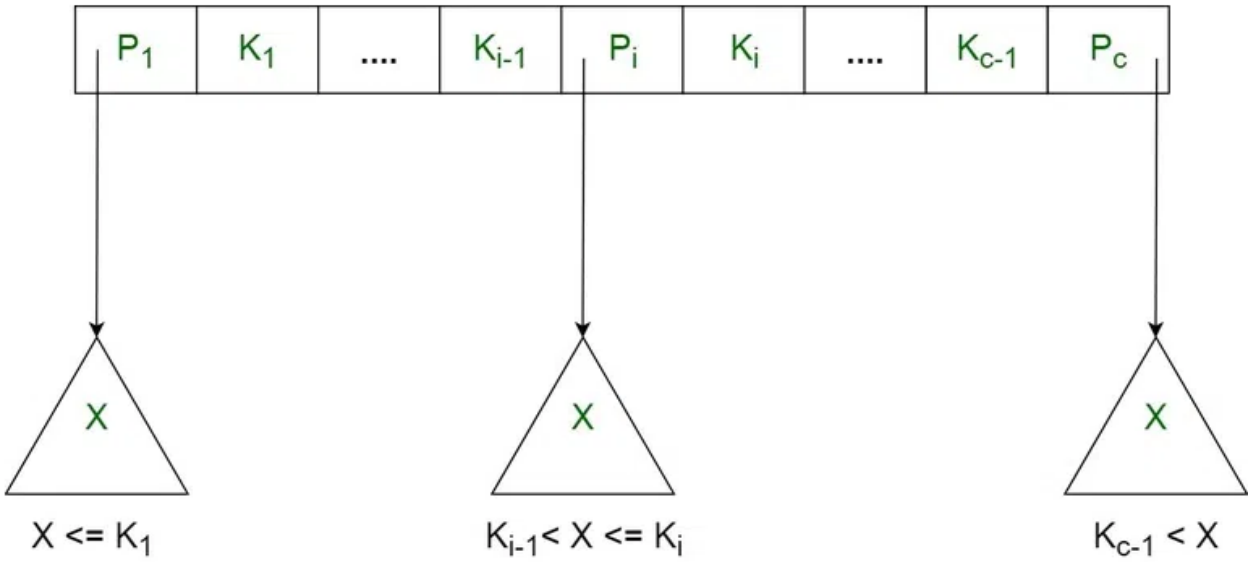
B+ Trees contain two types of nodes:

- **Internal Nodes:** Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
- **Leaf Nodes:** Leaf Nodes are the nodes that have n pointers.

The Structure of the Internal Nodes of a B+ Tree of Order 'a' is as Follows

- Each internal node is of the form: <P1, K1, P2, K2, ….., Pc-1, Kc-1, Pc> where c <= a and each **Pi is a tree pointer (i.e points to another node of the tree)** and, each **Ki is a key-value** (see diagram-I for reference).
- Every internal node has : K1 < K2 < …. < Kc-1
- For each search field value 'X' in the sub-tree pointed at by Pi, the following condition holds: Ki-1 < X <= Ki, for 1 < I < c and, Ki-1 < X, for i = c (See diagram I for reference)
- Each internal node has at most 'aa tree pointers.
- The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.

- If an internal node has ‘c’ pointers, c <= a, then it has ‘c – 1’ key values.



Structure of internal node

The Structure of the Leaf Nodes of a B+ Tree of Order ‘b’ is as Follows

- Each leaf node is of the form: <<K1, D1>, <K2, D2>, ….., <Kc-1, Dc-1>, Pnext> where c <= b and each **Di is a data pointer (i.e points to actual record in the disk whose key value is Ki or to a disk file block containing that record)** and, each **Ki is a key value** and, **Pnext points to next leaf node in the B+ tree** (see diagram II for reference).
- Every leaf node has : K1 < K2 < …. < Kc-1, c <= b
- Each leaf node has at least \ceil(b/2) values.
- All leaf nodes are at the same level.

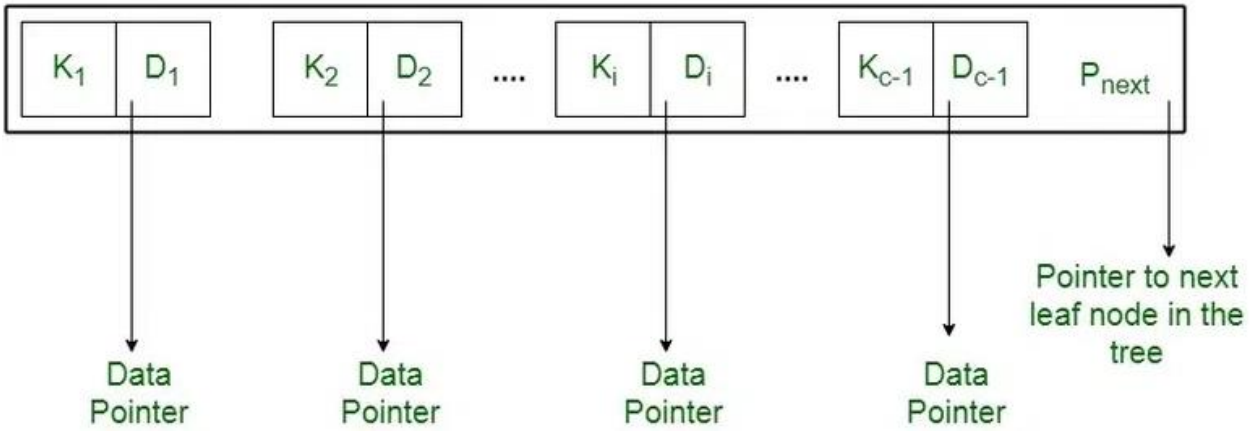
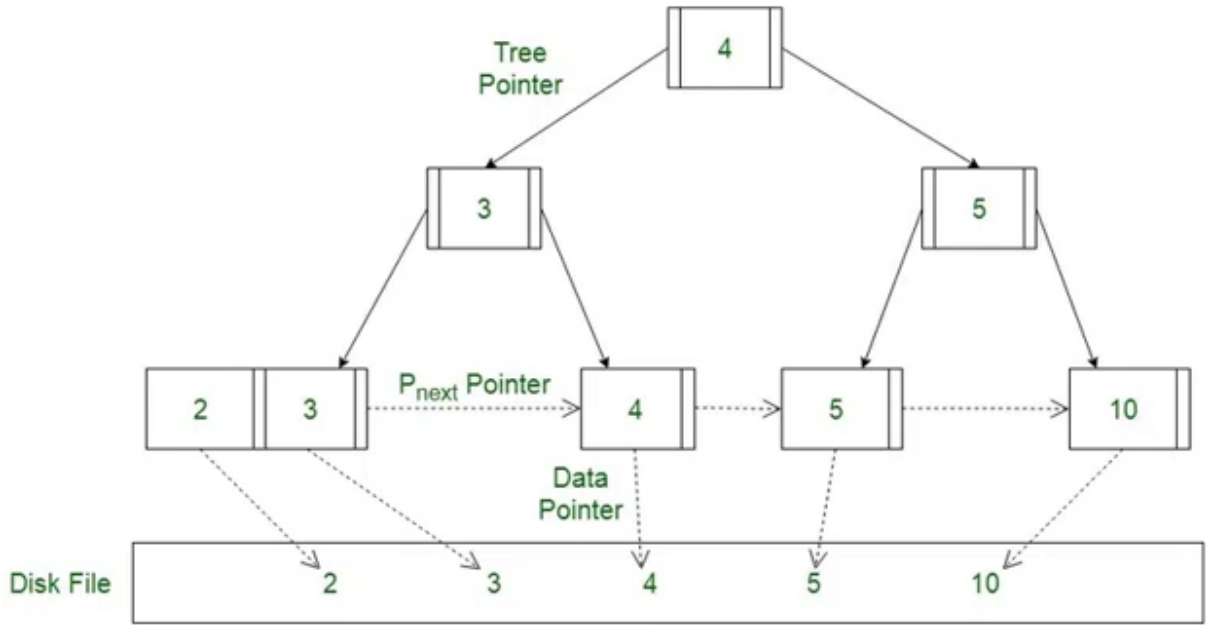
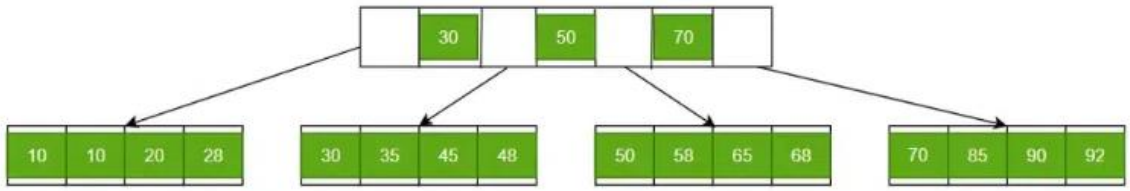


Diagram-II Using the Pnext pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.



Searching a Record in B+ Trees



Let us suppose we have to find 58 in the B+ Tree. We will start by fetching from the root node then we will move to the leaf node, which might contain a record of 58. In the image given above, we will get 58 between 50 and 70. Therefore, we will we are getting a leaf node in the third leaf node and get 58 there. If we are unable to find that node, we will return that 'record not founded' message.

Insertion in B+ Trees

Insertion in B+ Trees is done via the following steps.

- Every element in the tree has to be inserted into a leaf node. Therefore, it is necessary to go to a proper leaf node.
- Insert the key into the leaf node in increasing order if there is no overflow.

Deletion in B+Trees

Deletion in B+ Trees is just not deletion but it is a combined process of Searching, Deletion, and Balancing. In the last step of the Deletion Process, it is mandatory to balance the B+ Trees, otherwise, it fails in the property of B+ Trees.

Advantages of B+Trees

- A B+ tree with 'l' levels can store more entries in its internal nodes compared to a B-tree having the same 'l' levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and the presence of Pnext pointers imply that the B+ trees is very quick and efficient in accessing records from disks.
- Data stored in a B+ tree can be accessed both sequentially and directly.
- It takes an equal number of disk accesses to fetch records.
- B+trees have redundant search keys, and storing search keys repeatedly is not possible.

Disadvantages of B+ Trees

- The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.

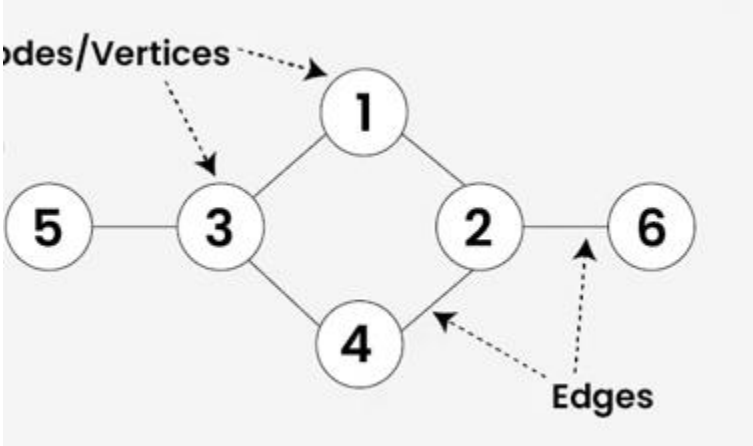
Application of B+ Trees

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing

Chapter 3
Graph

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E).**

Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. They are particularly useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.



Components of a Graph:

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

Operations on Graphs:

Basic Operations:

- Insertion of Nodes/Edges in the graph – Insert a node into the graph.

- Deletion of Nodes/Edges in the graph – Delete a node from the graph.
- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph**.**
- Shortest Paths : From a source to a destination, a source to all other nodes and between all pairs.
- Minimum Spanning Tee : In a weighted, connected undirected graph, finding the minimum weight edges to connect all.

