Unit-1

Introduction:

WHAT IS DATA STRUCTURE?

In computer science, a data structure is a way of organizing and storing data in a computer program so that it can be accessed and used efficiently. Data structures provide a means of managing large amounts of data, enabling efficient searching, sorting, insertion and deletion of data.

Data structures can be categorized into two types: **Primitive data structures** and **non-primitive data structures.**

Primitive data structures are the most basic data structures available in a programming language, such as integers, floating-point numbers, characters and Booleans.

Non-primitive data structures are complex data structures that are built using primitive data types, such as arrays, linked lists, stacks, queues, trees, graphs and hash tables.

The choice of data structure for a particular task depends on the type and amount of data to be

processed, the operations that need to be performed on the data and the efficiency requirements of the program. Efficient use of data structures can greatly improve the performance of a program, making it faster and more memory-efficient. A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks.

The choice of a good data structure makes it possible to perform a variety of critical operations effectively. An efficient data structure also uses minimum memory space and execution time to process the structure.

Data Structure is a systematic way to organize data in order to use it efficiently.

Following terms are the foundation terms of a data structure.

- **Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.

- **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

1. Efficient data processing: Data structures provide a way to organize and store data in a way that allows for efficient retrieval, manipulation and storage of data. For example, using a hash table to store data can provide constant-time access to data.

2. Memory management: Proper use of data structures can help to reduce memory usage and

optimize the use of resources. For example, using dynamic arrays can allow for more efficient use of memory than using static arrays.

3. Code reusability: Data structures can be used as building blocks in various algorithms and programs, making it easier to reuse code.

4. Abstraction: Data structures provide a level of abstraction that allows programmers to focus on the logical structure of the data and the operations that can be performed on it, rather than on the details of how the data is stored and manipulated.

5. Algorithm design: Many algorithms rely on specific data structures to operate efficiently. Understanding data structures is crucial for designing and implementing efficient algorithms.

6. Data Search − Consider an inventory of 1 million (106) items of a store. If the application is to search an item, it has to search an item in 1 million (106) items every time slowing down the search. As data grows, search will become slower.

7. Processor Speed − Processor speed although being very high, falls limited if the data grows to billion records.

8. Multiple Requests − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

Importance of Data Structures in Programming

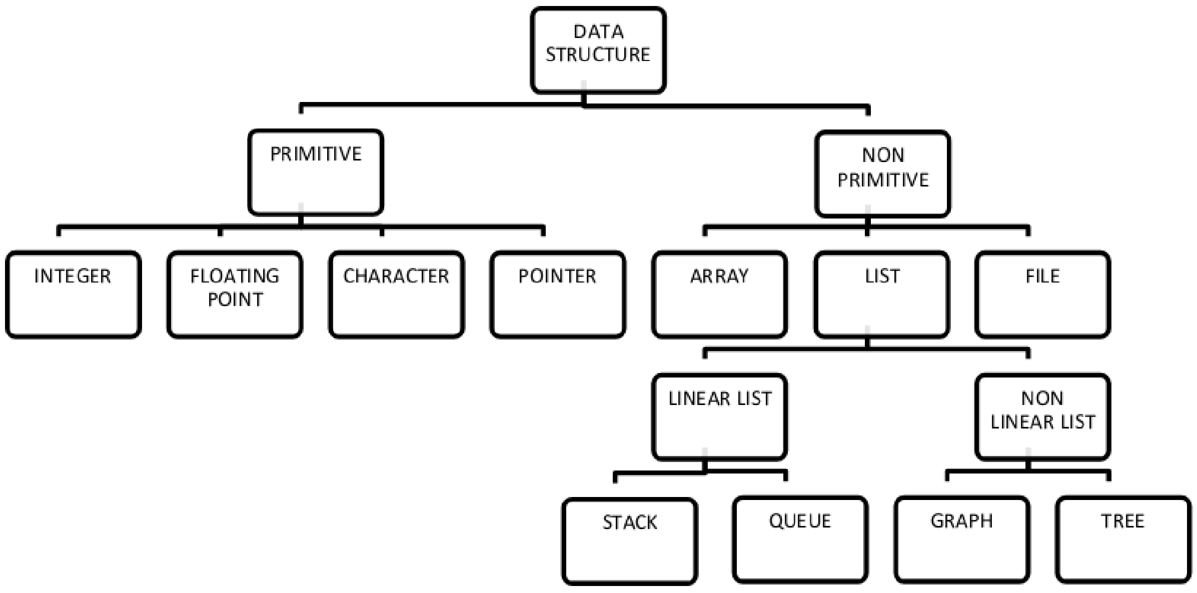
- **Efficient data access and manipulation:** Data structures enable quick access and manipulation of data. For example, an array allows constant-time access to elements using their index, while a hash table allows fast access to elements based on their key. Without data structures, programs would have to search through data sequentially, leading to slow performance.
- **Memory management:** Data structures allow efficient use of memory by allocating and deallocating memory dynamically. For example, a linked list can dynamically allocate memory for each element as needed, rather than allocating a fixed amount of memory upfront. This helps avoid memory wastage and enables efficient memory management.
- **Code reusability:** Data structures can be reused across different programs and projects. For example, a generic stack data structure can be used in multiple programs that require LIFO (Last-In-First-Out) functionality, without having to rewrite the same code each time.
- **Optimization of algorithms:** Data structures help optimize algorithms by enabling efficient data access and manipulation. For example, a binary search tree allows fast searching and insertion of elements, making it ideal for implementing searching and sorting algorithms.
- **Scalability:** Data structures enable programs to handle large amounts of data effectively. For example, a hash table can store large amounts of data while providing fast access to elements based on their key.

Basic Terminology

- **Data** − Data are values or set of values.
- **Data Item** − Data item refers to single unit of values.

- **Group Items** – Data items that are divided into sub items are called as Group Items.
- **Elementary Items** – Data items that cannot be divided are called as Elementary Items.
- **Attribute and Entity** – An entity is that which contains certain attributes or properties, which may be assigned values.
- **Entity Set** – Entities of similar attributes form an entity set.
- **Field** – Field is a single elementary unit of information representing an attribute of an entity.
- **Record** – Record is a collection of field values of a given entity.
- **File** – File is a collection of records of the entities in a given entity set.

<u>Classification of Data Structures</u>



<u>Data structures can broadly be classified into two main categories:</u>

<u>1. Primitive Data Structures</u>

Primitive data structures are the most basic types of data structures provided by programming languages. They include:

- **Integer**: Whole numbers (e.g., 1, 2, 100).
- **Float**: Numbers with decimal points (e.g., 3.14, 0.001).
- **Character**: Single characters (e.g., ‘A’, ‘B’).
- **Boolean**: Logical values (true or false).

Sample Program in C++ (Primitive Data Types)

```
#include <iostream>
using namespace std;
int main() {
   int num = 10;          // Integer
   float pi = 3.14;       // Float
   char letter = 'A';     // Character
   bool isHappy = true;   // Boolean
   cout << "Integer: " << num << endl;
   cout << "Float: " << pi << endl;
   cout << "Character: " << letter << endl;
   cout << "Boolean: " << isHappy << endl;
   return 0;
}
```

2. Non-Primitive Data Structures

Non-primitive data structures are more complex and are built using primitive data types. They are further divided into:

a. **Linear Data Structures**

In linear data structures, elements are arranged sequentially, and each element has a unique predecessor and successor (except the first and last elements).

- **Array**: A collection of elements of the same type, stored in contiguous memory locations.
- **Linked List**: A collection of nodes where each node contains data and a pointer to the next node.
- **Stack**: A collection of elements following the Last In First Out (LIFO) principle.
- **Queue**: A collection of elements following the First In First Out (FIFO) principle.

Sample Program in C++ (Array)

```
#include <iostream>
using namespace std;
int main() {
    int arr[5] = {1, 2, 3, 4, 5}; // Array of integers
    cout << "Array elements:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

b. Non-Linear Data Structures

In non-linear data structures, elements are not arranged sequentially but follow a hierarchical or interconnected structure.

- **Tree**: A hierarchical structure where each node has a parent node and zero or more child nodes.
- **Graph**: A set of nodes (vertices) connected by edges, representing relationships.

Sample Program in C++ (Basic Tree Traversal)

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = right = nullptr;
    }
};
void inOrder(Node* root) {
    if (root == nullptr) return;
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(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);
    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;
    return 0;
}
```

c. File Structures

Specialized data structures designed for storing data in secondary storage for efficient retrieval and updates. Examples include B-trees and hash files.

Applications of Data Structures:

Arrays: Arrays are used to store a collection of homogeneous elements contiguous memory locations. They are commonly used to implement other data structures, such as stacks and queues, and to represent matrices and tables.

Linked lists: Linked lists are used to store a collection of heterogeneous elements with dynamic memory allocation. They are commonly used to implement stacks, queues and hash tables.

Trees: Trees are used to represent hierarchical data structures, such as file systems, organization charts, and network topologies. Binary search trees are commonly used to implement dictionaries and symbol tables.

Graphs: Graphs are used to represent complex relationships between data elements, such as social networks, transportation networks and computer networks. They are commonly used to implement shortest path algorithms and graph traversal algorithms.

Hash tables: Hash tables are used to implement associative arrays, which store key-value pairs. They provide fast access to data elements based on their keys.

- Stacks: Stacks are used to store a collection of elements in a last-in-first-out (LIFO) order. They are commonly used to implement undo-redo functionality, recursive function calls and expression evaluation.
- Queues: Queues are used to store a collection of elements in a first-in-first-out (FIFO) order. They are commonly used to implement waiting lines, message queues and job scheduling.

Common operations on various Data Structures:

Data Structure is the way of storing data in computer's memory so that it can be used easily and efficiently. There are different data-structures used for the storage of data. It can also be defined as a mathematical or logical model of a particular organization of data items. The representation of particular data structure in the main memory of a computer is called as storage structure.

For Examples: Array, Stack, Queue, Tree, Graph, etc.

Operations on different Data Structure:

There are different types of operations that can be performed for the manipulation of data in every data structure. Some operations are explained and illustrated below:

- **Traversing:** Traversing a Data Structure means to visit the element stored in it. It visits data in a systematic manner. This can be done with any type of DS.
- **Searching:** Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found. Searching is the operation which we can performed on data-structures like array, linked-list, tree, graph, etc.
- **Insertion:** It is the operation which we apply on all the data-structures. Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure. It is unsuccessful in some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element. The insertion has the same name as an insertion in the data-structure as an array, linked-list, graph, tree. In stack, this operation is called Push. In the queue, this operation is called Enqueue.

- **Deletion:** It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure. The deletion has the same name as a deletion in the data- structure as an array, linked-list, graph, tree, etc. In stack, this operation is called Pop. In Queue this operation is called Dequeue.
- **Create:** It reserves memory for program elements by declaring them. The creation of data structure can be done during
    - Compile-time
    - Run-time.
- **Selection:** It selects specific data from present data. You can select any specific data by giving condition in loop.
- **Update:** It updates the data in the data structure. You can also update any specific data by giving some condition in loop like select approach.
- **Sort:** Sorting data in a particular order (ascending or descending). We can take the help of many sorting algorithms to sort data in less time. Example: bubble sort which takes O (n^2) time to sort data. There are many algorithms present like merge sort, insertion sort, selection sort, quick sort, etc.
- **Merge:** Merging data of two different orders in a specific order may ascend or descend. We use merge sort to merge sort data.
- **Split Data:** Dividing data into different sub-parts to make the process complete in less time.

Algorithm

Specifications:

What is an Algorithm? Algorithm Basics

The word **Algorithm** means" A set of finite rules or instructions to be followed in calculations or other problem-solving operations" or" A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations". Therefore, Algorithm refers to a sequence of finite steps to solve a particular problem.