Applications of Graph:

Following are the real-life applications:

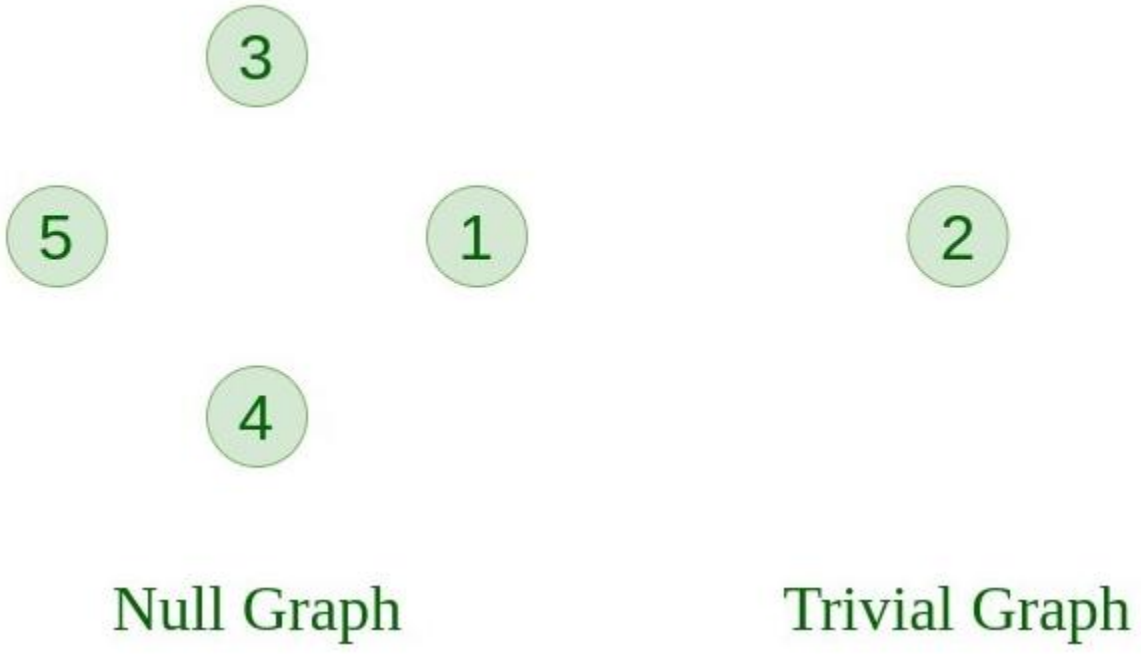
- If we recall all the previous data structures that we have studied like array, linked list, tree, etc. All these had some restrictions on structure (mostly linear and tree hierarchical which means no loops). Graph allows random connections between nodes which is useful in many real world problems where do have restrictions of previous data structures.
- Used heavily in social networks. Everyone on the network is a vertex (or node) of the graph and if connected, then there is an edge. Now imagine all the features that you see, mutual friends, people that follow you, etc can seen as graph problems.
- **Neural Networks:** Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.
- **Compilers:** Graph Data Structure is used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
- **Robot planning:** Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.
- **In GPS.** The problems like finding the closest route, closest petrol pumps, etc are all soled using graph problems.
- For optimizing the cost of connecting all locations of a network. For example, minimizing wire length in a wired network to make sure all devices are connected is a standard Graph problem called Minimum Spanning Tree.

- Can be used to represent the interactions between players on a team, such as passes, shots, and tackles. Analyzing these interactions can provide insights into team dynamics and areas for improvement.
- Can be used to represent the topology of computer networks, such as the connections between routers and switches.
- Graphs are used to represent the connections between different places in a transportation network, such as roads and airports.
- Dependencies in a software project (or any other type of project) can be seen as graph and generating a sequence to solve all tasks before dependents is a standard graph topological sorting algorithm.

Types Of Graphs in Data Structure and Algorithms

1. Null Graph
A graph is known as a null graph if there are no edges in the graph.

2. Trivial Graph
Graph having only a single vertex, it is also the smallest graph possible.

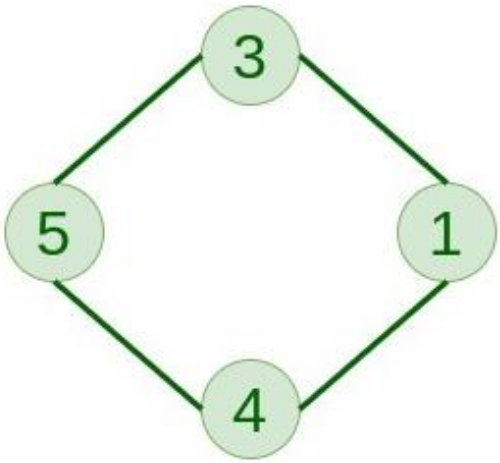


3. Undirected Graph

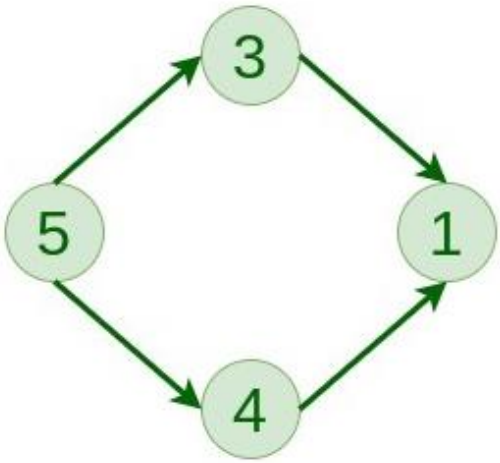
A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

4. Directed Graph

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.



Undirected Graph



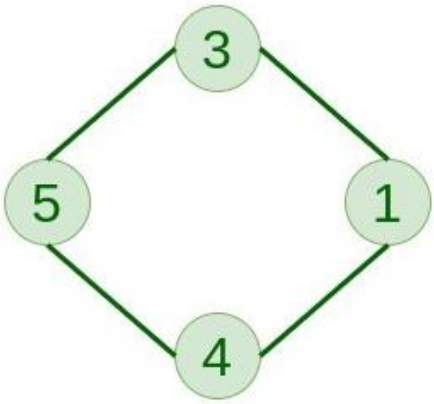
Directed Graph

5. Connected Graph

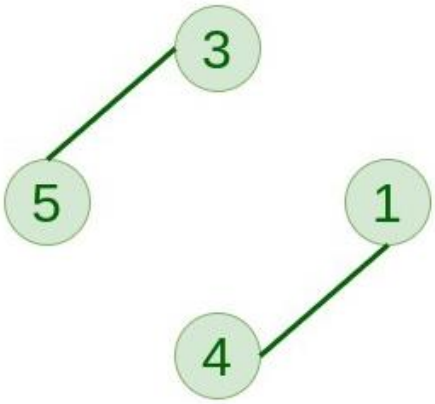
The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. Disconnected Graph

The graph in which at least one node is not reachable from a node is known as a disconnected graph.



Connected Graph



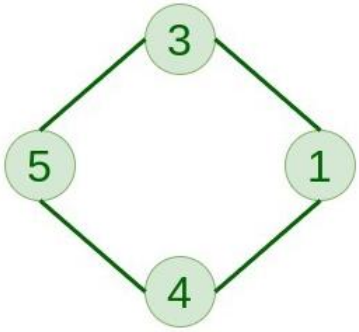
Disconnected Graph

7. Regular Graph

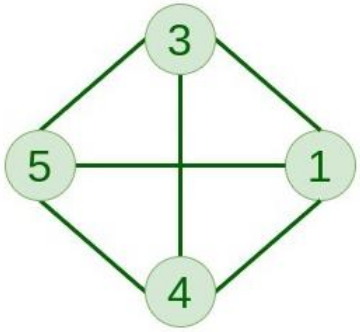
The graph in which the degree of every vertex is equal to K is called K regular graph.

8. Complete Graph

The graph in which from each node there is an edge to each other node.



2-Regular



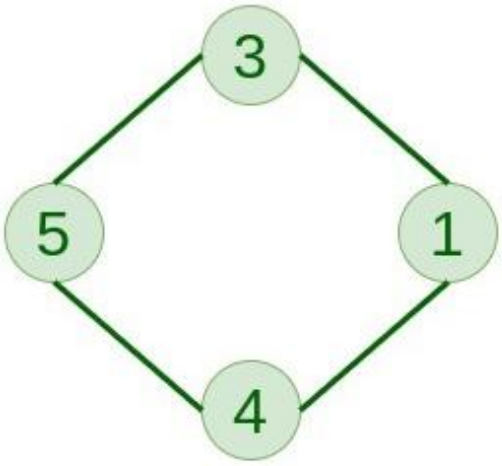
Complete Graph

9. Cycle Graph

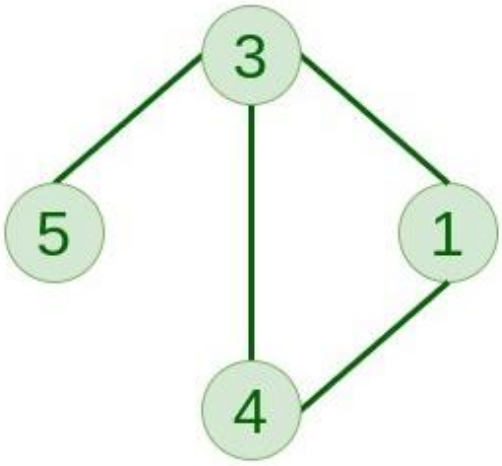
The graph in which the graph is a cycle in itself, the minimum value of degree of each vertex is 2.

10. Cyclic Graph

A graph containing at least one cycle is known as a Cyclic graph.



Cycle Graph



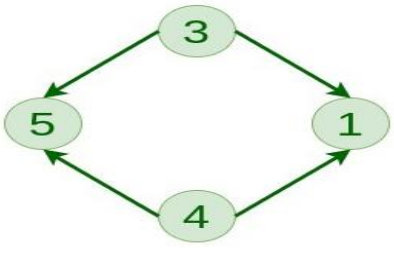
Cyclic Graph

11. Directed Acyclic Graph

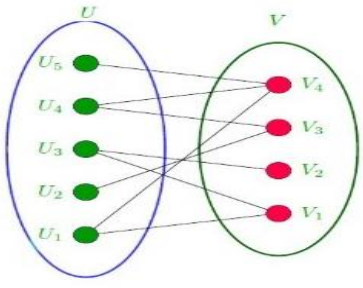
A Directed Graph that does not contain any cycle.

12. Bipartite Graph

A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



Directed Acyclic Graph



Bipartite Graph

13.Weighted Graph

- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

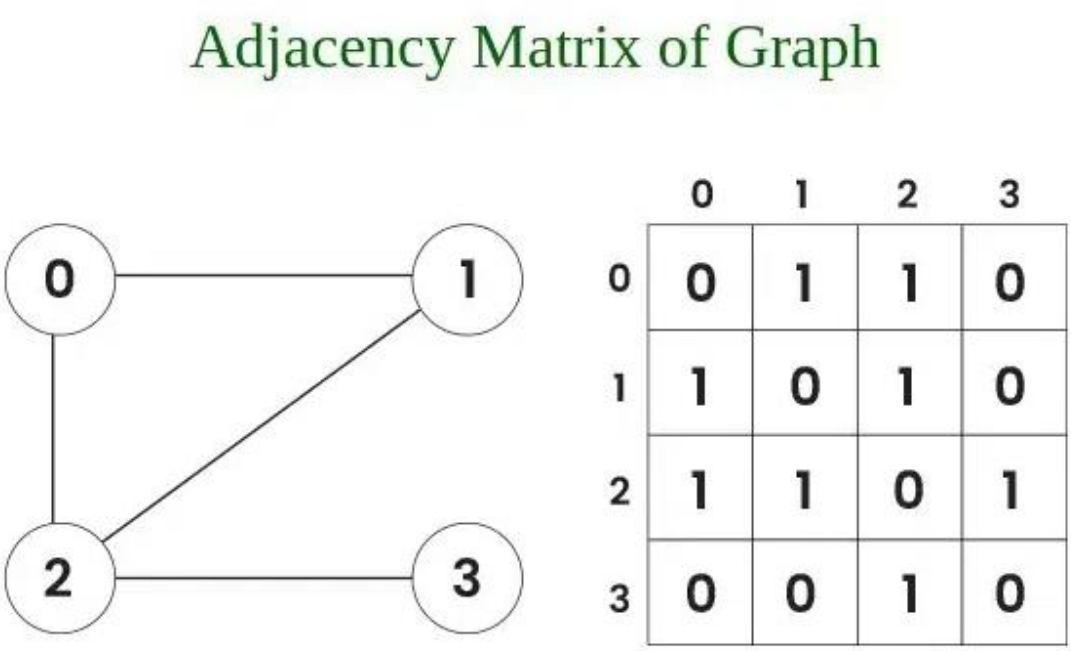
Representation of Graph Data Structure:

There are multiple ways to store a graph: The following are the most common representations.

- Adjacency Matrix
- Adjacency List

Adjacency Matrix Representation of Graph Data Structure :-

n this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.