Use of the Algorithms:

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

Computer Science: Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.

Mathematics: Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.

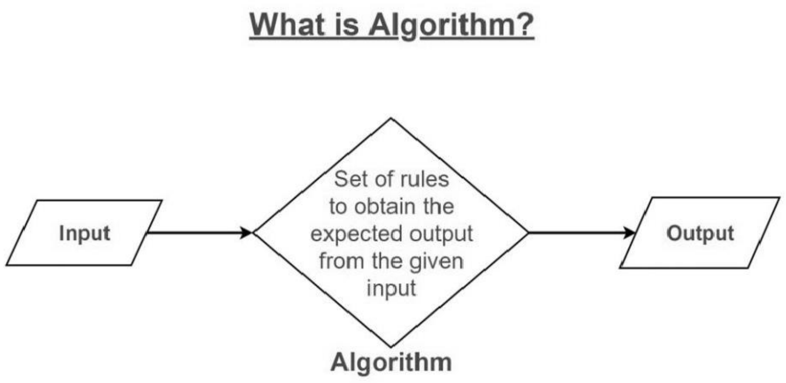
Operations Research: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.

Artificial Intelligence: Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing and decision-making.

Data Science: Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance and healthcare.

These are just a few examples of the many applications of algorithms. The use of algorithms is continually expanding as new technologies and fields emerge, making it a vital component of modern society.

Algorithms can be simple and complex depending on what you want to achieve.


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

It can be understood by taking the example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and executes them one by one, in the given sequence. The result thus obtained is the new dish is coo ed perfectly. Every time you use your phone, computer, laptop or calculator you are using Algorithms. Similarly, algorithms help to do a task programming to get the expected output.

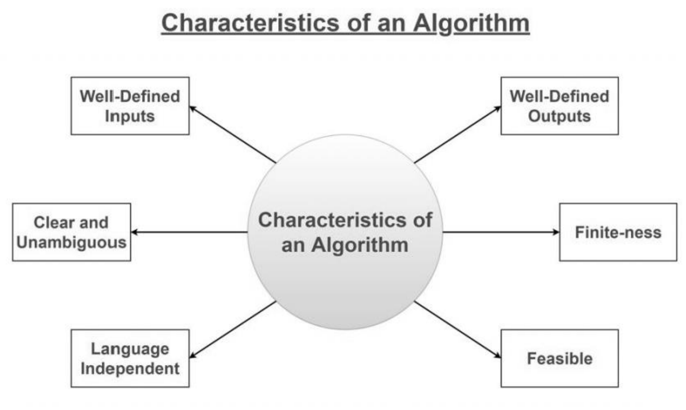
The Algorithm designed is language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

What is the need for algorithms?

Algorithms are necessary for solving complex problems efficiently and effectively.

1. They help to automate processes and make them more reliable, faster and easier to perform.
2. Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
3. They are used in various field such as mathematics, computer science, engineering, finance and 22many others to optimize cases, analyse data, make predictions and provide solutions to problems.

What are the characteristics of an Algorithm?



As one would not follow any written instructions to cook the recipe, but only the standard one. Similarly, not all written instructions for programming is an algorithm. In order for some instructions to be an algorithm, it must have the following characteristics:

- **Clear and Unambiguous**: The algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
- **Input**: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.
- **Output**: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.
- **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one

can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.

- **Finiteness:** An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
- **Effectiveness:** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Advantages of Algorithms:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms.

How to Design an Algorithm?

In order to write an algorithm, the following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm i.e. clear problem definition.
2. The **constraints** of the problem must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem is solved.
5. The **solution** to this problem, is within the given constraints.

How to express an Algorithm?

1. **Natural Language:** Here we express the Algorithm in natural English language. It is too hard to understand the algorithm from it.
2. **Flow Chart:** Here we express the Algorithm by making graphical/pictorial representation of it. It is easier to understand than Natural Language.
3. **Pseudo Code:** Here we express the Algorithm in the form of annotations and informative text written in plain English which is very much similar to the real code but as it has no syntax like any of the programming language, it can't be compiled or interpreted by the computer. It is the best way to express an algorithm because it can be understood by even a layman with some school level programming knowledge.

Recursion:

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), In order/Preorder/Post order Tree Traversals, DFS of Graph, etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So, we can say that every time the function calls itself with a simpler version of the original problem.

Need of Recursion:

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number.

Properties of Recursion:

Performing the same operations multiple times with different inputs. In every step, we try smaller inputs to make the problem smaller.

Base condition is needed to stop the recursion otherwise infinite loop will occur.

Algorithm: Steps

The algorithmic steps for implementing recursion in a function are as follows:

Step1 - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 - Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

Step4 - Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

How are recursive functions stored in memory?

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know the factorial of (n- 1). The base case for factorial would be n = 0. We return 1 when n = 0.

1. Tower of Hanoi:

   Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod A. The objective of the puzzle is to move the entire stack to another rod (here considered C), obeying the following simple rules:

Rules:

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

1. Only one disk can be moved among the towers at any given time.

2. Only the "top" disk can be removed.
3. No large disk can sit over a small disk.

Examples:

Input: 2

Output:

Disk 1 moved from A to B Disk 2 moved from A to C Disk 1 moved from B to C

Input: 3

Output:

Disk 1 moved from A to C Disk 2 moved from A to B Disk 1 moved from C to B Disk 3 moved from A to C Disk 1 moved from B to A Disk 2 moved from B to C Disk 1 moved from A to C

COMPARISON BETWEEN ITERATIVE AND RECURSIVE FUNCTIONS-

Parameter	**Recursion**	**Iteration**
Definition	Recursion involves a recursive function which calls itself repeatedly until a base condition is not reached.	Iteration involves the usage of loops through which a set of statements are executed repeatedly until the condition is not false.
Terminatio n Condition	Here termination condition is a base case defined within the recursive function.	Termination condition is the condition specified in the definition of the loop.
Infinite Case	If base case is never reached it leads to infinite recursion leading to memory crash.	If condition is never false, it leads to infinite iteration with computers CPU cycle being used repeatedly.
Memory Usage	Recursion uses stack area to store the current state of the function, due to which memory usage is high.	Iteration uses the permanent storage area only for the variables involved in its code block, hence memory usage is less.
Code Size	Code size is comparatively smaller.	Code size is comparatively larger.
Performan ce	Since stack are is used to store and restore the state of recursive function after every function call, performance is comparatively slow.	Since iteration does not have to keep re-initializing its component variables and neither has to store function states, the performance is fast.
Memory Runout	There is a possibility of running out of memory, since for each function call stack area gets used.	There is no possibility of running out of memory as stack area is not used.
Overhead	Recursive functions involve extensive overhead, as for each	There is no overhead in Iteration.

	function call the current state, parameters etc. have to be pushed and popped out from stack.	
Applications	Factorial, Fibonacci Series etc.	Finding average of a data series, creating multiplication table etc.

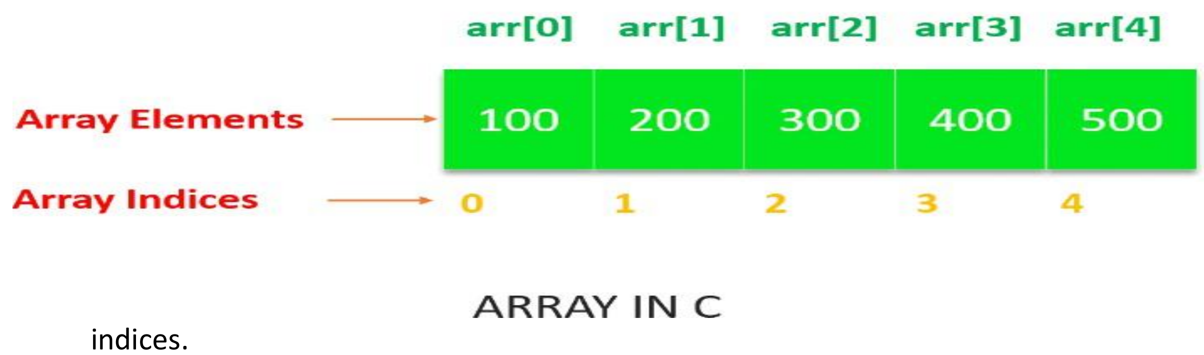
ARRAYS:

Definition- An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations

Types of arrays-

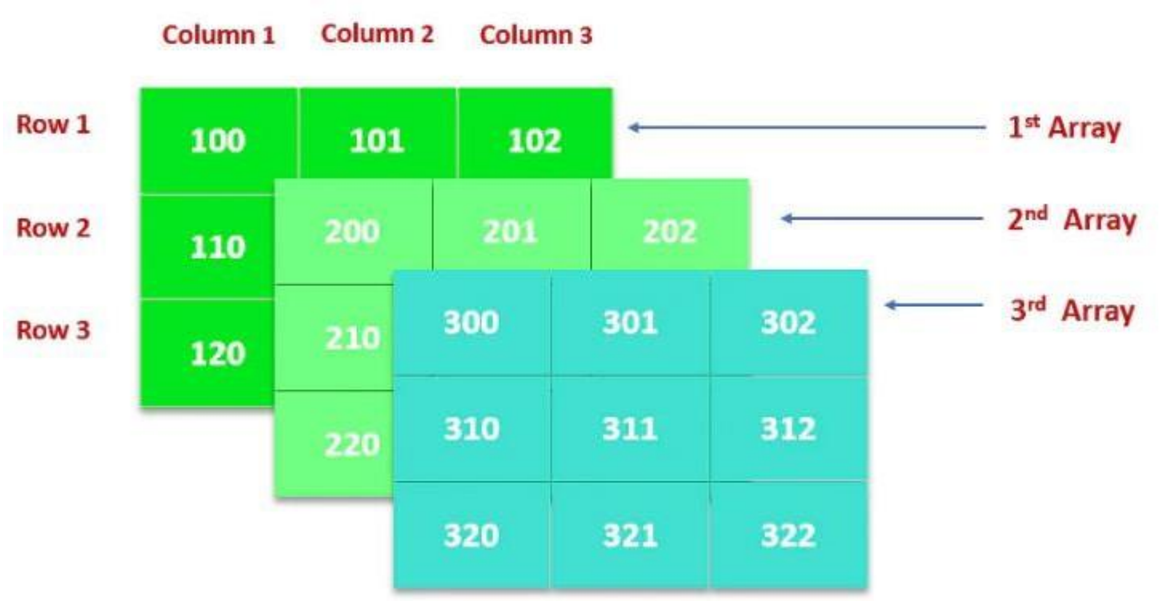
Array in Care of two types; Single dimensional arrays and Multidimensional arrays.

- **Single Dimensional Arrays:** Single dimensional array or 1-D array is the simplest form of arrays that can be found in C. This type of array consists of elements of similar types and these elements can be accessed through their



  indices.

- **Multi-dimensional Arrays:** The most common type of multi-dimensional array that is used in the C language is a 2-D array. However, the number of dimensions can be more than 2 depending upon the compiler of the user's system. These arrays consist of elements that are array themselves.



OPERATION ON ARRAYS

There are a number of operations that can be performed on an array which are:

1. Traversal
2. Sorting
3. Insertion
4. Deletion
5. Searching
6. Merging

Traversal: Traversal means accessing each array element for a specific purpose, either to perform an operation on them.

Sorting: It is a process of arranging the elements in the array.

Insertion: it is a process of adding a new element to the array in the specified location.

Deletion: it is a process of removing an element from the array from the specified location.

Searching: it is a process of finding an element in an array

Merging: it is a process of combining the elements of two array.

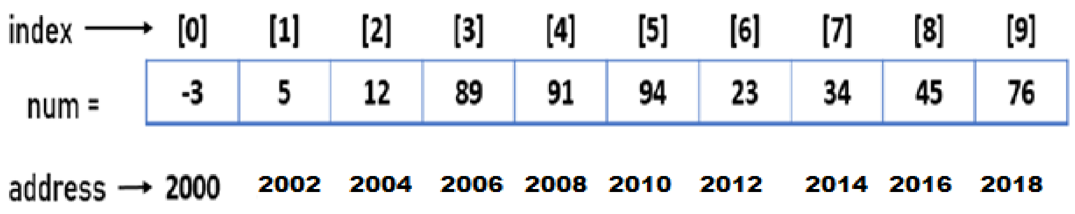
ABSTRACT DATA TYPE(ADT): Abstract Data Type (ADT) is a data type, where only behaviour is defined but not implementation. Examples: **Array, List, Queue, Stack, Tree** are ADTs.

MEMORY REPRESENTATION

One dimensional array-

Let’s first declare and initialise an one-dimensional array num with

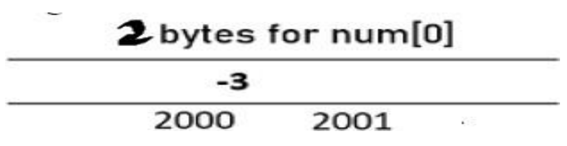
10 elements int num[] = {-3, 5, 12, 89, 91, 94, 23, 34, 45, 76};



Memory Representation of an array with elements, address and indexes

Like other variables and constants an array also requires memory blocks to store the elements. An array is a collection of similar elements, so it requires contiguous memory blocks. Look at the image depicted above , at the bottom of the image the addresses of each elements are displayed. Now let’s understand the addressing and indexing of the **num** array.

Addressing- The type of **num** is **int**, so each elements of the **num** array will take 2 bytes of memory. For example, consider the starting address of first memory block is 2000. So the next elements of the array will get next contiguous 4 bytes memory block and the memory address will be 2002 (as 2000 + 2 bytes) and so on for other elements. In the above figure, the addresses are displayed here, are the starting address of each memory block assigned to individual element of the array.



2 bytes memory block assigned to each element

What is Indexing

Indexing is a process to get the location of an element in the array. So, when we write **num[0]**, it informs the compiler about the location of the first element.

 In C, array indexing is started from 0 for the first element. So, 0 will be passed with subsscripting operator **[ ]**, to access the first element of the array.

num[0] = 9; //it updates the value of first element

An index value must be greater than or equal to 0

num[-1] = 4; //an illegal; statement and raised to run-time error.

The highest index of an array is equal to **size - 1**, size is the number of elements in the array.

How to calculate the address of the 1D array

Address of A[i] = Base_Address + W(i – Lower_Bound)

Where **Base Address** is the address of the first element in an array.

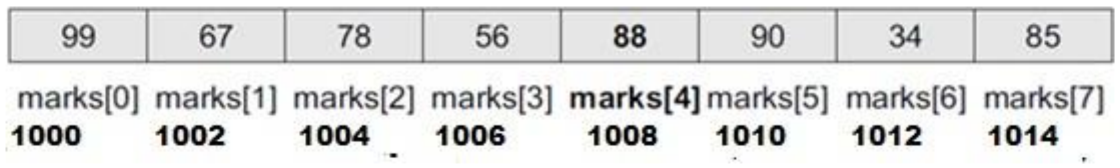
W= is the weight (size) of a data type.

i = is the index of array element to be calculated.

Lower-bound= is the index of the first element.

Ex: Given an array int marks[] = {99,67,78,56,88,90,34,85}, calculate the address of marks[4] if the base address = 1000.

Solution :



99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	**1002**	**1004**	**1006**	**1008**	**1010**	**1012**	**1014**

We know that storing an integer value requires 2 bytes, therefore, its size is 4 bytes.

Address of A[i] = Base Address + W(i - Lower_Bound)

Marks[4]=1000+2(4-0)

=1000+ 2(4)

= **1008** [Ans]

Sorting Techniques

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following is some of the examples of sorting in real-life scenarios

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

A sorting algorithm is an algorithm that puts elements of a list in a certain order. • Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists.

Why Sorting?

- “When in doubt, sort” –one of the principles of algorithm design. Sorting used as a subroutine in many of the algorithms:
- Searching in databases: we can do binary search on sorted data
- A large number of computer graphics and computational geometry problems
- Closest pair, element uniqueness, frequency distribution
- A large number of algorithms developed representing different algorithm design techniques.
- A lower bound for sorting Ω(n log n)is used to prove lower bounds of other problems
- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$ where **n** is the number of items.

Bubble Sort Algorithm

- Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.
- We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

Step 1 − Check if the first element in the input array is greater than the next element in the array.

Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step 3 − Repeat Step 2 until we reach the end of the array.

Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step 5 − The final output achieved is the sorted array.

Algorithm: Sequential-Bubble-Sort (A)

```
fori ← 1 to length [A] do
```

```
for j ← length [A] down-to i +1 do
  if A[A] < A[j-1] then
    Exchange A[j] ⟷ A[j-1]
```

Pseudocode

- We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.
- To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.
- Pseudocode of bubble sort algorithm can be written as follows −

```
Void bubbleSort(int numbers[], intarray_size)
{
        inti, j, temp;
        for (i = (array_size - 1); i>= 0; i--)
        for (j = 1; j <= i; j++)
        if (numbers[j-1] > numbers[j]){
          temp = numbers[j-1];
          numbers[j-1] = numbers[j];
          numbers[j] = temp;
       }
       }
```

Analysis

Here, the number of comparisons are

$1 + 2 + 3 + ... + (n - 1) = n(n - 1)/2 = O(n2)$

Clearly, the graph shows the $\mathbf{n^2}$ nature of the bubble sort.

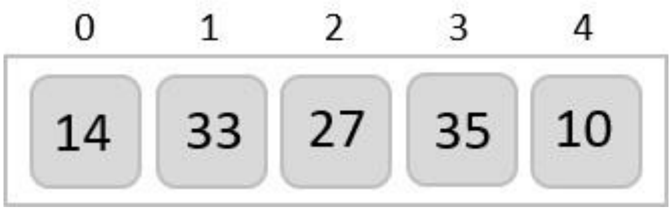
In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

Memory Requirement

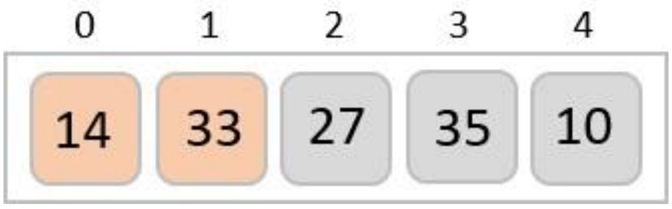
From the algorithm stated above, it is clear that bubble sort does not require extra memory.

Example

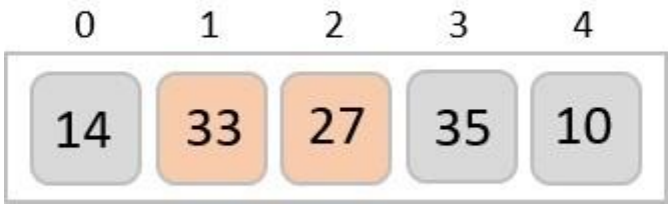
We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.



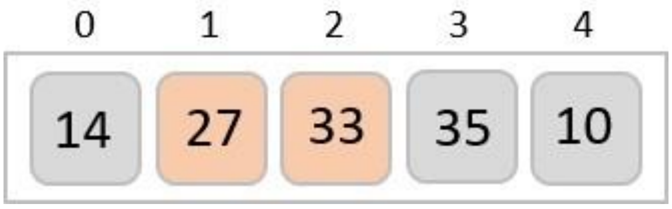
Bubble sort starts with very first two elements, comparing them to check which one is greater.



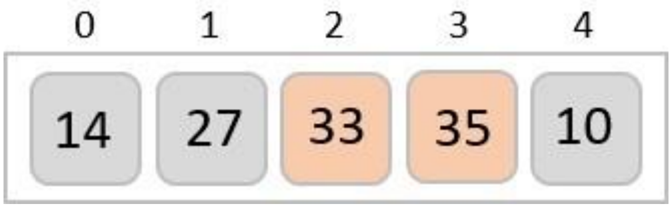
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



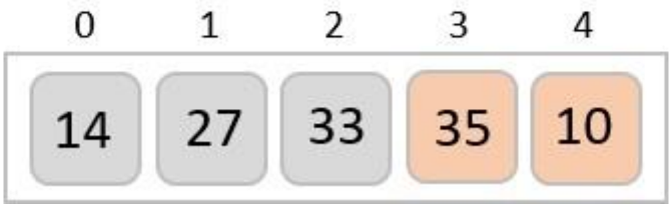
We find that 27 is smaller than 33 and these two values must be swapped.



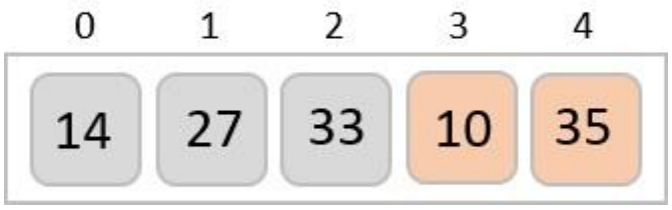
Next we compare 33 and 35. We find that both are in already sorted positions.



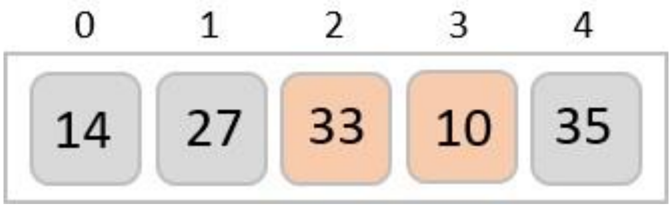
Then we move to the next two values, 35 and 10.

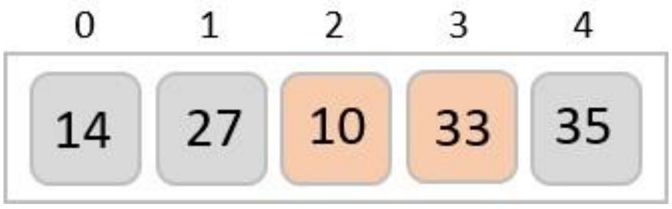


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

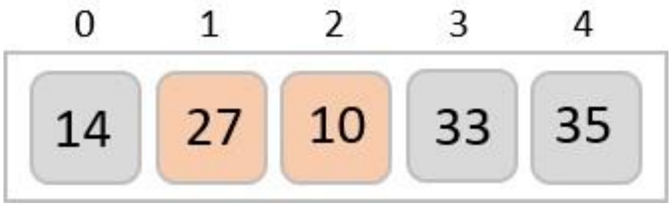


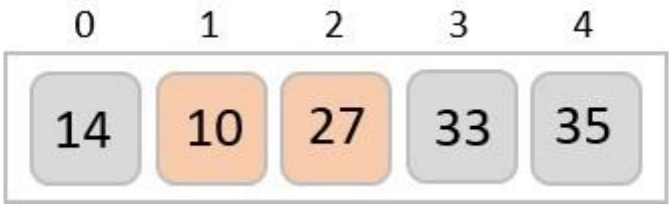
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

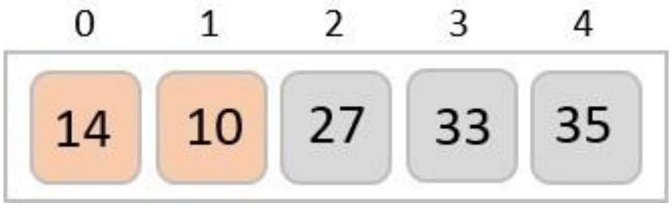


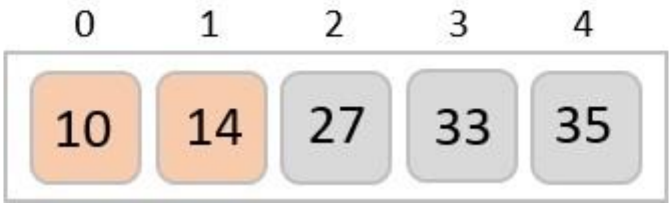


Notice that after each iteration, at least one value moves at the end.