Below is the implementation of Graph Data Structure represented using Adjacency Matrix:

```
// C++ program to demonstrate Adjacency Matrix
// representation of undirected and unweighted graph
#include <bits/stdc++.h>
using namespace std;

void addEdge(vector<vector<int>> &mat, int i, int j)
{
   mat[i][j] = 1;
   mat[j][i] = 1; // Since the graph is undirected
}

void displayMatrix(vector<vector<int>> &mat)
{
   int V = mat.size();
   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < V; j++)
         cout << mat[i][j] << " ";
      cout << endl;
   }
}

int main()
{

   // Create a graph with 4 vertices and no edges
   // Note that all values are initialized as 0
   int V = 4;
   vector<vector<int>> mat(V, vector<int>(V, 0));

   // Now add edges one by one
   addEdge(mat, 0, 1);
   addEdge(mat, 0, 2);
   addEdge(mat, 1, 2);
   addEdge(mat, 2, 3);

   /* Alternatively we can also create using below
      code if we know all edges in advacem

    vector<vector<int>> mat = {{ 0, 1, 0, 0 },
```

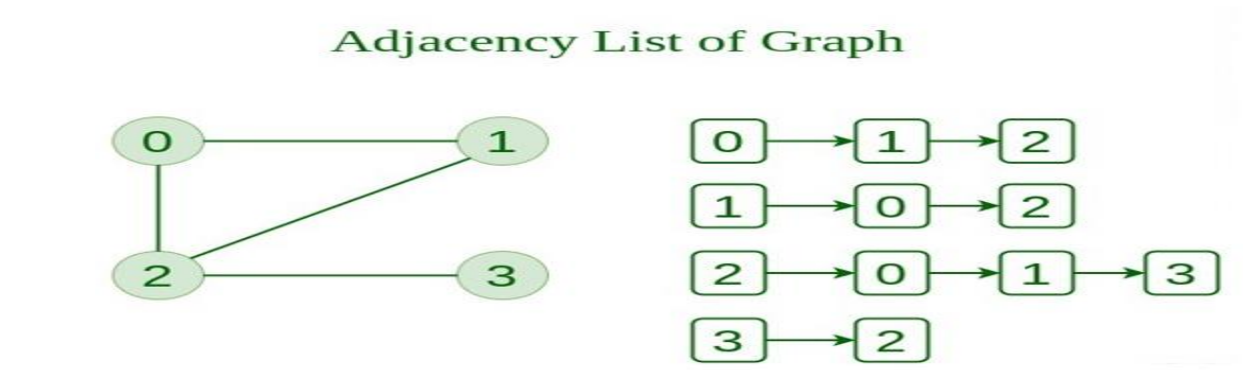
```
                    { 1, 0, 1, 0 },
                    { 0, 1, 0, 1 },
                    { 0, 0, 1, 0 } }; */

    cout << "Adjacency Matrix Representation" << endl;
    displayMatrix(mat);

    return 0;
}
```

Output

```
Adjacency Matrix Representation
  0 1 1 0
  1 0 1 0
  1 1 0 1
  0 0 1 0
```

Adjacency List Representation of Graph :-

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Below is the implementation of Graph Data Structure represented using Adjacency List:

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

// Function to add an edge between two vertices
void addEdge(vector<vector<int>>& adj, int i, int j) {
    adj[i].push_back(j);
    adj[j].push_back(i); // Undirected
```

```
}

// Function to display the adjacency list
void displayAdjList(const vector<vector<int>>& adj) {
    for (int i = 0; i < adj.size(); i++) {
        cout << i << ": "; // Print the vertex
        for (int j : adj[i]) {
            cout << j << " "; // Print its adjacent
        }
        cout << endl;
    }
}

// Main function
int main() {
    // Create a graph with 4 vertices and no edges
    int V = 4;
    vector<vector<int>> adj(V);

    // Now add edges one by one
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 1, 2);
    addEdge(adj, 2, 3);

    cout << "Adjacency List Representation:" << endl;
    displayAdjList(adj);

    return 0;
}
```

Output

```
Adjacency List Representation:
0: 1 2
1: 0 2
2: 0 1 3
3: 2
```

Comparison between Adjacency Matrix and Adjacency List

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as Prim's and Dijkstra adjacency matrix is used to have less complexity.

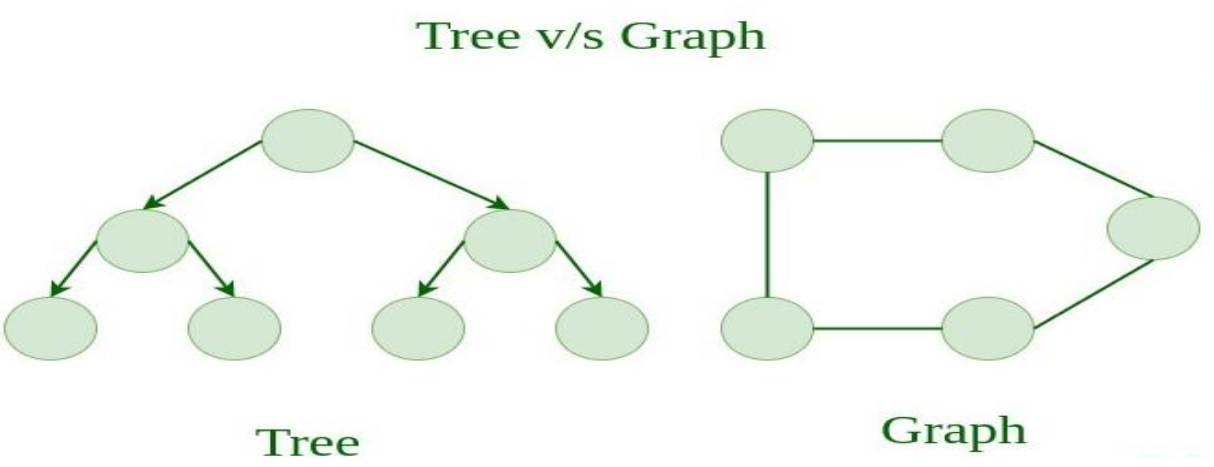
Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing an edge	O(1)	O(N)
Initializing	O(N*N)	O(N)

Basic Operations on Graph Data Structure:

- **Insertion or Deletion of Nodes in the graph**
  - Add and Remove vertex in Adjacency List representation of Graph
  - Add and Remove vertex in Adjacency Matrix representation of Graph
- **Insertion or Deletion of Edges in the graph**
  - Add and Remove Edge in Adjacency List representation of a Graph
  - Add and Remove Edge in Adjacency Matrix representation of a Graph
- **Searching in Graph Data Structure- Search an entity in the graph.**
- **Traversal of Graph Data Structure- Traversing all the nodes in the graph.**

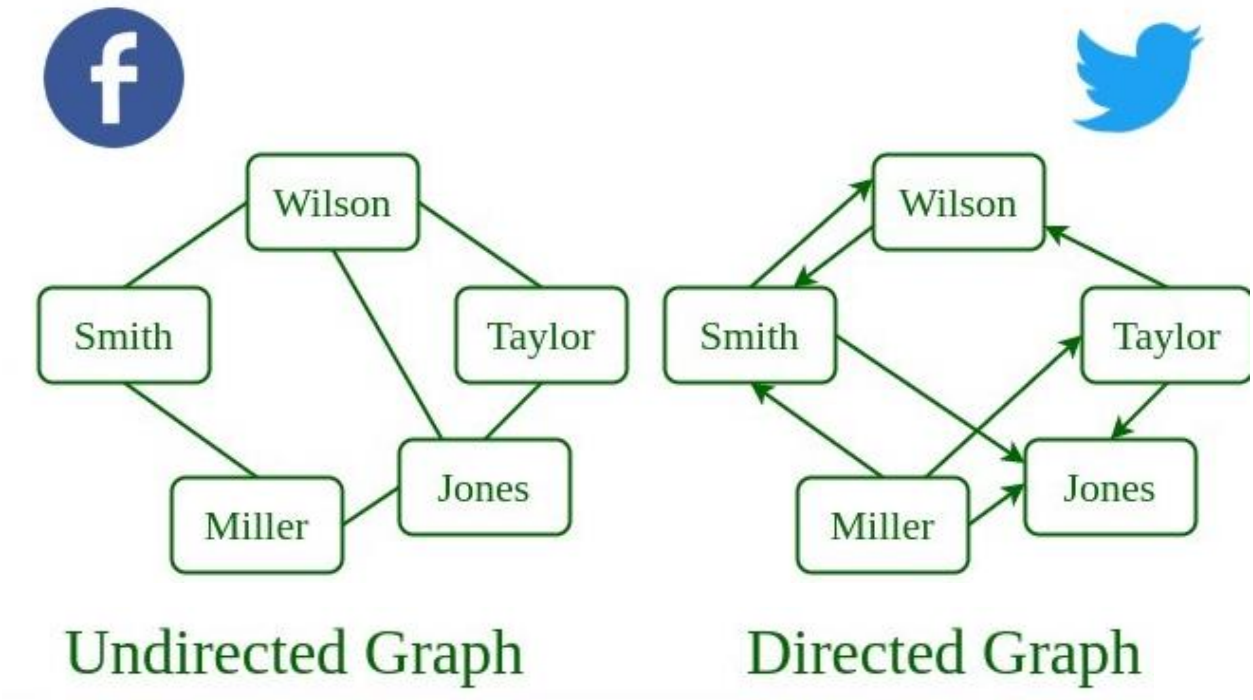
Difference between Tree and Graph:

Tree is a restricted type of Graph Data Structure, just with some more rules. Every tree will always be a graph but not all graphs will be trees. Linked List, Trees, and Heaps all are special cases of graphs.



Real-Life Applications of Graph Data Structure:

Graph Data Structure has numerous real-life applications across various fields. Some of them are listed below:



- If we recall all the previous data structures that we have studied like array, linked list, tree, etc. All these had some restrictions on structure (mostly linear and tree hierarchical which means no loops). Graph allows random connections between nodes which is useful in many real world problems where do have restrictions of previous data structures.
- Used heavily in social networks. Everyone on the network is a vertex (or node) of the graph and if connected, then there is an edge. Now imagine all the features that you see, mutual friends, people that follow you, etc can see as graph problems.
- Used to represent the topology of computer networks, such as the connections between routers and switches.
- Used to represent the connections between different places in a transportation network, such as roads and airports.
- **Neural Networks:** Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.
- **Compilers:** Graph Data Structure is used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation,

and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

- **Robot planning:** Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.
- Dependencies in a software project (or any other type of project) can be seen as graph and generating a sequence to solve all tasks before dependents is a standard graph topological sorting algorithm.
- For optimizing the cost of connecting all locations of a network. For example, minimizing wire length in a wired network to make sure all devices are connected is a standard Graph problem called Minimum Spanning Tree.

Advantages of Graph Data Structure:

- Graph Data Structure used to represent a wide range of relationships as we do not have any restrictions like previous data structures (Tree cannot have loops and have to be hierarchical. Arrays, Linked List, etc are linear)
- They can be used to model and solve a wide range of problems, including pathfinding, data clustering, network analysis, and machine learning.
- Any real world problem where we certain set of items and relations between them can be easily modeled as a graph and a lot of standard graph algorithms like BFS, DFS, Spanning Tree, Shortest Path, Topological Sorting and Strongly Connected
- Graph Data Structure can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

Disadvantages of Graph Data Structure:

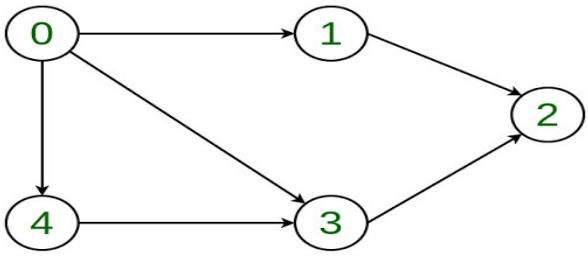
- Graph Data Structure can be complex and difficult to understand, especially for people who are not familiar with graph theory or related algorithms.
- Creating and manipulating graphs can be computationally expensive, especially for very large or complex graphs.
- Graph algorithms can be difficult to design and implement correctly, and can be prone to bugs and errors.
- Graph Data Structure can be difficult to visualize and analyze, especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

Basic Operations on Graph Data Structure

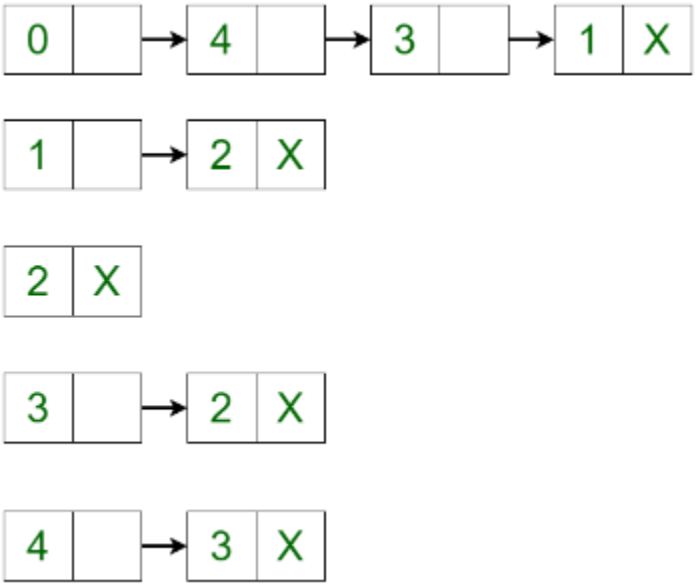
Insertion or Deletion of Nodes in the graph

- **Add and Remove vertex in Adjacency List representation of Graph:-**

adding and removing a vertex is discussed in a given adjacency list representation. Let the Directed Graph be:



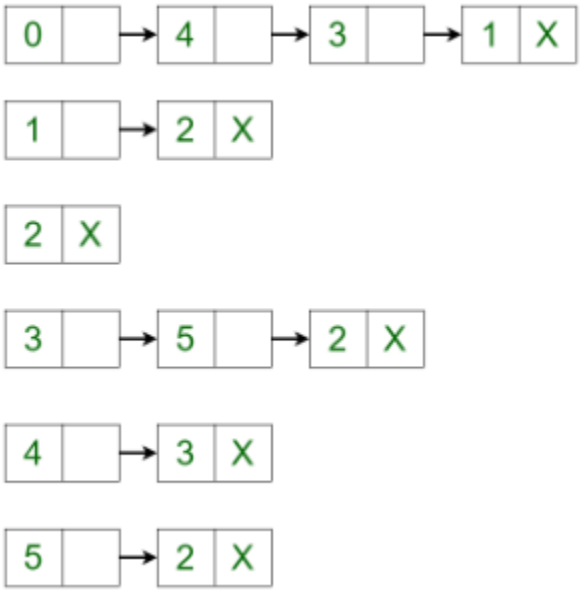
The graph can be represented in the Adjacency List representation as:



It is a Linked List representation where the head of the linked list is a vertex in the graph and all the connected nodes are the vertices to which the first vertex is connected. For example, from the graph, it is clear that vertex **0** is connected to vertex **4**, **3** and **1**. The same is represented in the adjacency list(or Linked List) representation.

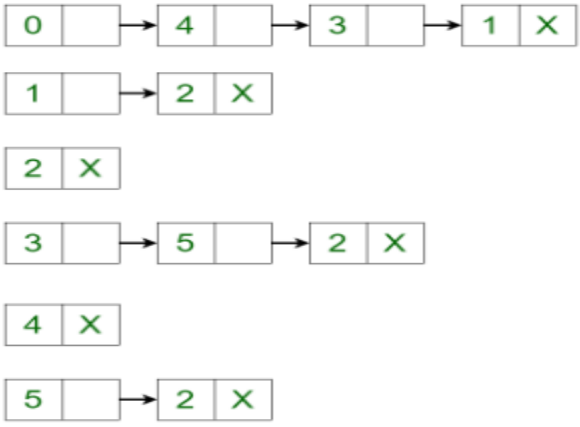
Adding a Vertex in the Adjacency List:

To add a vertex in the graph, the adjacency list can be iterated to the place where the insertion is required and the new node can be created using linked list implementation. For example, if 5 needs to be added between vertex 2 and vertex 3 such that vertex 3 points to vertex 5 and vertex 5 points to vertex 2, then a new edge is created between vertex 5 and vertex 3 and a new edge is created from vertex 5 and vertex 2. After adding the vertex, the adjacency list changes to:



Removing a Vertex in Adjacency List:

To delete a vertex in the graph, iterate through the list of each vertex if an edge is present or not. If the edge is present, then delete the vertex in the same way as delete is performed in a linked list. For example, the adjacency list translates to the below list if vertex 4 is deleted from the list:



Below is the implementation of the above approach:

```
#include <iostream>

using namespace std;

// Node to store adjacency list
class AdjNode {
public:
    int vertex;
    AdjNode* next;
    AdjNode(int data)
    {
        vertex = data;
        next = NULL;
    }
};

// Adjacency List representation
class AdjList {
private:
    int v;
    AdjNode** graph;

public:
    AdjList(int vertices)
    {
        v = vertices;
        graph = new AdjNode*[v];
        for (int i = 0; i < v; ++i)
            graph[i] = NULL;
    }

    // Function to add an edge from a source vertex
    // to a destination vertex
    void addEdge(int source, int destination)
    {
        AdjNode* node = new AdjNode(destination);
        node->next = graph[source];
        graph[source] = node;
    }

    // Function to add a vertex between two vertices
    void addVertex(int vk, int source, int destination)
    {
        addEdge(source, vk);
        addEdge(vk, destination);
    }

    // Function to print the graph
    void printGraph()
```

```
    {
        for (int i = 0; i < v; ++i) {
            cout << i << " ";
            AdjNode* temp = graph[i];
            while (temp != NULL) {
                cout << "-> " << temp->vertex << " ";
                temp = temp->next;
            }
            cout << endl;
        }
    }

    // Function to delete a vertex
    void delVertex(int k)
    {
        // Iterate through all the vertices of the graph
        for (int i = 0; i < v; ++i) {
            AdjNode* temp = graph[i];
            if (i == k) {
                graph[i] = temp->next;
                temp = graph[i];
            }
            // Delete the vertex using linked list concept
            while (temp != NULL) {
                if (temp->vertex == k) {
                    break;
                }
                AdjNode* prev = temp;
                temp = temp->next;
                if (temp == NULL) {
                    continue;
                }
                prev->next = temp->next;
                temp = NULL;
            }
        }
    }
};

int main()
{
    int V = 6;
    AdjList graph(V);
    graph.addEdge(0, 1);
    graph.addEdge(0, 3);
    graph.addEdge(0, 4);
    graph.addEdge(1, 2);
    graph.addEdge(3, 2);
    graph.addEdge(4, 3);

    cout << "Initial adjacency list" << endl;
```

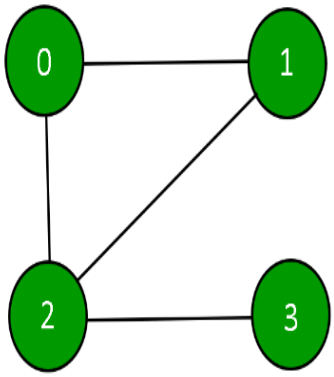
```
    graph.printGraph();

    // Add vertex
    graph.addVertex(5, 3, 2);
    cout << "Adjacency list after adding vertex" << endl;
    graph.printGraph();

    // Delete vertex
    graph.delVertex(4);
    cout << "Adjacency list after deleting vertex" << endl;
    graph.printGraph();

    return 0;
}
```

Output

```
Initial adjacency list
0 -> 4 -> 3 -> 1
1 -> 2
2
3 -> 2
4 -> 3
5
Adjacency list after adding vertex
0 -> 4 -> 3 -> 1
1 -> 2
2
3 -> 5 -> 2
4 -> 3
5 -> 2
Adjacency list after deleting vertex...
```

Add and Remove vertex in Adjacency Matrix representation of Graph

A **graph** is a presentation of a set of entities where some pairs of entities are linked by a connection. Interconnected entities are represented by points referred to as vertices, and the connections between the vertices are termed as edges. Formally, a graph is a pair of sets (V, E), where V is a collection of vertices, and E is a collection of edges joining a pair of vertices.



A graph can be represented by using an **Adjacency Matrix.**

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

initialization of Graph: The adjacency matrix will be depicted using a 2D array, a constructor will be used to assign the size of the array and each element of that array will be initialized to 0. Showing that the degree of each vertex in the graph is zero.

```
class Graph {
private:
        // number of vertices
        int n;

        // adjacency matrix
        int g[10][10];

public:
        // constructor
        Graph(int x)
        {
                n = x;

                // initializing each element of the adjacency matrix to zero
                for (int i = 0; i < n; ++i) {
                        for (int j = 0; j < n; ++j) {
                                g[i][j] = 0;
```

```
                        }
                }
        }
};
```

Here the adjacency matrix is g[n][n] in which the degree of each vertex is zero.

Displaying the Graph: The graph is depicted using the adjacency matrix g[n][n] having the number of vertices n. The 2D array(adjacency matrix) is displayed in which if there is an edge between two vertices ‘x’ and ‘y’ then g[x][y] is 1 otherwise 0.

```
void displayAdjacencyMatrix()
{
        cout << "\n\n Adjacency Matrix:";

        // displaying the 2D array
        for (int i = 0; i < n; ++i) {
                cout << "\n";
                for (int j = 0; j < n; ++j) {
                        cout << " " << g[i][j];
                }
        }
}
```

The above method is a public member function of the class Graph which displays the graph using an adjacency matrix.

Adding Edges between Vertices in the Graph: To add edges between two existing vertices such as vertex ‘x’ and vertex ‘y’ then the elements g[x][y] and g[y][x] of the adjacency matrix will be assigned to 1, depicting that there is an edge between vertex ‘x’ and vertex ‘y’.

```
void addEdge(int x, int y)
{

        // checks if the vertex exists in the graph
        if ((x >= n) || (y > n)) {
                cout << "Vertex does not exists!";
        }

        // checks if the vertex is connecting to itself
        if (x == y) {
                cout << "Same Vertex!";
        }
        else {
                // connecting the vertices
                g[y][x] = 1;
                g[x][y] = 1;
        }
}
```

Here the above method is a public member function of the class Graph which connects any two existing vertices in the Graph.

Adding a Vertex in the Graph: To add a vertex in the graph, we need to increase both the row and column of the existing adjacency matrix and then initialize the new elements related to that vertex to 0.(i.e the new vertex added is not connected to any other vertex)

```
void addVertex()
{
        // increasing the number of vertices
        n++;
        int i;

        // initializing the new elements to 0
        for (i = 0; i < n; ++i) {
                g[i][n - 1] = 0;
                g[n - 1][i] = 0;
        }
}
```

The above method is a public member function of the class Graph which increments the number of vertices by 1 and the degree of the new vertex is 0.

Removing a Vertex in the Graph: To remove a vertex from the graph, we need to check if that vertex exists in the graph or not and if that vertex exists then we need to shift the rows to the left and the columns upwards of the adjacency matrix so that the row and column values of the given vertex gets replaced by the values of the next vertex and then decrease the number of vertices by 1.In this way that particular vertex will be removed from the adjacency matrix.

```
void removeVertex(int x)
{
        // checking if the vertex is present
        if (x > n) {
                cout << "\nVertex not present!";
                return;
        }
        else {
```

```
            int i;

            // removing the vertex
            while (x < n) {
                // shifting the rows to left side
                for (i = 0; i < n; ++i) {
                    g[i][x] = g[i][x + 1];
                }

                // shifting the columns upwards
                for (i = 0; i < n; ++i) {
                    g[x][i] = g[x + 1][i];
                }
                x++;
            }

            // decreasing the number of vertices
            n--;
        }
}
```

The above method is a public member function of the class Graph which removes an existing vertex from the graph by shifting the rows to the left and shifting the columns up to replace the row and column values of that vertex with the next vertex and then decreases the number of vertices by 1 in the graph.

Following is a complete program that uses all of the above methods in a Graph.