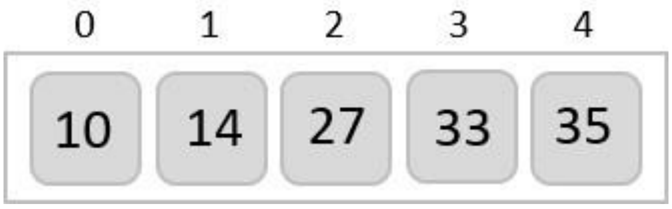








And when there's no swap required, bubble sort learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

```
#include<iostream>
using namespace std;
void bubbleSort(int *array, int size){
  for(int i = 0; i<size; i++) {
    int swaps = 0; //flag to detect any swap is there or not
    for(int j = 0; j<size-i-1; j++) {
      if(array[j] > array[j+1]) { //when the current item is bigger than next
        int temp;
        temp = array[j];
        array[j] = array[j+1];
        array[j+1] = temp;
        swaps = 1; //set swap flag
      }
    }
    if(!swaps)
      break; // No swap in this pass, so array is sorted
  }
}
int main(){
  int n;
  n = 5;
  int arr[5] = {67, 44, 82, 17, 20}; //initialize an array
  cout << "Array before Sorting: ";
  for(int i = 0; i<n; i++)
    cout << arr[i] << " ";
  cout << endl;
  bubbleSort(arr, n);
  cout << "Array after Sorting: ";
  for(int i = 0; i<n; i++)
    cout << arr[i] << " ";
  cout << endl;
}
```

Output

```
Array before Sorting: 67 44 82 17 20
Array after Sorting: 17 20 44 67 82
```

Selection Sort Algorithm

Selection sort is a simple sorting algorithm. This sorting algorithm, like insertion sort, is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.
This algorithm is not suitable for large data sets as its average and worst-case complexities are of **$O(n^2)$**, where **n** is the number of items.

Selection Sort Algorithm

This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and we continue the process in this way until the entire array is sorted.

1. Set MIN to location 0.
2. Search the minimum element in the list.
3. Swap with value at location MIN.
4. Increment MIN to point to next element.
5. Repeat until the list is sorted.

Pseudocode

```
Algorithm: Selection-Sort (A)
fori← 1 to n-1 do
  min j ←i;
  min x ← A[i]
  for j ←i + 1 to n do
    if A[j] < min x then
      min j ← j
      min x ← A[j]
  A[min j] ← A [i]
  A[i] ← min x
```

Analysis

Selection sort is among the simplest of sorting techniques and it works very well for small files. It has a quite important application as each item is actually moved at the most once.
Section sort is a method of choice for sorting files with very large objects (records) and small keys. The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.

Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test if $A[j]$ < **A[j] < min x** is executed exactly the same number of times in every case.

Selection sort spends most of its time trying to find the minimum element in the unsorted part of the array. It clearly shows the similarity between Selection sort and Bubble sort.

- Bubble sort selects the maximum remaining elements at each stage, but wastes some effort imparting some order to an unsorted part of the array.
- Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

For each **i** from **1** to **n - 1**, there is one exchange and **n - i** comparisons, so there is a total of **n - 1** exchanges and

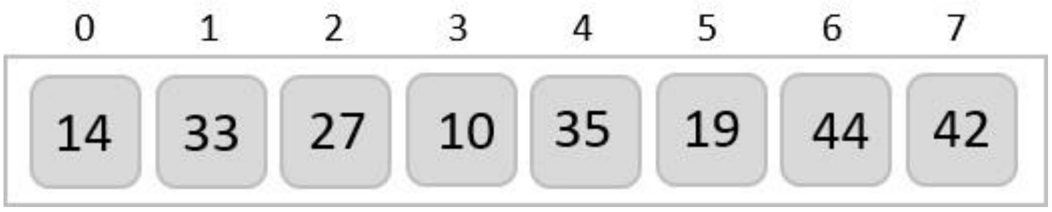
(n − 1) + (n − 2) + ...+2 + 1 = n(n − 1)/2 comparisons.

These observations hold, no matter what the input data is.

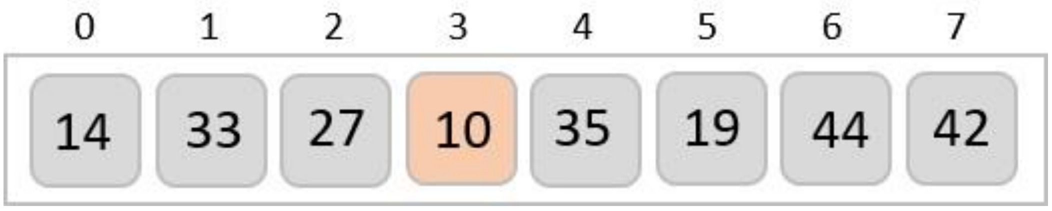
In the worst case, this could be quadratic, but in the average case, this quantity is **O(n log n)**. It implies that the **running time of Selection sort is quite insensitive to the input**.

Example

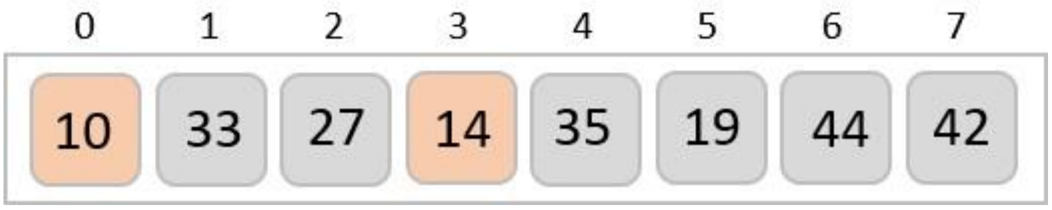
Consider the following depicted array as an example.



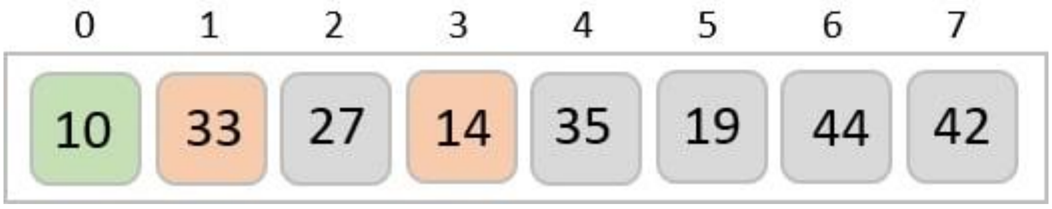
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



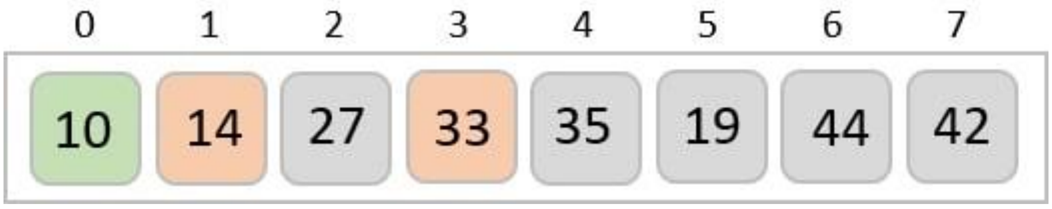
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

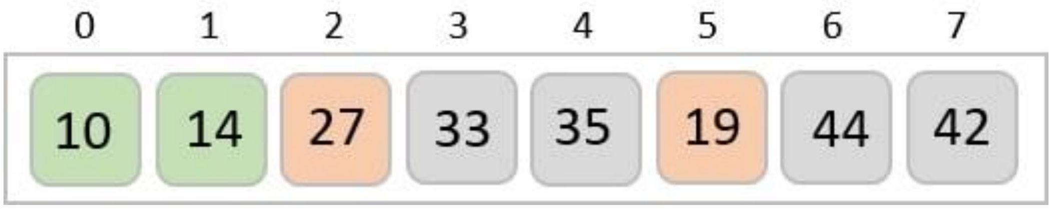


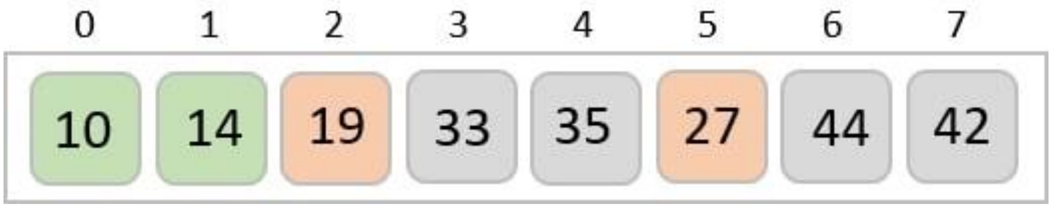
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

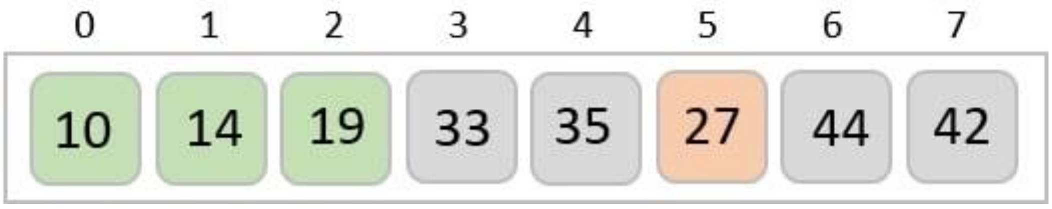


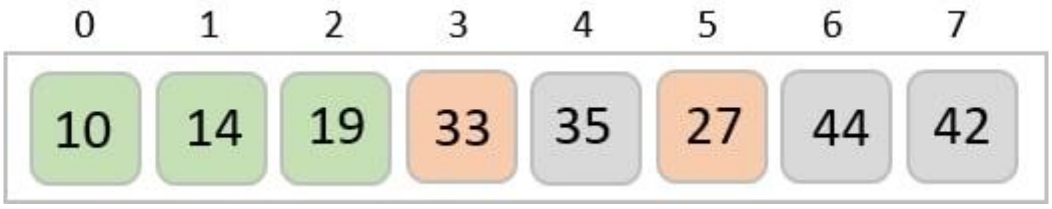
After two iterations, two least values are positioned at the beginning in a sorted manner.