```
// C++ program to add and remove Vertex in Adjacency Matrix

#include <iostream>

using namespace std;

class Graph {
private:
    // number of vertices
    int n;

    // adjacency matrix
    int g[10][10];

public:
```

```
// constructor
Graph(int x)
{
	n = x;

	// initializing each element of the adjacency matrix to zero
	for (int i = 0; i < n; ++i) {
		for (int j = 0; j < n; ++j) {
			g[i][j] = 0;
		}
	}
}

void displayAdjacencyMatrix()
{
	cout << "\n\n Adjacency Matrix:";

	// displaying the 2D array
	for (int i = 0; i < n; ++i) {
		cout << "\n";
		for (int j = 0; j < n; ++j) {
			cout << " " << g[i][j];
		}
	}
}

void addEdge(int x, int y)
{

	// checks if the vertex exists in the graph
	if ((x >= n) || (y > n)) {
		cout << "Vertex does not exists!";
	}

	// checks if the vertex is connecting to itself
	if (x == y) {
		cout << "Same Vertex!";
	}
	else {
		// connecting the vertices
```

```
                g[y][x] = 1;
                g[x][y] = 1;
        }
}

void addVertex()
{
        // increasing the number of vertices
        n++;
        int i;

        // initializing the new elements to 0
        for (i = 0; i < n; ++i) {
                g[i][n - 1] = 0;
                g[n - 1][i] = 0;
        }
}

void removeVertex(int x)
{
        // checking if the vertex is present
        if (x > n) {
                cout << "\nVertex not present!";
                return;
        }
        else {
                int i;

                // removing the vertex
                while (x < n) {
                        // shifting the rows to left side
                        for (i = 0; i < n; ++i) {
                                g[i][x] = g[i][x + 1];
                        }

                        // shifting the columns upwards
                        for (i = 0; i < n; ++i) {
                                g[x][i] = g[x + 1][i];
                        }
                        x++;
```

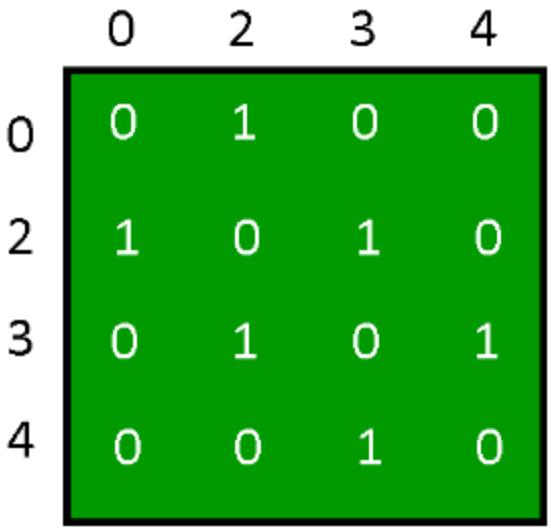
```
                        }

                        // decreasing the number of vertices
                        n--;
                }
        }
};

int main()
{
        // creating objects of class Graph
        Graph obj(4);

        // calling methods
        obj.addEdge(0, 1);
        obj.addEdge(0, 2);
        obj.addEdge(1, 2);
        obj.addEdge(2, 3);
        // the adjacency matrix created
        obj.displayAdjacencyMatrix();

        // adding a vertex to the graph
        obj.addVertex();
        // connecting that vertex to other existing vertices
        obj.addEdge(4, 1);
        obj.addEdge(4, 3);
        // the adjacency matrix with a new vertex
        obj.displayAdjacencyMatrix();

        // removing an existing vertex in the graph
        obj.removeVertex(1);
        // the adjacency matrix after removing a vertex
        obj.displayAdjacencyMatrix();

        return 0;
}
```

Output:-

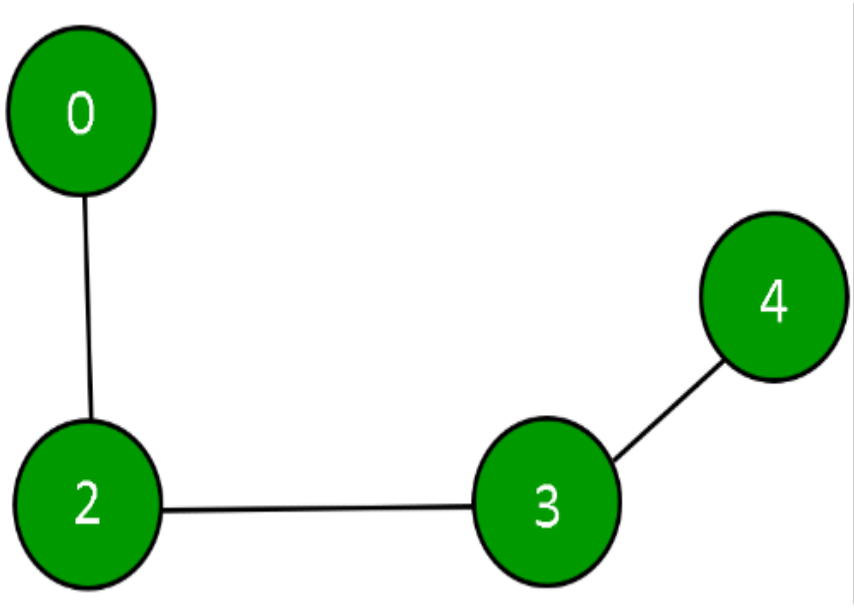
```
Adjacency Matrix:
0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0

Adjacency Matrix:
0 1 1 0 0
1 0 1 0 1
1 1 0 1 0
0 0 1 0 1
0 1 0 1 0

Adjacency Matrix:
0 1 0 0
1 0 1 0
0 1 0 1
0 0 1 0
```

Adjacency matrices waste a lot of memory space. Such matrices are found to be very sparse. This representation requires space for n*n elements, the time complexity of the addVertex() method is O(n), and the time complexity of the removeVertex() method is O(n*n) for a graph of n vertices.

From the output of the program, the Adjacency Matrix is:



And the Graph depicted by the above Adjacency Matrix is:

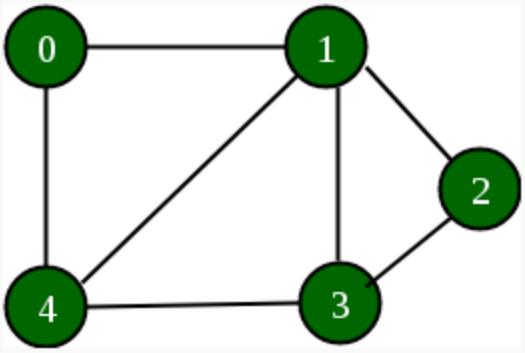


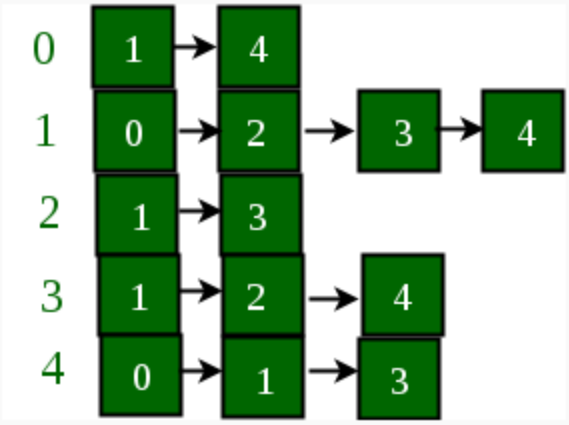
Add and Remove Edge in Adjacency List representation of a Graph

adding and removing edge is discussed in a given adjacency list representation. A vector has been used to implement the graph using adjacency list representation. It is used to store the adjacency lists of all the vertices. The vertex number is used as the index in this vector.

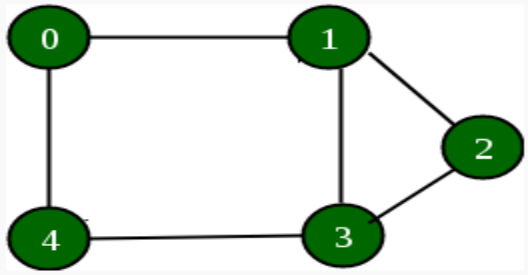
Example:

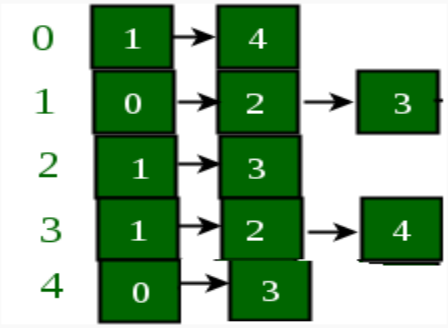
Below is a graph and its adjacency list representation:





*If the edge between **1** and **4** has to be removed, then the above graph and the adjacency list transforms to:*





Approach: The idea is to represent the graph as an array of vectors such that every vector represents adjacency list of the vertex.

- **Adding an edge:** Adding an edge is done by inserting both of the vertices connected by that edge in each others list. For example, if an edge between **(u, v)** has to be added, then **u** is stored in **v's vector list** and **v** is stored in **u's vector list**. (push_back)
- **Deleting an edge:** To delete edge between **(u, v)**, **u's adjacency list** is traversed until **v** is found and it is removed from it. The same operation is performed for **v**.(erase)

Below is the implementation of the approach:

```
// C++ implementation of the above approach

#include <bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
        adj[u].push_back(v);
        adj[v].push_back(u);
}

// A utility function to delete an edge in an
// undirected graph.
void delEdge(vector<int> adj[], int u, int v)
{
        // Traversing through the first vector list
        // and removing the second element from it
        for (int i = 0; i < adj[u].size(); i++) {
                if (adj[u][i] == v) {
                        adj[u].erase(adj[u].begin() + i);
                        break;
                }
        }

        // Traversing through the second vector list
        // and removing the first element from it
```

```
        for (int i = 0; i < adj[v].size(); i++) {
                if (adj[v][i] == u) {
                        adj[v].erase(adj[v].begin() + i);
                        break;
                }
        }
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
        for (int v = 0; v < V; ++v) {
                cout << "vertex " << v << " ";
                for (auto x : adj[v])
                        cout << "-> " << x;
                printf("\n");
        }
        printf("\n");
}

// Driver code
int main()
{
        int V = 5;
        vector<int> adj[V];

        // Adding edge as shown in the example figure
        addEdge(adj, 0, 1);
        addEdge(adj, 0, 4);
        addEdge(adj, 1, 2);
        addEdge(adj, 1, 3);
        addEdge(adj, 1, 4);
        addEdge(adj, 2, 3);
        addEdge(adj, 3, 4);

        // Printing adjacency matrix
        printGraph(adj, V);

        // Deleting edge (1, 4)
```

```
        // as shown in the example figure
        delEdge(adj, 1, 4);

        // Printing adjacency matrix
        printGraph(adj, V);

        return 0;
}
```

Output:-

```
vertex 0 -> 1-> 4
vertex 1 -> 0-> 2-> 3-> 4
vertex 2 -> 1-> 3
vertex 3 -> 1-> 2-> 4
vertex 4 -> 0-> 1-> 3

vertex 0 -> 1-> 4
vertex 1 -> 0-> 2-> 3
vertex 2 -> 1-> 3
vertex 3 -> 1-> 2-> 4
vertex 4 -> 0-> 3
```

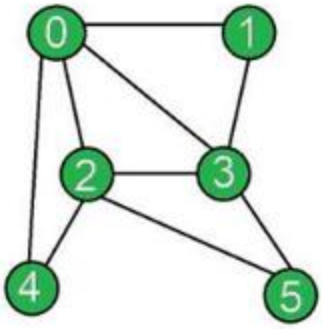
Time Complexity: Removing an edge from adjacent list requires, on the average time complexity will be O(|E| / |V|) , which may result in cubical complexity for dense graphs to remove all edges.
Auxiliary Space: O(V) **,** here V is number of vertices.
Add and Remove Edge in Adjacency Matrix representation of a Graph

Given an adjacency matrix **g[][]** of a graph consisting of N vertices, the task is to modify the matrix after insertion of all **edges[]** and removal of edge between vertices **(X, Y)**. In an adjacency matrix, if an edge exists between vertices **i** and **j** of the graph, then **g[i][j] = 1** and **g[j][i] = 1**. If no edge exists between these two vertices, then **g[i][j] = 0** and **g[j][i] = 0**.
Examples:
Input: *N = 6, Edges[] = {{0, 1}, {0, 2}, {0, 3}, {0, 4}, {1, 3}, {2, 3}, {2, 4}, {2, 5}, {3, 5}}, X = 2, Y = 3*
Output:

Adjacency matrix after edge insertion:
0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 1 1 1
1 1 1 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0
Adjacency matrix after edge removal:
0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 0 1 1
1 1 0 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0

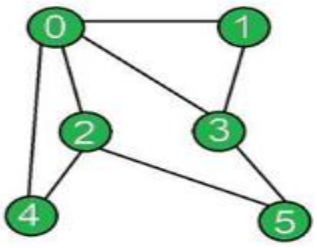
Explanation:

The graph and the corresponding adjacency matrix after insertion of edges:



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

*The graph after removal and adjacency matrix after removal of edge between vertex **X** and **Y**:*



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	0	1	1
3	1	1	0	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

Input: *N = 6, Edges[] = {{0, 1}, {0, 2}, {0, 3}, {0, 4}, {1, 3}, {2, 3}, {2, 4}, {2, 5}, {3, 5}}, X = 3, Y = 5*

Output:

Adjacency matrix after edge insertion:

0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 1 1 1
1 1 1 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0

Adjacency matrix after edge removal:

0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 1 1 1
1 1 1 0 0 0
1 0 1 0 0 0
0 0 1 0 0 0

Approach:

Initialize a matrix of dimensions **N x N** and follow the steps below:

- **Inserting an edge:** To insert an edge between two vertices suppose **i** and **j**, set the corresponding values in the adjacency matrix equal to 1, i.e. ***g[i][j]=1*** and ***g[j][i]=1*** if both the vertices **i** and **j** exists.
- **Removing an edge:** To remove an edge between two vertices suppose **i** and **j**, set the corresponding values in the adjacency matrix equal to 0. That is, set ***g[i][j]=0*** and ***g[j][i]=0*** if both the vertices **i** and **j** exists.

Below is the implementation of the above approach:

```
// C++ program to add and remove edge
// in the adjacency matrix of a graph

#include <iostream>
using namespace std;

class Graph {
private:
        // Number of vertices
        int n;
```

```
        // Adjacency matrix
        int g[10][10];

public:
        // Constructor
        Graph(int x)
        {
                n = x;

                // Initializing each element of the
                // adjacency matrix to zero
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < n; j++) {
                                g[i][j] = 0;
                        }
                }
        }

        // Function to display adjacency matrix
        void displayAdjacencyMatrix()
        {
                // Displaying the 2D matrix
                for (int i = 0; i < n; i++) {
                        cout << "\n";
                        for (int j = 0; j < n; j++) {
                                cout << " " << g[i][j];
                        }
                }
        }

        // Function to update adjacency
        // matrix for edge insertion
        void addEdge(int x, int y)
        {
                // Checks if the vertices
                // exist in the graph
                if ((x < 0) || (x >= n)) {
                        cout << "Vertex" << x
                                << " does not exist!";
                }
```

```
        if ((y < 0) || (y >= n)) {
                cout << "Vertex" << y
                        << " does not exist!";
        }

        // Checks if it is a self edge
        if (x == y) {
                cout << "Same Vertex!";
        }

        else {
                // Insert edge
                g[y][x] = 1;
                g[x][y] = 1;
        }
}

// Function to update adjacency
// matrix for edge removal
void removeEdge(int x, int y)
{
        // Checks if the vertices
        // exist in the graph
        if ((x < 0) || (x >= n)) {
                cout << "Vertex" << x
                        << " does not exist!";
        }
        if ((y < 0) || (y >= n)) {
                cout << "Vertex" << y
                        << " does not exist!";
        }

        // Checks if it is a self edge
        if (x == y) {
                cout << "Same Vertex!";
        }

        else {
                // Remove edge
                g[y][x] = 0;
```

```
                        g[x][y] = 0;
                }
        }
};

// Driver Code
int main()
{
        int N = 6, X = 2, Y = 3;

        Graph obj(N);

        // Adding edges to the graph
        obj.addEdge(0, 1);
        obj.addEdge(0, 2);
        obj.addEdge(0, 3);
        obj.addEdge(0, 4);
        obj.addEdge(1, 3);
        obj.addEdge(2, 3);
        obj.addEdge(2, 4);
        obj.addEdge(2, 5);
        obj.addEdge(3, 5);

        cout << "Adjacency matrix after"
                << " edge insertions:\n";
        obj.displayAdjacencyMatrix();

        obj.removeEdge(X, Y);

        cout << "\nAdjacency matrix after"
                << " edge removal:\n";
        obj.displayAdjacencyMatrix();

        return 0;
}
```

Output

```
Adjacency matrix after edge insertions:

 0 1 1 1 1 0
 1 0 0 1 0 0
 1 0 0 1 1 1
 1 1 1 0 0 1
 1 0 1 0 0 0
 0 0 1 1 0 0
Adjacency matrix after edge removal:

 0 1 1 1 1 0
 1 0 0 1 0 0
 1 0 0 0 1 1
 1 1 0 0 0 1
 1 0 1 0 0 0
 0 0 1 1 0 0
```

Time Complexity: *Insertion and Deletion of an edge requires O(1) complexity while it takes $O(N^2)$ to display the adjacency matrix.*
Auxiliary Space: *$O(N^2)$*

Breadth First Search or BFS for a Graph

Breadth First Search (BFS) is a fundamental **graph traversal algorithm.** It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. This is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level. A lot of popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS. BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

BFS from a Given Source:

The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to the Breadth-First Traversal of a tree. Like tree, we begin with the given source (in tree, we begin with root) and traverse vertices level by level using a queue data structure. The only catch here is that, unlike trees,

graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a **boolean visited** array.
Initialization: Enqueue the given source vertex into a queue and mark it as visited.

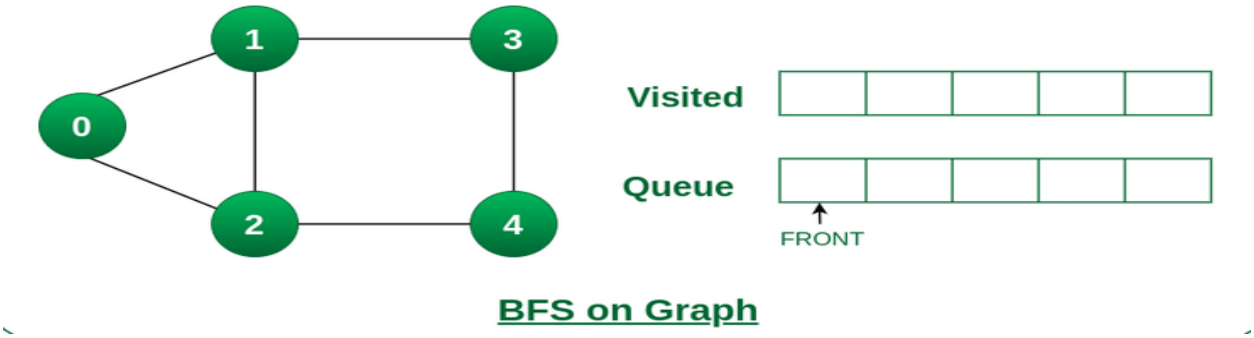
1. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
        - Enqueue the neighbor into the queue.
        - Mark the neighbor as visited.
2. **Termination:** Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

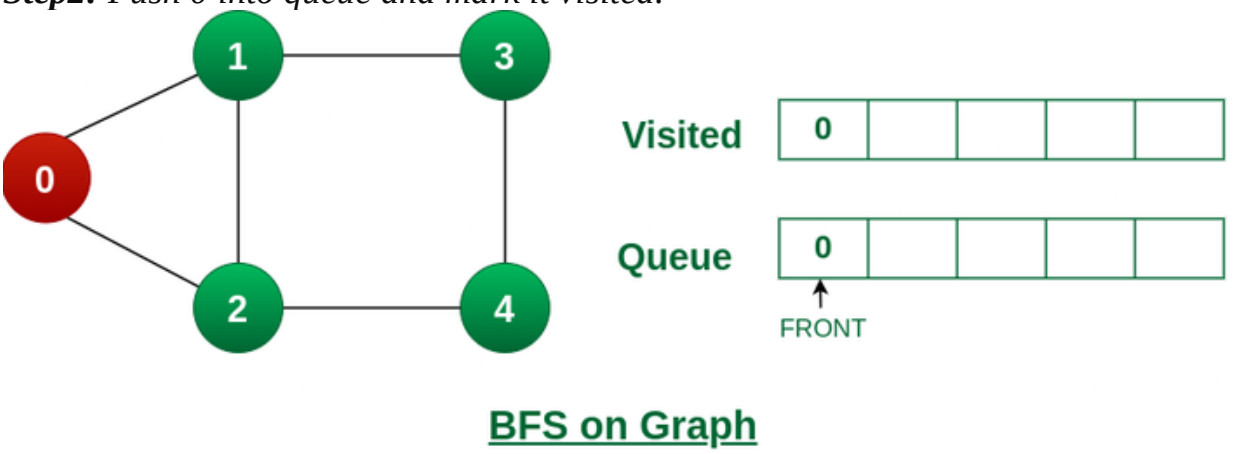
How Does the BFS Algorithm Work?

Let us understand the working of the algorithm with the help of the following example where the **source vertex is 0**.

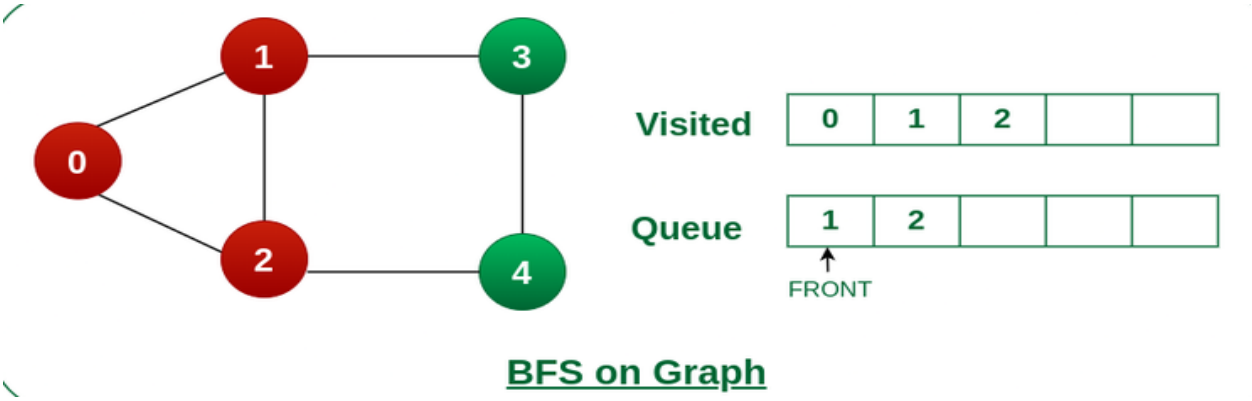
Step1: *Initially queue and visited arrays are empty.*



Step2: *Push 0 into queue and mark it visited.*



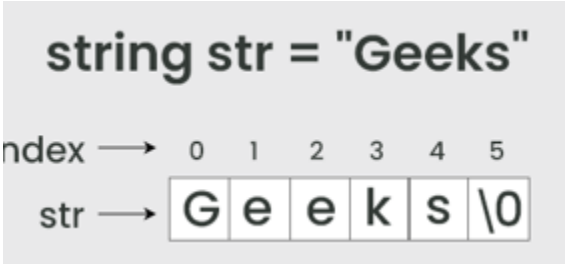
Step 3: *Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



BFS on Graph

Chapter 4
String in Data Structure

A **string** is a sequence of characters used to represent text. **Strings** are commonly used for storing and manipulating textual data in computer programs. They can be manipulated using various operations like **concatenation, substring extraction, and comparison**.



String is considered a **data type** in general and is typically represented as arrays of bytes (or words) that store a sequence of characters. String is defined as an **array of characters**. The difference between a character array and a string is the string is terminated with a special character **'\0'**. Some examples of strings are: "geeks" , "for", "geeks", "GeeksforGeeks", "Geeks for Geeks", "123Geeks", "@123 Geeks".

String Data Type:

In most programming languages, strings are treated as a distinct **data type**. This means that strings have their own set of operations and properties. They can be declared and manipulated using specific string-related functions and methods.

String Operations:

Strings support a wide range of operations, including concatenation, substring extraction, length calculation, and more. These operations allow developers to manipulate and process string data efficiently.

Below are fundamental operations commonly performed on strings in programming.

- **Concatenation:** Combining two strings to create a new string.
- **Length:** Determining the number of characters in a string.
- **Access:** Accessing individual characters in a string by index.
- **Substring:** Extracting a portion of a string.
- **Comparison:** Comparing two strings to check for equality or order.
- S**earch:** Finding the position of a specific substring within a string.
- **Modification:** Changing or replacing characters within a string.

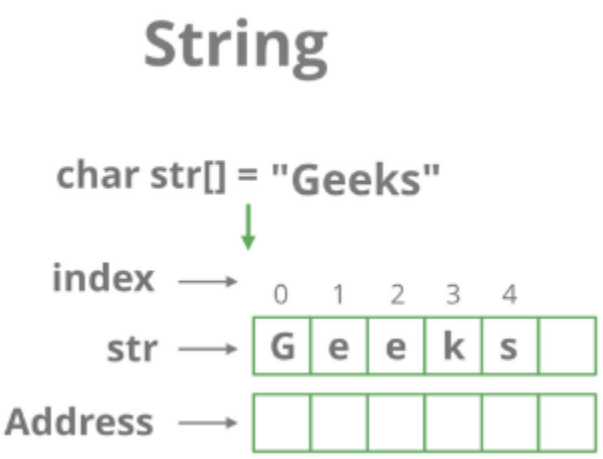
Applications of String:

- **Text Processing:** Strings are extensively used for text processing tasks such as searching, manipulating, and analyzing textual data.
- **Data Representation**: Strings are fundamental for representing and manipulating data in formats like JSON, XML, and CSV.
- **Encryption and Hashing:** Strings are commonly used in encryption and hashing algorithms to secure sensitive data and ensure data integrity.
- **Database Operations:** Strings are essential for working with databases, including storing and querying text-based data.
- **Web Development:** Strings are utilized in web development for constructing URLs, handling form data, processing input from web forms, and generating dynamic content.
- Below are some examples of strings:

"geeks" , "for", "geeks", "GeeksforGeeks", "Geeks for Geeks", "123Geeks", "@123 Geeks"

How String is represented in Memory?

In C, a string can be referred to either using a character pointer or as a character array. When strings are declared as character arrays, they are stored like other types of arrays in C. For example, if str[] is an auto variable then the string is stored in the stack segment, if it's a global or static variable then stored in the data segment, etc.



How to Declare Strings in various languages?

Below is the representation of strings in C++ languages:

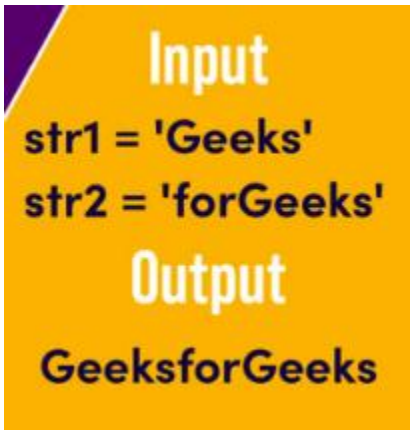
```
// C++ program to demonstrate String
// using Standard String representation