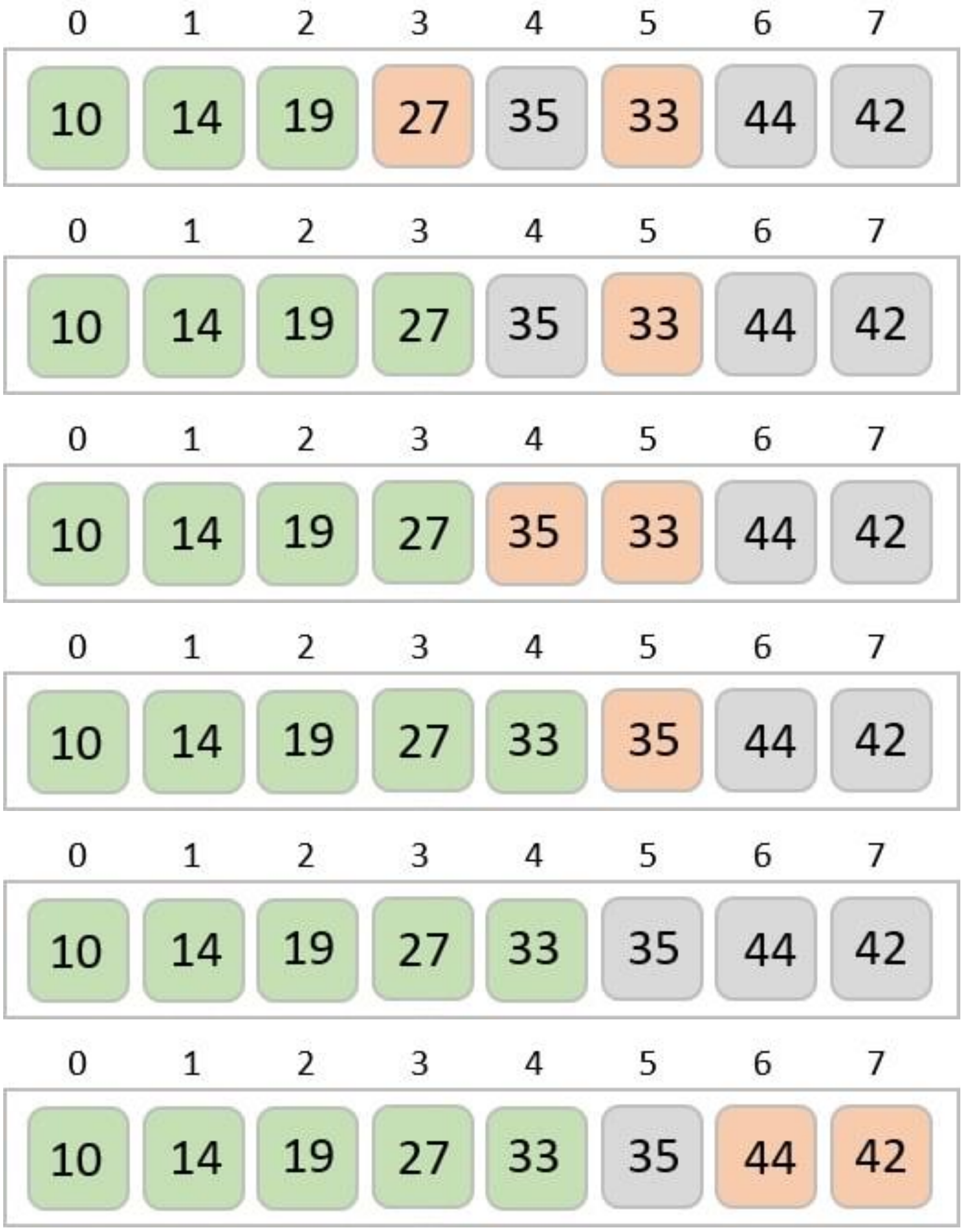
The same process is applied to the rest of the items in the array –

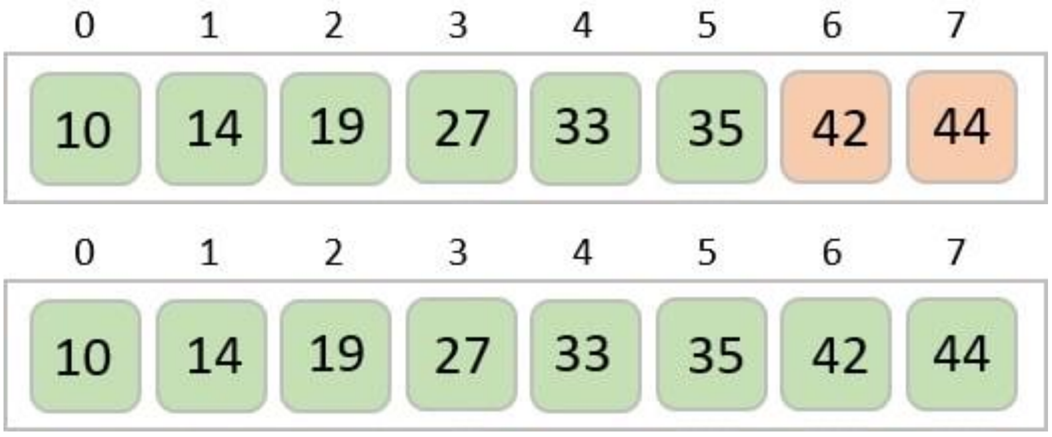









0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 33 35 44 42
0 1 2 3 4 5 6 7
10 14 19 27 33 35 44 42
0 1 2 3 4 5 6 7
10 14 19 27 33 35 44 42



Implementation

The selection sort algorithm is implemented in four different programming languages below. The given program selects the minimum number of the array and swaps it with the element in the first index. The second minimum number is swapped with the element present in the second index. The process goes on until the end of the array is reached.

```
#include<iostream>
using namespace std;
void swapping(int &a, int &b) {  //swap the content of a and b
  int temp;
  temp = a;
  a = b;
  b = temp;
}
void selectionSort(int *array, int size){
  int i, j, imin;
  for(i = 0; i<size-1; i++) {
    imin = i; //get index of minimum data
    for(j = i+1; j<size; j++)
      if(array[j] < array[imin])
        imin = j;

    //placing in correct position
    swap(array[i], array[imin]);
  }
}
int main(){
  int n;
  n = 5;
  int arr[5] = {12, 19, 55, 2, 16}; // initialize the array
  cout << "Array before Sorting: ";
  for(int i = 0; i<n; i++)
```

```
      cout << arr[i] << " ";
   cout << endl;
   selectionSort(arr, n);
   cout << "Array after Sorting: ";
   for(int i = 0; i<n; i++)
      cout << arr[i] << " ";
   cout << endl;
}
```

Output

```
Array before Sorting: 12 19 55 2 16
Array after Sorting: 2 12 16 19 55
```

Comparison of Bubble Sort and Selection Sort

Feature	Bubble Sort	Selection Sort
Time Complexity	O(n^2) (average)	O(n^2) (average)
Space Complexity	O(1)	O(1)
Stability	Stable	Not Stable
Best Case	O(n) (sorted array)	O(n^2)
Use Case	Small datasets, educational purposes	Small datasets, educational purposes

Data Structures - Searching Algorithms

In the previous section, we have discussed various Sorting Techniques and cases in which they can be used. However, the main idea behind performing sorting is to arrange the data in an orderly way, making it easier to search for any element within the sorted data.
Searching is a process of finding a particular record, which can be a single element or a small chunk, within a huge amount of data. The data can be in various forms: arrays, linked lists, trees, heaps, and graphs etc. With the increasing amount of data nowadays, there are multiple techniques to perform the searching operation.

Searching Algorithms in Data Structures

Various searching techniques can be applied on the data structures to retrieve certain data. A search operation is said to be successful only if it returns the desired element or data; otherwise, the searching method is unsuccessful.
There are two categories these searching techniques fall into. They are −

- Sequential Searching
- Interval Searching

Sequential Searching

As the name suggests, the sequential searching operation traverses through each element of the data sequentially to look for the desired data. The data need not be in a sorted manner for this type of search.

Example − Linear Search

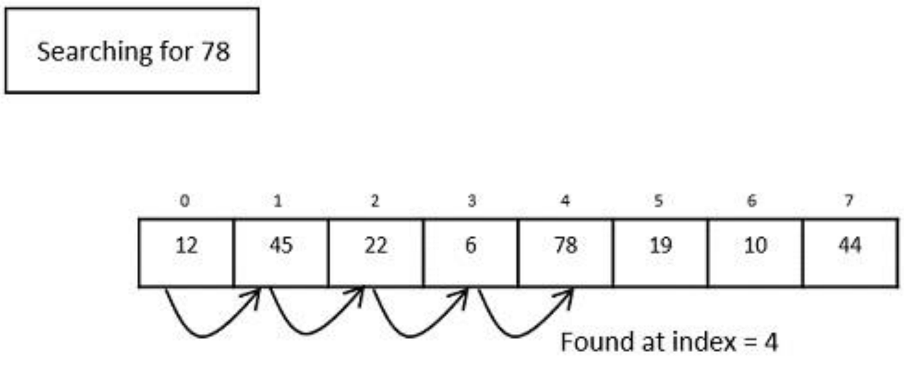
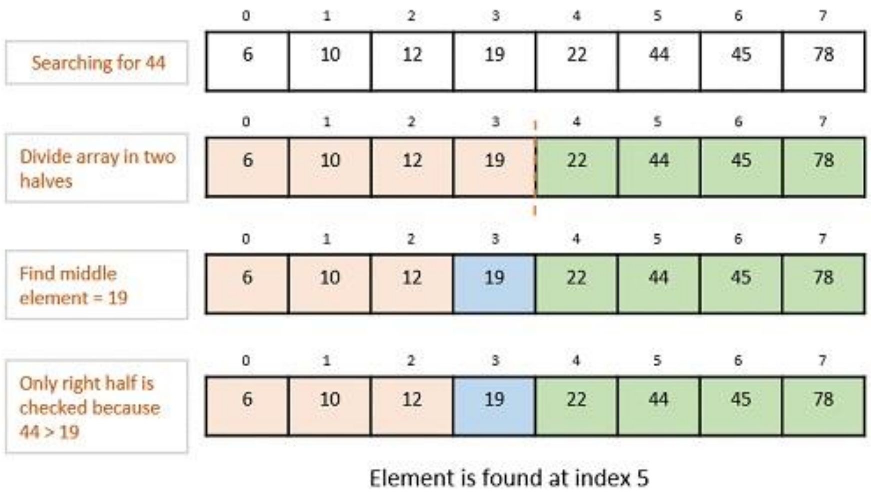


Fig. 1: Linear Search Operation

Interval Searching

Unlike sequential searching, the interval searching operation requires the data to be in a sorted manner. This method usually searches the data in intervals; it could be done by either dividing the data into multiple sub-parts or jumping through the indices to search for an element.

Example − Binary Search, Jump Search etc.

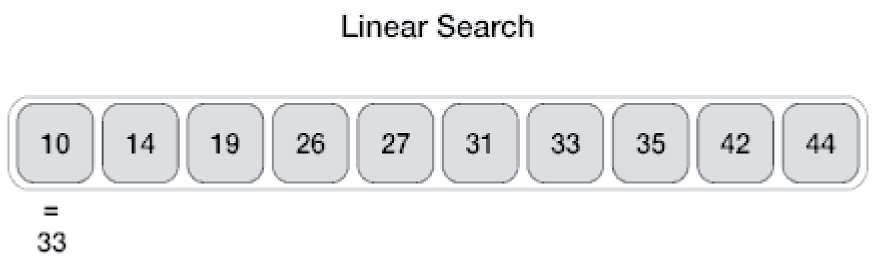


Searching for 44
Divide array in two halves
Find middle element = 19
Only right half is checked because 44 > 19
0 1 2 3 4 5 6 7
6 10 12 19 22 44 45 78
Element is found at index 5

Fig. 2: Binary Search Operation

Linear Search Algorithm

Linear search is a type of sequential searching algorithm. In this method, every element within the input array is traversed and compared with the key element to be found. If a match is found in the array the search is said to be successful; if there is no match found the search is said to be unsuccessful and gives the worst-case time complexity.
For instance, in the given animated diagram, we are searching for an element 33. Therefore, the linear search method searches for it sequentially from the very first element until it finds a match. This returns a successful search.



In the same diagram, if we have to search for an element 46, then it returns an unsuccessful search since 46 is not present in the input.

Linear Search Algorithm

The algorithm for linear search is relatively simple. The procedure starts at the very first index of the input array to be searched.
Step 1 − Start from the 0th index of the input array, compare the key value with the value present in the 0th index.
Step 2 − If the value matches with the key, return the position at which the value was found.
Step 3 − If the value does not match with the key, compare the next element in the array.
Step 4 − Repeat Step 3 until there is a match found. Return the position at which the match was found.
Step 5 − If it is an unsuccessful search, print that the element is not present in the array and exit the program.

Pseudocode

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

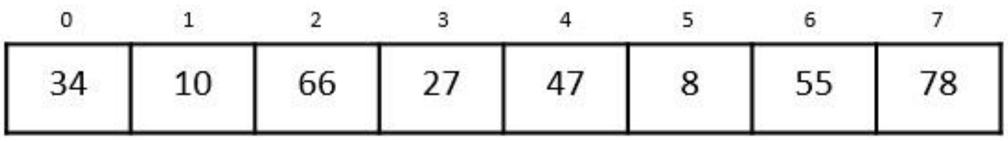
Analysis

Linear search traverses through every element sequentially therefore, the best case is when the element is found in the very first iteration. The best-case time complexity would be **O(1)**.

However, the worst case of the linear search method would be an unsuccessful search that does not find the key value in the array, it performs n iterations. Therefore, the worst-case time complexity of the linear search algorithm would be **O(n)**.

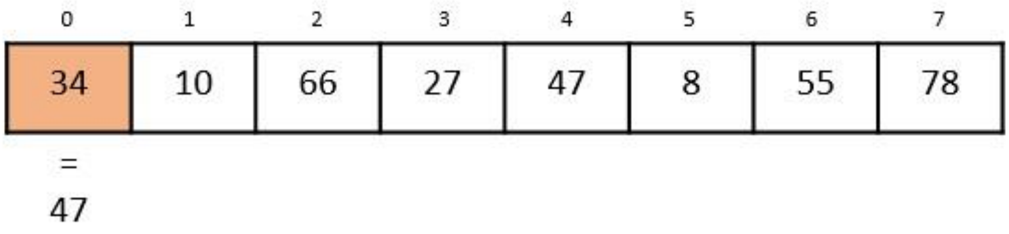
Example

Let us look at the step-by-step searching of the key element (say 47) in an array using the linear search method.



Step 1

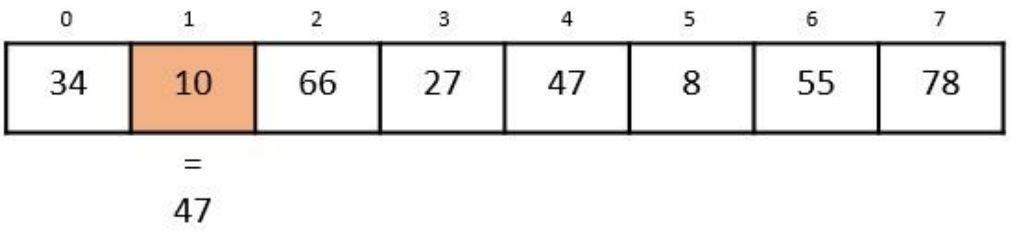
The linear search starts from the 0th index. Compare the key element with the value in the 0th index, 34.



However, 47 ≠ 34. So it moves to the next element.

Step 2

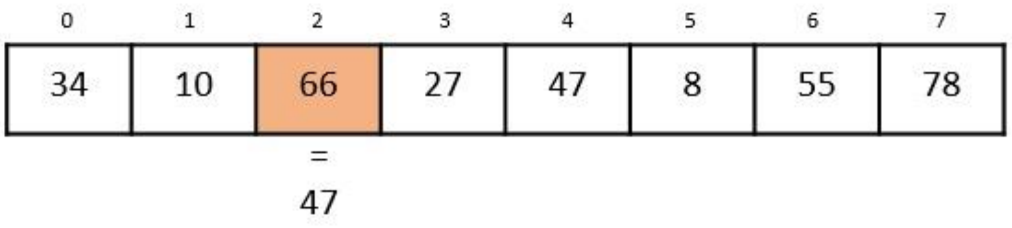
Now, the key is compared with value in the 1st index of the array.



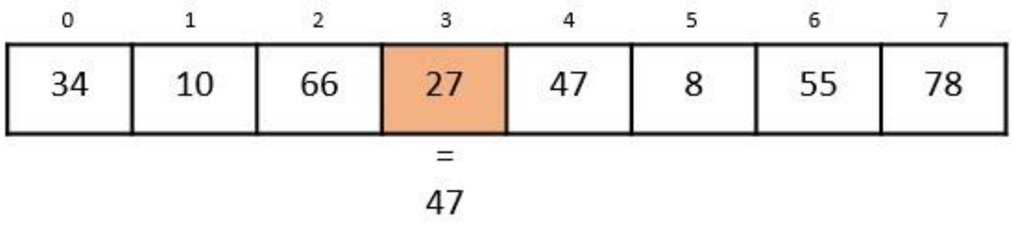
Still, 47 ≠ 10, making the algorithm move for another iteration.

Step 3

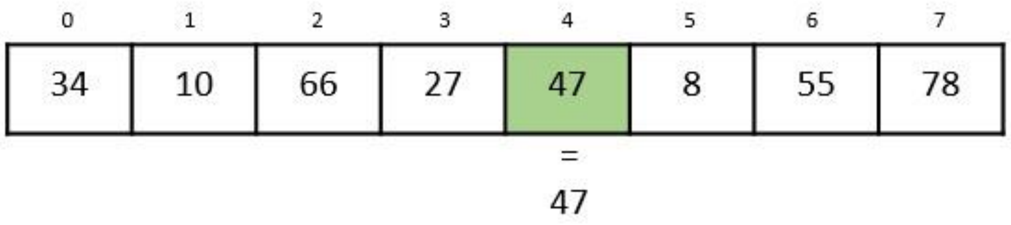
The next element 66 is compared with 47. They are both not a match so the algorithm compares the further elements.



Step 4
Now the element in 3rd index, 27, is compared with the key value, 47. They are not equal so the algorithm is pushed forward to check the next element.



Step 5
Comparing the element in the 4th index of the array, 47, to the key 47. It is figured that both the elements match. Now, the position in which 47 is present, i.e., 4 is returned.



The output achieved is “Element found at 4th index”.

Implementation
In this tutorial, the Linear Search program can be seen implemented in four programming languages. The function compares the elements of input with the key value and returns the position of the key in the array or an unsuccessful search prompt if the key is not present in the array.

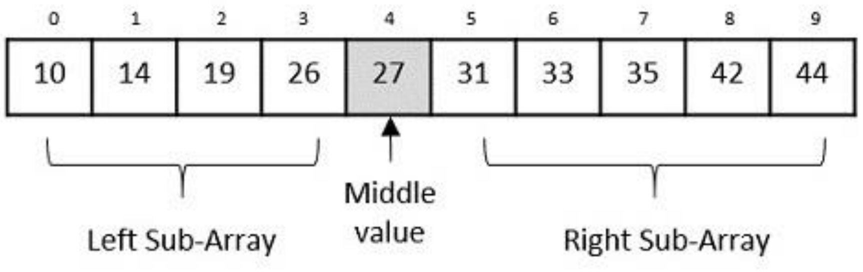
```
#include <iostream>
using namespace std;
void linear_search(int a[], int n, int key){
  int i, count = 0;
  for(i = 0; i < n; i++) {
    if(a[i] == key) { // compares each element of the array
      cout << "The element is found at position " << i+1 <<endl;
      count = count + 1;
    }
  }
  if(count == 0) // for unsuccessful search
    cout << "The element is not present in the array" <<endl;
}
int main(){
  int i, n, key;
  n = 6;
  int a[10] = {12, 44, 32, 18, 4, 10};
  key = 18;
  linear_search(a, n, key);
  key = 23;
  linear_search(a, n, key);
  return 0;

}
```

Output

```
The element is found at position 4
The element is not present in the array
```

Binary Search Algorithm

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer, since it divides the array into half before searching. For this algorithm to work properly, the data collection should be in the sorted form.
Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero.



Binary Search Algorithm

Binary Search algorithm is an interval searching method that performs the searching in intervals only. The input taken by the binary search algorithm must always be in a sorted array since it divides the array into subarrays based on the greater or lower values. The algorithm follows the procedure below −

Step 1 − Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

Step 2 − If it does not match the key value, check if the key value is either greater than or less than the median value.

Step 3 − If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.

Step 4 − Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.

Step 5 − If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while
end procedure
```

Analysis

Since the binary search algorithm performs searching iteratively, calculating the time complexity is not as easy as the linear search algorithm.

The input array is searched iteratively by dividing into multiple sub-arrays after every unsuccessful iteration. Therefore, the recurrence relation formed would be of a dividing function.

To explain it in simpler terms,

- During the first iteration, the element is searched in the entire array. Therefore, length of the array = n.
- In the second iteration, only half of the original array is searched. Hence, length of the array = n/2.
- In the third iteration, half of the previous sub-array is searched. Here, length of the array will be = n/4.
- Similarly, in the i^{th} iteration, the length of the array will become $n/2^i$
- To achieve a successful search, after the last iteration the length of array must be 1. Hence,

  n/2i = 1

  That gives us –

  n = 2i

  Applying log on both sides,

  log n = log 2i

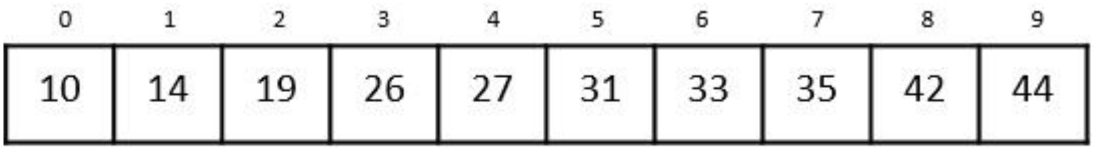
  log n = i. log 2

  i = log n

  The time complexity of the binary search algorithm is **O(log n)**

Example

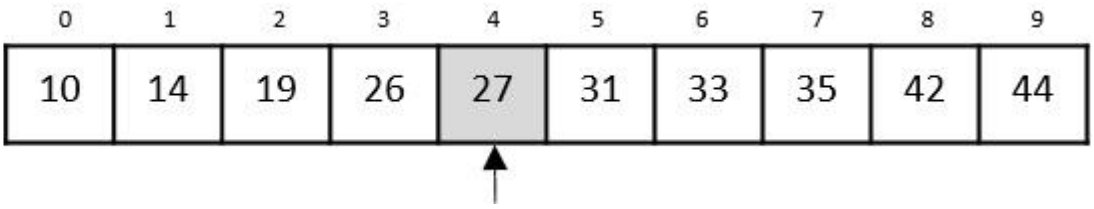
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



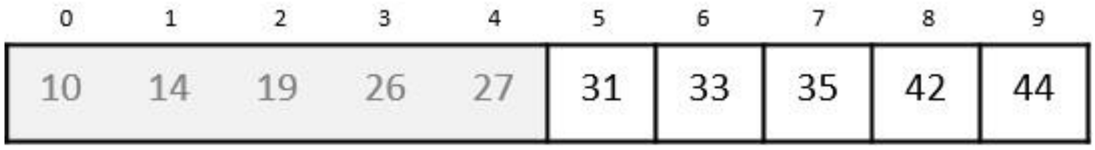
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

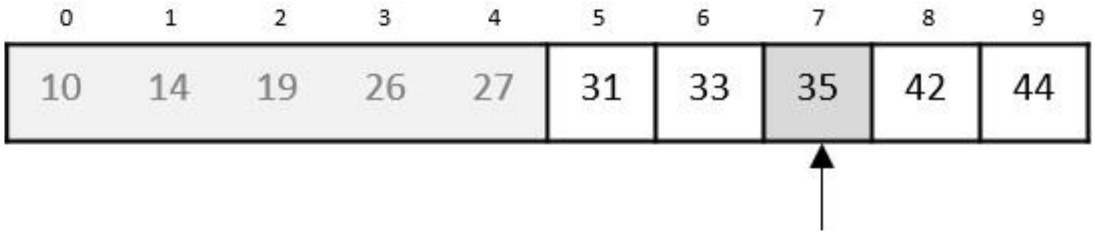


We change our low to mid + 1 and find the new mid value again.

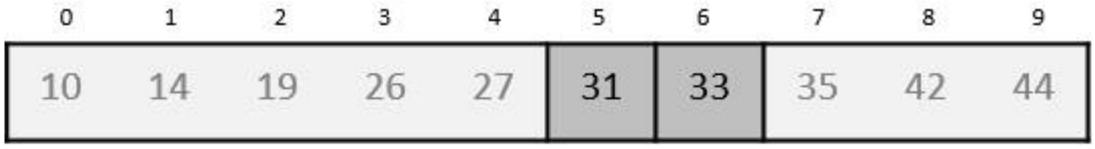
low = mid + 1

mid = low + (high - low) / 2

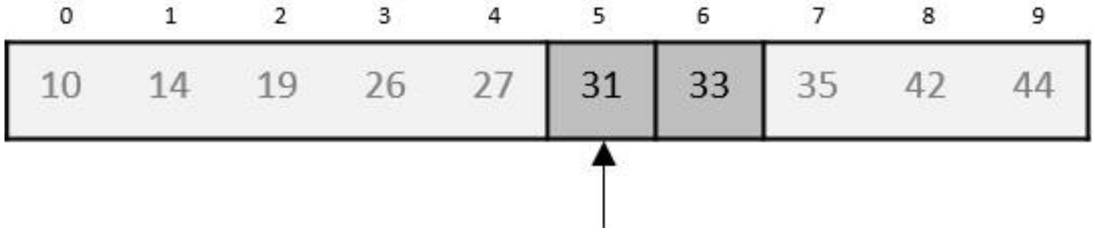
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



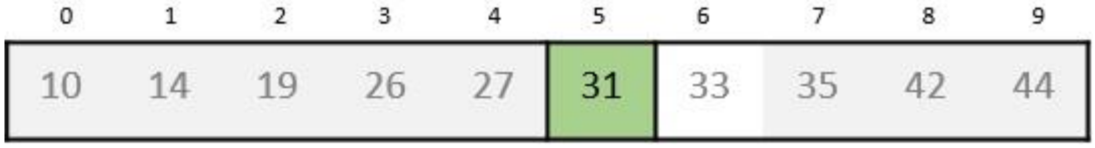
The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

<u>Implementation</u>

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in a sorted form.