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

int main()
{

    // Declare and initialize the string
    string str1 = "Welcome to GeeksforGeeks!";

    // Initialization by raw string
    string str2("A Computer Science Portal");

    // Print string
    cout << str1 << endl << str2;

    return 0;
}
```

General Operations performed on String:

Here we are providing you with some must-know concepts of string:

1. Concatenation of Strings

The process of combining more than one string together is known as Concatenation. String Concatenation is the technique of combining two strings.



There are two ways to concatenate two strings:

a) String concatenation without using any inbuilt methods:

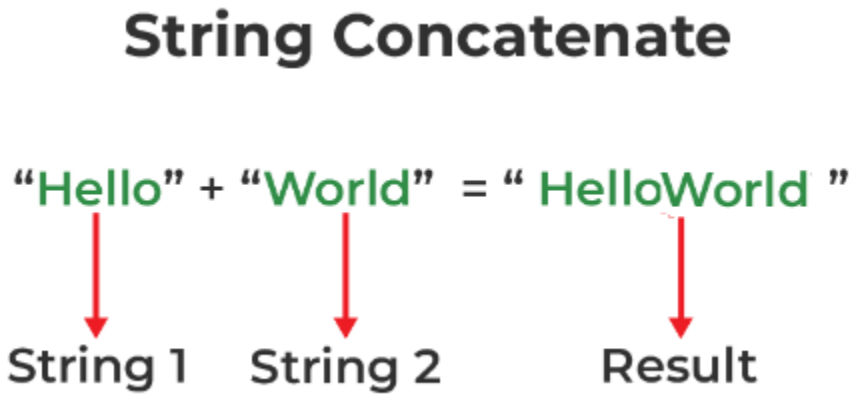
Below is the algorithm for the Concatenation of two strings:

Algorithm: CONCATENATE (STR1, STR2, STR3)

```
1. LEN1 = LENGTH(STR1).
2. LEN2 = LENGTH(STR2).
3. SET I = 0.
4. Repeat Steps 5 and 6 while I < LEN1-1:
5.     STR3[I] = STR1[I].
6.     SET I = I+1.
7. SET J = 0.
8. Repeat Steps 9 to 11 while I < (LEN1 + LEN2 - 2):
9.     STR3[I] = STR2[J].
10.    J = J+1.
11.    I = I+1.
12.Exit.
```

b) String concatenation using inbuilt methods:

The string is a type of data structure used for storing characters. Concatenating strings in C++ is one of the most discussed topics related to strings. There are multiple methods to concat strings using user-defined methods, and a couple of methods for the concatenation of strings using pre-defined methods. Let's check on all of these methods.



Methods of Concatenate String

There are 6 methods to Concatenate String as mentioned below:

1. Using append( ) Function.
2. Using ‘+’ Operator.
3. Using strcat( ) Function.
4. Using C++ for Loop.
5. Using Inheritance.
6. Using the Friend Function and strcat() Function.

1. Using append() Function

The append() function is a member function of the **std::string class**. Using this function, we can concatenate two std::string objects (C++ style strings) as shown in the below example.

Syntax:

```
string& string::append (const string& str);
Here,
str: String to be appended.
```

Below is the C++ program for string concatenation using the append() function:

```
// C++ Program for string
// concatenation using append
#include <iostream>
using namespace std;

// Driver code
int main()
{
        string init("this is init");
```

```
        string add(" added now");

        // Appending the string.
        init.append(add);

        cout << init << endl;
        return 0;
}
```

Output

```
this is init added now
```

2. Using '+' Operator

This is the easiest method for the concatenation of two strings.

The + operator **adds strings** and returns a concatenated string. This method only works for C++ style strings (std::string objects) and doesn't work on C style strings (character array).

Syntax:

```
string new_string = init + add;
```

Below is the C++ program for string concatenation using '+' operator:

- C++

```
// C++ Program for string
// concatenation using '+' operator
#include <iostream>
using namespace std;

// Driver code
int main()
{
    string init("this is init");
    string add(" added now");

    // Appending the string.
    init = init + add;

    cout << init << endl;
    return 0;
}
```

Output

```
this is init added now
```

3. Using strcat( ) Function

The C++ strcat( ) function is a built-in function defined in **<string.h>** header file. This function concatenates the two strings **init** and **add** and the result is stored in the **init** string. This function only works for C-style strings (character arrays) and doesn't work for C++-style strings (std::string objects).

Syntax:

```
char * strcat(char * init, const char * add);
```

Below is the C++ program for string concatenation using strcat() function:

```
// C++ Program for string
// concatenation using strcat
#include <iostream>
#include <string.h>
using namespace std;

// Driver code
int main()
{
        char init[] = "this is init";
        char add[] = " added now";

        // Concatenating the string.
        strcat(init, add);

        cout << init << endl;

        return 0;
}
```

Output

```
this is init added now
```

4. Using for Loop

Using a loop is one of the most basic methods of string concatenation. Here, we are adding elements one by one while traversing the whole string and

then another string. The final result will be the concatenated string formed from both strings.

Below is the C++ program for string concatenation using for loop:

```
// C++ Program for string
// concatenation using for loop
#include <iostream>
using namespace std;

// Driver code
int main()
{
	string init("this is init");
	string add(" added now");

	string output;

	// Adding element inside output
	// from init
	for (int i = 0; init[i] != '\0'; i++)
	{
		output += init[i];
	}

	// Adding element inside output
	// fromt add
	for (int i = 0; add[i] != '\0'; i++)
	{
		output += add[i];
	}

	cout << output << endl;
	return 0;
}
```

Output

```
this is init added now
```

5. Using Inheritance

Below is the C++ program for string concatenation using inheritance:

```
// C++ program for string concatenation
// using inheritance
#include <iostream>
#include <string>
using namespace std;

// Base class
class base
{
        protected:
                virtual string concatenate(string &str1,
                                                        string &str2) = 0;
};

// Derive class
class derive: protected base {
        public:
                string concatenate (string &str1,
                                                string &str2)
                {
                        string temp;
                        temp = str1 + str2;
                        return temp;
                }
};

// Driver code
int main()
{
        string init("this is init");
        string add(" added now");

        // Create string object
        derive obj;

        // Print string
        cout << obj.concatenate (init, add);
```

```
        return 0;
}
```

Output

```
this is init added now
```

6. Using the Friend Function and strcat() function

Below is the C++ program for string concatenation using the friend function and strcat() function:

```
// C++ program for string concatenation
// using friend function and strcat()
#include <iostream>
#include <string.h>
using namespace std;

// Base class
class Base {
        public:
        char init[100] = "this is init";
        char add[100] = " added now";

        friend void myfun(Base b);
};

void myfun (Base b)
{
        // Pass parameter to concatenate
        strcat (b.init, b.add);

        cout << b.init;
}

// Driver code
int main()
{

        // Create object of base class
        Base b;
```

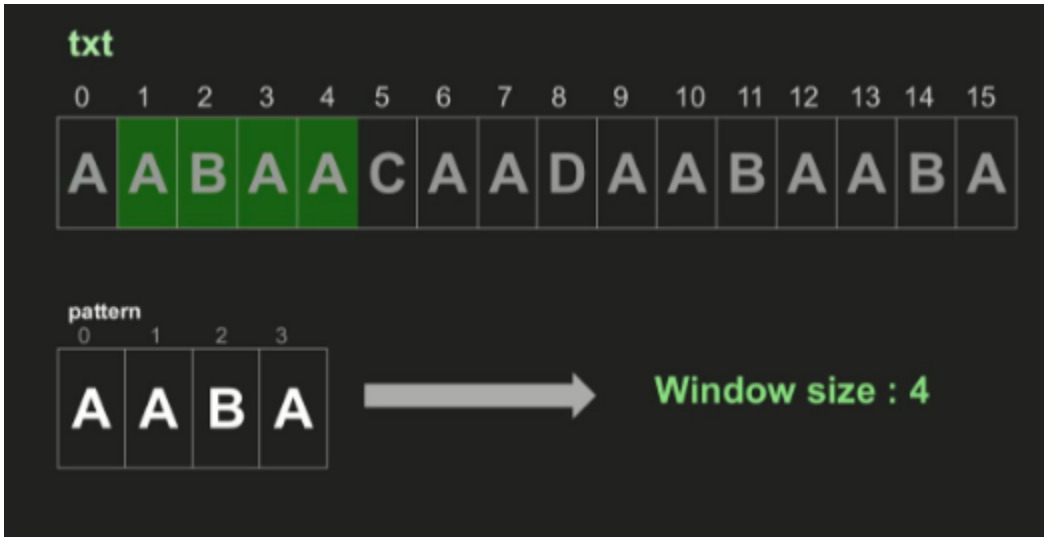
```
        // pass b object to myfun() to print
        // the concatenated string
        myfun(b);

        return 0;
}
```

Output

```
this is init added now
```

2. Find in String

A very basic operation performed on Strings is to find something in the given whole string. Now, this can be to find a given character in a string, or to find a complete string in another string.



a character in the string. These types of problems are very competitive programming where you need to locate the position of the character in a string.

b) Find a substring in another string:

Consider there to be a string of length N and a substring of length M. Then run a nested loop, where the outer loop runs from 0 to (N-M) and the inner loop from 0 to M. For every index check if the sub-string traversed by the inner loop is the given sub-string or not.

An efficient solution is to use a O(n) searching algorithm like KMP algorithm, Z algorithm, etc.

Language implementations:

b) Find a character in string:

Given a string and a character, your task is to find the first position of the

Substring in C++

The substring function is used for handling string operations like *strcat(), append()*, etc. It generates a new string with its value initialized to a copy of a sub-string of this object. In C++, the header file which is required for std::substr(), string functions is **<string>**.

The substring function takes two values **pos** and **len** as an argument and returns a newly constructed string object with its value initialized to a copy of a sub-string of this object. Copying of string starts from *pos* and is done till *pos+len* means [pos, pos+len).

Syntax:

```
string substr (size_t pos, size_t len) const;
```

Parameters:

- **pos:** Index of the first character to be copied.
- **len:** Length of the sub-string.
- **size_t:** It is an unsigned integral type.

Return Value: It returns a string object.

Example :

```
// C++ program to demonstrate functioning of substr()
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // Take any string
   string s1 = "Geeks";

   // Copy two characters of s1 (starting
   // from index 3)
```

```
    string r = s1.substr(3, 2);

    // prints the result
    cout << "String is: " << r;

    return 0;
}
```

Output

```
String is: ks
```

- **Time complexity:** O(N)
- **Auxiliary Space:** O(N)

Important Points to Remember

1. The index of the first character is 0 (not 1).
2. If **pos** is equal to the string length, the function returns an empty string.

If **pos** is greater than the string length, it throws out_of_range. If this happens, there are no changes in the string.

4. If the requested sub-string **len** is greater than the size of a string, then returned sub-string is **[pos, size())**.
5. If **len** is not passed as a parameter, then returned sub-string is **[pos, size()).**

Applications of Substring

- Get a Sub-String after a character
- Get a Sub-String before a character
- Print all Sub-Strings of a given String
- Sum of all Substrings of a string representing a number
- Print the maximum value of all substrings of a string representing a number
- Print the minimum value of all substrings of a string representing a number

Get a Sub-String after a Character

In this, a string and a character are given and you have to print the sub-string followed by the given character.
Extract everything after the *":"* in the string *"**dog:cat**"*.

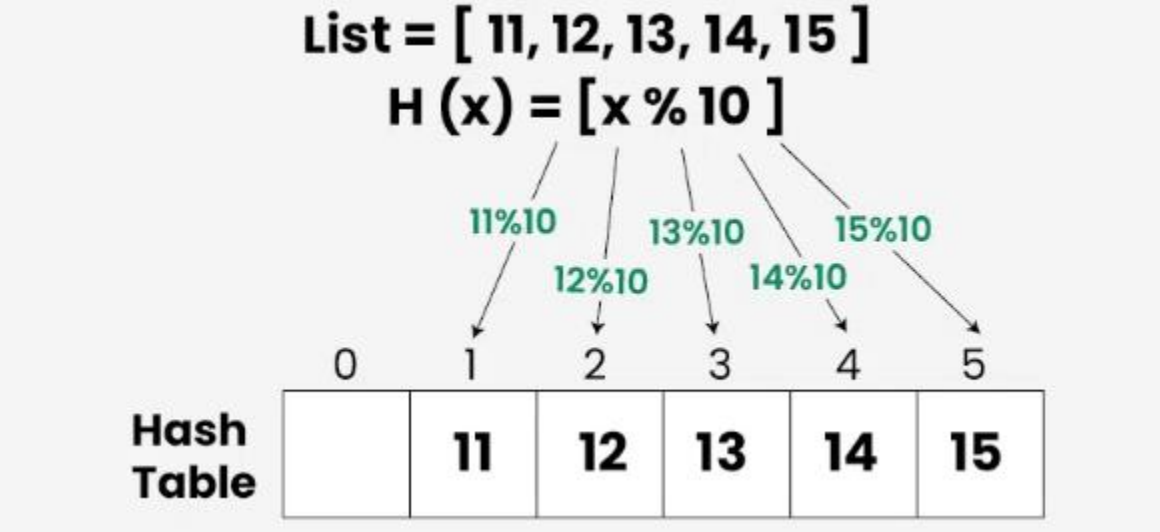
Example :-

```
// C++ program to demonstrate functioning of substr()
#include <iostream>
#include <string>
using namespace std;

int main()
{
    // Take any string
    string s = "dog:cat";

    // Find position of ':' using find()
    int pos = s.find(":");

    // Copy substring after pos
    string sub = s.substr(pos + 1);

    // prints the result
    cout << "String is: " << sub;

    return 0;
}
```

Chapter 5

Introduction to Hashing

Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.



Hashing in Data Structures refers to the process of transforming a given key to another value. It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key. The transformation of a key to the corresponding value is done using a **Hash Function** and the value obtained from the hash function is called **Hash Code** .

Need for Hash data structure

Every day, the data on the internet is increasing multifold and it is always a struggle to store this data efficiently. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A very common data structure that is used for such a purpose is the Array data structure.

Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word ” **efficiency** “. Though storing in

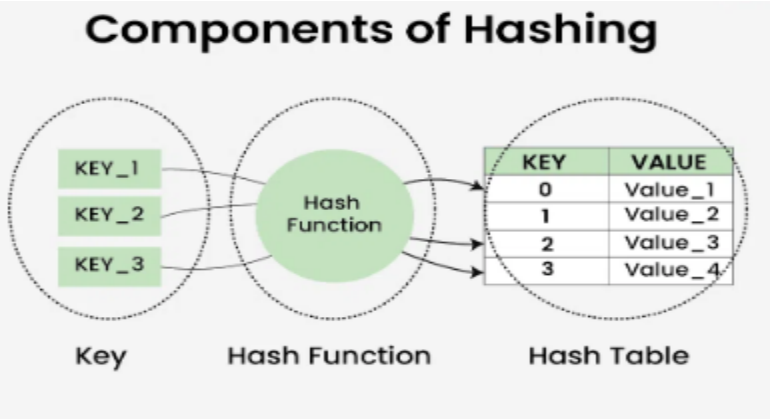
Array takes O(1) time, searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient.

So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in O(1) time. This is how Hashing data structure came into play. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well.

Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index** .
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.



How does Hashing work?

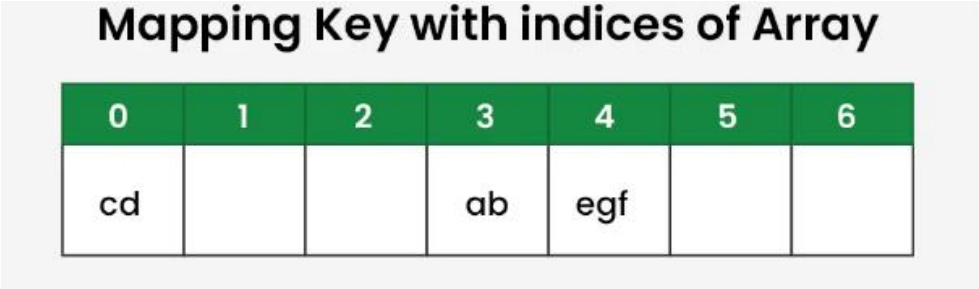
Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

Our main objective here is to search or update the values stored in the table quickly in O(1) time and we are not concerned about the ordering of strings in the table. So the given set of strings can act as a key and the string itself will act as the value of the string but how to store the value corresponding to the key?

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
  - "a" = 1,
  - "b"=2, .. etc, to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:

  - *"ab" = 1 + 2 = 3,*
  - *"cd" = 3 + 4 = 7 ,*
  - *"efg" = 5 + 6 + 7 = 18*

- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size** . We can compute the location of the string in the array by taking the **sum(string) mod 7** .
- **Step 5:** So we will then store
  - "ab" in 3 mod 7 = 3,
  - "cd" in 7 mod 7 = 0, and
  - "efg" in 18 mod 7 = 4.



The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table.

What is a Hash function?

The hash function creates a mapping between key and value, this is done through the use of mathematical formulas known as hash functions. The result of the hash function is referred to as a hash value or hash. The hash value is a representation of the original string of characters but usually smaller than the original.

For example: Consider an array as a Map where the key is the index and the value is the value at that index. So for an array A if we have index **i** which will be treated as the key then we can find the value by simply looking at the value at A[i].

Types of Hash functions:

There are many hash functions that use numeric or alphanumeric keys. This chapter focuses on discussing different hash functions :

1. Division Method.
2. Mid Square Method
3. Folding Method.
4. Multiplication Method

Properties of a Good hash function

A hash function that maps every item into its own unique slot is known as a perfect hash function. We can construct a perfect hash function if we know the items and the collection will never change but the problem is that there is no systematic way to construct a perfect hash function given an arbitrary collection of items. Fortunately, we will still gain performance efficiency even if the hash function isn't

perfect. We can achieve a perfect hash function by increasing the size of the hash table so that every possible value can be accommodated. As a result, each item will have a unique slot. Although this approach is feasible for a small number of items, it is not practical when the number of possibilities is large.

So, We can construct our hash function to do the same but the things that we must be careful about while constructing our own hash function.

A good hash function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position is equally likely for each.
3. Should minimize collisions.
4. Should have a low load factor(number of items in the table divided by the size of the table).

Complexity of calculating hash value using the hash function

- Time complexity: O(n)
- Space complexity: O(1)

Problem with Hashing

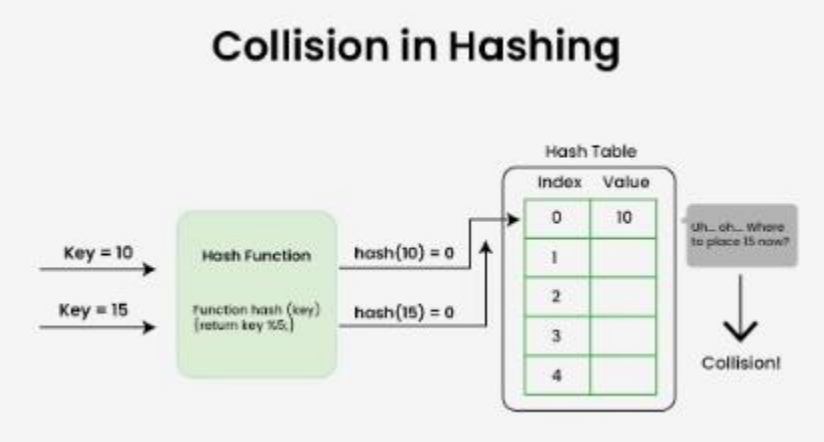
If we consider the above example, the hash function we used is the sum of the letters, but if we examined the hash function closely then the problem can be easily visualized that for different strings same hash value is begin generated by the hash function.

For example: {“ab”, “ba”} both have the same hash value, and string {“cd”,”be”} also generate the same hash value, etc. This is known as **collision** and it creates problem in searching, insertion, deletion, and updating of value.

What is Collision?

Collision in Hashing occurs when two different keys map to the same hash value. Hash collisions can be intentionally created for many hash algorithms. The probability of a hash collision depends on the size of the algorithm, the distribution of hash values and the efficiency of Hash function.

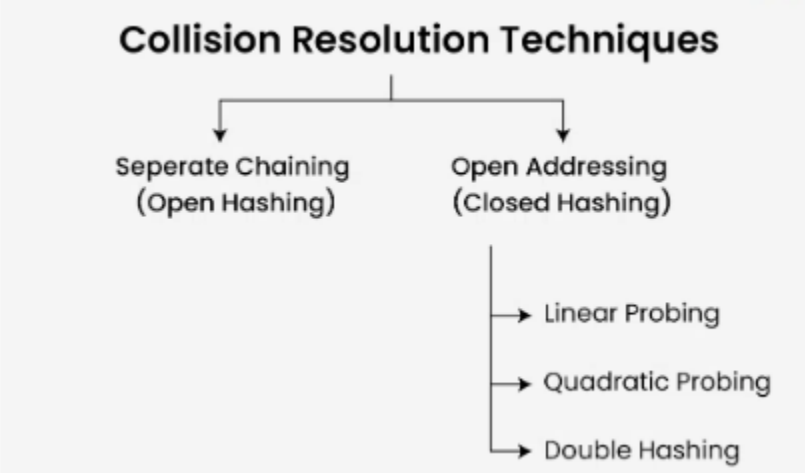
The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing



1) Separate Chaining

The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

```
Hash function = key % 5,
Elements = 12, 15, 22, 25 and 37.
```

2) Open Addressing

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

2.a) Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

Algorithm:

1. *Calculate the hash key. i.e.* ***key = data % size***
2. *Check, if* ***hashTable[key]*** *is empty*
   - *store the value directly by* ***hashTable[key] = data***
3. *If the hash index already has some value then*
   - *check for next index using* ***key = (key+1) % size***
4. *Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.*
5. *Do the above process till we find the space.*

2.b) Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

An example sequence using quadratic probing is:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2 \ldots\ldots\ldots\ldots\ldots\ldots H + k^2$$

This method is also known as the mid-square method because in this method we look for i 2 'th probe (slot) in i'th iteration and the value of i = 0, 1, . . . n – 1. We always start from the original hash location. If only the location is occupied then we check the other slots.

Let hash(x) be the slot index computed using the hash function and n be the size of the hash table.

If the slot hash(x) % n is full, then we try (hash(x) + 1^2) % n.
If (hash(x) + 1^2) % n is also full, then we try (hash(x) + 2^2) % n.
If (hash(x) + 2^2) % n is also full, then we try (hash(x) + 3^2) % n.
*This process will be repeated for all the values of **i** until an empty slot is found*

2.c) Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing make use of two hash function,

- The first hash function is **h1(k)** which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key.
- But in case the location is occupied (collision) we will use secondary hash-function **h2(k)** in combination with the first hash-function **h1(k)** to find the new location on the hash table.

This combination of hash functions is of the form

```
h(k, i) = (h1(k) + i * h2(k)) % n
```

where

- i is a non-negative integer that indicates a collision number,
- k = element/key which is being hashed
- n = hash table size.

Complexity of the Double hashing algorithm:

```
Time complexity: O(n)
```

What is meant by Load Factor in Hashing?

The load factor of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hash function or want to add more elements to the existing hash table.

It helps us in determining the efficiency of the hash function i.e. it tells whether the hash function which we are using is distributing the keys uniformly or not in the hash table.

```
Load Factor = Total elements in hash table/ Size of hash table
```

What is Rehashing?

As the name suggests, rehashing means hashing again. Basically, when the load factor increases to more than its predefined value (the default value of the load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double-sized array to maintain a low load factor and low complexity.

Applications of Hash Data structure

- Hash is used in databases for indexing.
- Hash is used in disk-based data structures.
- In some programming languages like Python, JavaScript hash is used to implement objects.

Real-Time Applications of Hash Data structure

- Hash is used for cache mapping for fast access to the data.
- Hash can be used for password verification.
- Hash is used in cryptography as a message digest.
- Rabin-Karp algorithm for pattern matching in a string.
- Calculating the number of different substrings of a string.

Advantages of Hash Data structure

- Hash provides better synchronization than other data structures.
- Hash tables are more efficient than search trees or other data structures
- Hash provides constant time for searching, insertion, and deletion operations on average.

Disadvantages of Hash Data structure

- Hash is inefficient when there are many collisions.
- Hash collisions are practically not avoided for a large set of possible keys.
- Hash does not allow null values.

References

1- **Handbook of Data Structures and Applications,** edited by Dinesh P. Mehta, Sartaj Sahni
2- Data Structures and Algorithm Analysis in C++, Fourth Edition Mark Allen Weiss Florida International University.
3- Tree Traversal Techniques - GeeksforGeeks