```
#include <iostream>
using namespace std;
void binary_search(int a[], int low, int high, int key){
  int mid;
  mid = (low + high) / 2;
  if (low <= high) {
    if (a[mid] == key)
      cout << "Element found at index: " << mid << endl;
    else if(key < a[mid])
      binary_search(a, low, mid-1, key);
    else if (a[mid] < key)
      binary_search(a, mid+1, high, key);
  } else if (low > high)
    cout << "Unsuccessful Search" <<endl;
}
int main(){
  int i, n, low, high, key;
  n = 5;
  low = 0;
  high = n-1;
  int a[10] = {12, 14, 18, 22, 39};
  key = 22;
  binary_search(a, low, high, key);
  key = 23;
  binary_search(a, low, high, key);
  return 0;

}
```

Output

```
Element found at index: 3
Unsuccessful Search
```

Heap: Operations and Applications

A **heap** is a specialized tree-based data structure that satisfies the heap property. Depending on whether it is a max-heap or a min-heap, the heap property ensures that:

- In a **max-heap**, the key of each parent node is greater than or equal to the keys of its children, and the highest key is at the root.
- In a **min-heap**, the key of each parent node is less than or equal to the keys of its children, and the lowest key is at the root.

Heap Operations

1. **Insertion**

To insert a new element into a heap:

- Place the element at the next available position to maintain the complete binary tree property.

- Compare the inserted element with its parent and perform a "heppify-up" (or "bubble-up") operation until the heap property is restored.

Time Complexity: O(log n), where n is the number of elements in the heap.

2. **Deletion (Extract)**

To remove the root element (maximum in max-heap, minimum in min-heap):

- Replace the root with the last element in the heap.
- Remove the last element.
- Perform a "heapify-down" (or "bubble-down") operation by comparing the new root with its children and swapping it with the appropriate child until the heap property is restored.

Time Complexity: O(log n).

3. **Peek (Find Maximum/Minimum)**

In a max-heap, the maximum value is at the root, and in a min-heap, the minimum value is at the root. The operation simply returns the root element.

Time Complexity: O(1).

4. **Heapify**

The heapify operation converts an arbitrary array into a valid heap.

- Start from the last non-leaf node and perform "heapify-down" for each node in reverse level-order.

Time Complexity: O(n).

5. **Increase/Decrease Key**

To change the value of a key in the heap:

- If the key is increased (max-heap) or decreased (min-heap), perform a "heapify-up."
- If the key is decreased (max-heap) or increased (min-heap), perform a "heapify-down."

Time Complexity: O(log n).

6. **Merge (Union)**

Combine two heaps into one:

- Add all elements from one heap to another and heapify the resulting heap.

Time Complexity: O(n + m), where n and m are the sizes of the two heaps.

Heap: Operations and Applications

A **heap** is a specialized tree-based data structure that satisfies the heap property. Depending on whether it is a max-heap or a min-heap, the heap property ensures that:

- In a **max-heap**, the key of each parent node is greater than or equal to the keys of its children, and the highest key is at the root.
- In a **min-heap**, the key of each parent node is less than or equal to the keys of its children, and the lowest key is at the root.

Applications of Heaps

1. **Priority Queues**

Heaps are widely used to implement priority queues, where elements with higher priorities are dequeued before elements with lower priorities.

2. **Heap Sort**

Heap sort is a comparison-based sorting algorithm that uses a binary heap to sort elements. Steps:

- Build a max-heap from the input array.
- Repeatedly swap the root with the last element and reduce the heap size, followed by heapifying the root.

Time Complexity: O(n log n).

3. **Median Maintenance**

Heaps are used to efficiently find the median in a stream of data:

- Maintain two heaps: a max-heap for the lower half and a min-heap for the upper half of the elements.
- Balance the two heaps to ensure the median can be quickly accessed.

4. **Graph Algorithms**

Heaps are used in graph algorithms like:

- **Dijkstra's Algorithm**: To find the shortest path.
- **Prim's Algorithm**: To find the minimum spanning tree.

5. **Kth Largest/Smallest Element**

Heaps efficiently find the Kth largest or smallest element in an array by maintaining a heap of size K.

6. **Heap-Based Data Structures**

Heaps form the basis of several advanced data structures, such as:

- Fibonacci Heaps: Used to improve the time complexity of graph algorithms.
- Binomial Heaps: Useful for union operations.

7. **Task Scheduling**

Heaps are used in task scheduling systems to prioritize tasks based on deadlines or execution times.

8. **Merge K Sorted Lists**

Heaps efficiently merge K sorted lists by maintaining a min-heap of size K to repeatedly extract the smallest element.

Advantages of Heaps

- Efficient insertion and deletion operations.

- Optimal for implementing priority queues.
- Well-suited for sorting and selection problems.

Disadvantages of Heaps

- Searching for arbitrary elements is inefficient (O(n)).
- Not cache-friendly due to scattered memory allocation in binary heap implementations.

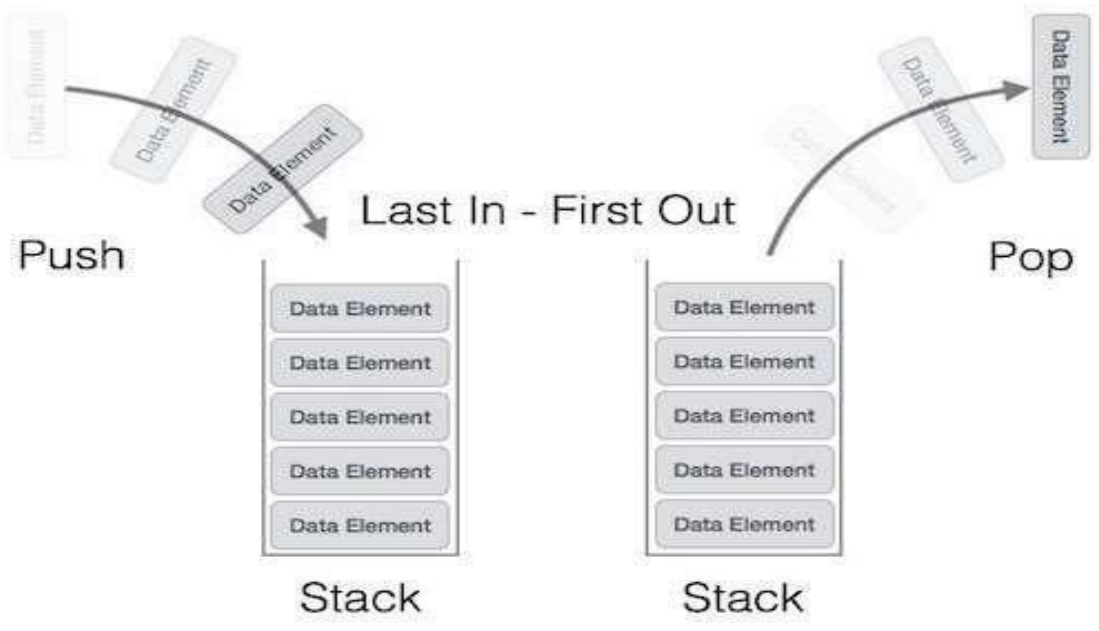
Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Memory Representation

Stacks can be implemented in two main ways:

1. Using an Array

- A stack is represented as a contiguous block of memory with a fixed size.
- A variable top keep tracks of the index of the topmost element in the stack.
- Initially, top = -1 (indicating an empty stack).
- When a new element is added, top is incremented, and the element is stored at stack[top].
- When an element is removed, the value of top is decremented.
- This implementation is simple but has a limitation of fixed size, which may lead to **overflow**.

2. Using a Linked List

A stack is represented as a dynamic linked list, where each node contains two fields:

- **Data**: The value of the node.
- **Next**: A pointer/reference to the next node in the stack.
    - The top pointer refers to the most recently added node.

  - When a new element is pushed, a new node is created, and top is updated to this new node.
  - This approach provides flexibility with dynamic size but involves additional memory for storing pointers.

Key Concepts in Memory Representation:

- **Top**: Pointer or index indicating the current top element of the stack.
- **Stack Overflow**: Occurs when attempting to push an element onto a full stack (array implementation).
- **Stack Underflow**: Occurs when attempting to pop an element from an empty stack.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek

   return stack[top]

end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return

end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
```

isempty()

Algorithm of isempty() function –

```
begin procedure

   if top less than
   1

      return

   true else

end
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

```
bool isempty() {
   if(top == -1)

      return true;
   else

      return false;
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

Step 1 – Checks if the stack is full.

Step 2 – If the stack is full, produces an error and exit.

Step 3 – If the stack is not full, increments **top** to point next empty space.

Step 4 – Adds data element to the stack location, where top is pointing.

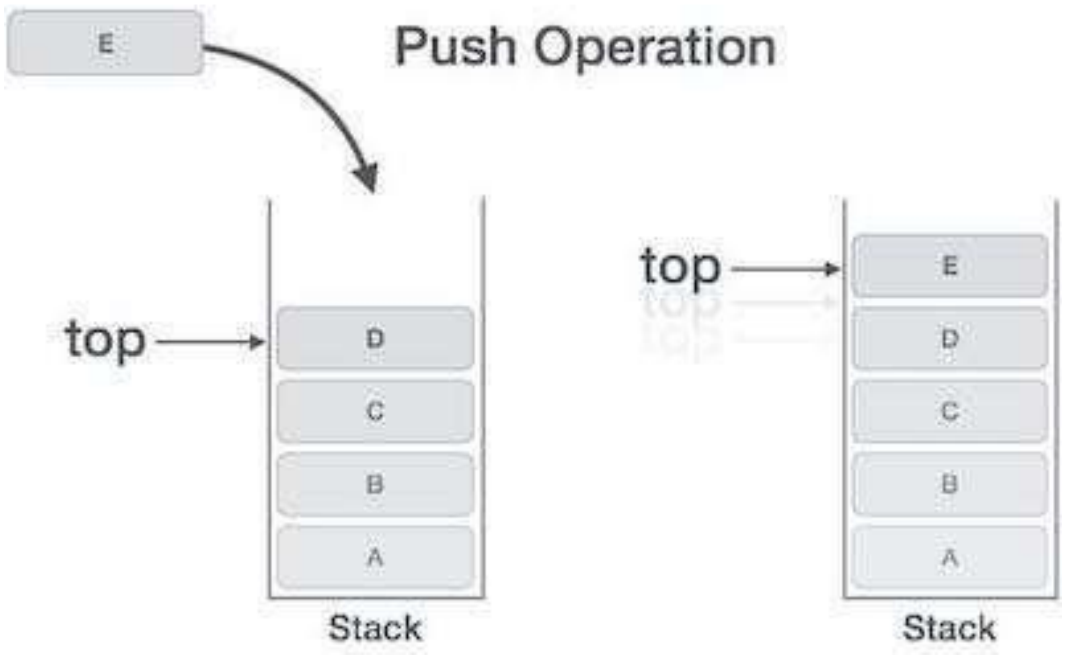
Step 5 – Returns success.

Algorithm (Array Implementation):

1. Check if the stack is full:

- If top == maxSize - 1, display "Stack Overflow" and terminate the operation.

2. Increment top by 1.
3. Add the new element at stack[top].



4. If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Pseudocode:

```
Push(stack, element):
  if top == maxSize - 1:
    print("Stack Overflow")
    return
  top = top + 1
  stack[top] = element
```

<u>**Pop Operation**</u>

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

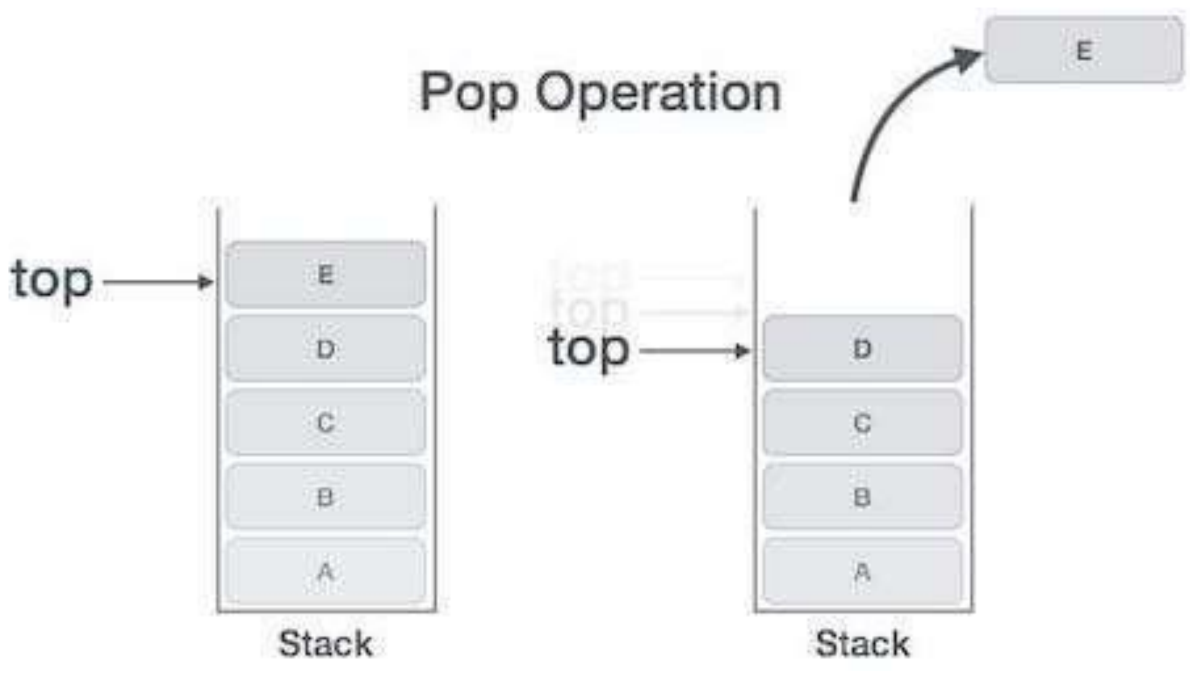
Step 1 – Checks if the stack is empty.

Step 2 – If the stack is empty, produces an error and exit.

Step 3 – If the stack is not empty, accesses the data element at which **top** is pointing.

Step 4 – Decreases the value of top by 1.

Step 5 – Returns success.



Algorithm (Array Implementation):

1. Check if the stack is empty:
   - If top == -1, display "Stack Underflow" and terminate the operation.
2. Retrieve the element at stack[top].
3. Decrement top by 1.

Pseudocode:

```
Pop(stack):
    if top == -1:
        print("Stack Underflow")
        return
    element = stack[top]
    top = top - 1
    return element
```

Peek Operation *(Retrieve the top element without removing it)*

The peek operation allows viewing the topmost element without modifying the stack.

Algorithm:

1. Check if the stack is empty:
   - If top == -1, display "Stack Underflow" and terminate the operation.
2. Return the element at stack[top].

Pseudocode:

```
Peek(stack):
   if top == -1:
      print("Stack Underflow")
      return
   return stack[top]
```

Applications of Stack

Stacks are an essential data structure with numerous real-world applications, including:

1. Expression Evaluation and Conversion

- **Postfix Evaluation**: Stacks are used to evaluate postfix (Reverse Polish Notation) expressions efficiently.
- **Infix to Postfix Conversion**: During the conversion of infix expressions to postfix or prefix, stacks store operators and ensure correct precedence and associativity.

2. Function Call Management

- Stacks manage function calls, especially in recursive programming.
- The **call stack** stores the return address, function parameters, and local variables of each active function.

3. Undo Operations

- Used in text editors, spreadsheets, and other applications to implement undo/redo functionality by storing a history of operations in a stack.

4. Parenthesis Matching

- Stacks are used to check for balanced parentheses in mathematical expressions and code.

5. Backtracking Algorithms

- Algorithms like Depth-First Search (DFS) in graphs, maze-solving, and puzzles (e.g., N-Queens) utilize stacks to explore all possible options and backtrack when necessary.

6. Memory Management

- Stacks are used for runtime memory management of local variables and function calls in programming languages.

7. Browser Navigation

- Used to implement the "Back" and "Forward" buttons in web browsers, where each page visit is pushed onto the stack.

Advantages of Stack

- Simple and easy to implement.
- Supports efficient addition and removal of elements (O(1) time complexity).
- Useful in solving problems requiring temporary data storage or LIFO behavior.

Disadvantages of Stack

- Fixed size in array implementation may lead to overflow.
- Limited access: Only the top element can be accessed directly.
- Overhead in linked list implementation due to extra memory for pointers.

Stack Program in c++

```
#include <iostream>
using namespace std;
class Stack {
private:
  int top;
  int arr[100]; // Array to hold stack elements
  int capacity; // Maximum capacity of the stack
public:
  // Constructor
  Stack(int size = 100) {
    top = -1;
    capacity = size;
}
// Push an element onto the stack
void push(int value) {
    if (top >= capacity - 1) {
        cout << "Stack Overflow! Cannot add more elements." << endl;
        return;
    }
    arr[++top] = value;
    cout << "Pushed " << value << " onto the stack." << endl;
}
// Pop an element from the stack
int pop() {
    if (isEmpty()) {
        cout << "Stack Underflow! Cannot remove elements." << endl;
        return -1;
    }
    return arr[top--];
}
// Peek at the top element without removing it
int peek() {
    if (isEmpty()) {
        cout << "Stack is empty!" << endl;
        return -1;
    }
    return arr[top];
}
// Check if the stack is empty
bool isEmpty() {
    return top == -1;
}
// Display all elements of the stack
```

```
    void display() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return;
        }
        cout << "Stack elements: ";
        for (int i = 0; i <= top; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Stack stack(10); // Create a stack with a capacity of 10
    stack.push(5);
    stack.push(10);
    stack.push(15);
    stack.display();
    cout << "Top element: " << stack.peek() << endl;
    cout << "Popped element: " << stack.pop() << endl;
    stack.display();

    cout << "Popped element: " << stack.pop() << endl;
    cout << "Popped element: " << stack.pop() << endl;
    cout << "Popped element: " << stack.pop() << endl;
    return 0;
}
```

Explanation:

1. **Class Structure**:

- The `Stack` class has an array `arr[]` for storing elements and an integer `top` to track the stack's top position.
- `capacity` sets the maximum number of elements the stack can hold.

2. **Methods**:
   - `push()`: Adds an element to the stack. Checks for overflow.
   - `pop()`: Removes and returns the top element. Checks for underflow.
   - `peek()`: Returns the top element without removing it.
   - `isEmpty()`: Checks if the stack is empty.
   - `display()`: Prints all elements in the stack.
3. **Main Function**:
   - Demonstrates the use of stack methods by performing various operations.

Sample Output:

```
Pushed 5 onto the stack.
Pushed 10 onto the stack.
Pushed 15 onto the stack.
Stack elements: 5 10 15
Top element: 15
Popped element: 15
Stack elements: 5 10
Popped element: 10
Popped element: 5
Stack Underflow! Cannot remove elements.
Popped element: -1
```

1. Program to perform stack operations.

A stack stores multiple elements in a specific order, called **LIFO**. **LIFO** stands for **Last in, First Out**.

A **stack** is an abstract data structure that contains a collection of elements. Stack implements the LIFO mechanism i.e. the element that is pushed at the end is popped out first. Some of the principle operations in the stack are −

- **Push** - This adds a data value to the top of the stack.
- **Pop** - This removes the data value on top of the stack
- **Peek** - This returns the top data value of the stack

```
#include <iostream>
using namespace std;
int stack[100], n = 100, top = -1;
void push(int val) {
   if(top >= n-1)
      cout<<"Stack Overflow"<<endl;
   else {
      top++;
      stack[top] = val;
   }
}
void pop() {
   if(top <= -1)
      cout<<"Stack Underflow"<<endl;
   else {
      cout<<"The popped element is "<< stack[top] <<endl;
      top--;
   }
}
void display() {
   if(top>= 0) {
      cout<<"Stack elements are:";
      for(int i = top; i>= 0; i--)
         cout<<stack[i]<<" ";
      cout<<endl;
   } else
```

```
    cout<<"Stack is empty"<<endl;
}
int main() {
  int ch, val;
  cout<<"1) Push in stack"<<endl;
  cout<<"2) Pop from stack"<<endl;
  cout<<"3) Display stack"<<endl;
  cout<<"4) Exit"<<endl;
  do
   {
    cout<<"Enter choice: "<<endl;
    cin>>ch;
    switch(ch) {
      case 1: {
        cout<<"Enter value to be pushed:"<<endl;
        cin>>val;
        push(val);
        break;
      }
      case 2: {
        pop();
        break;
      }
      case 3: {
        display();
        break;
      }
      case 4: {
        cout<<"Exit"<<endl;
```

```
            break;
          }
          default: {
            cout<<"Invalid Choice"<<endl;
          }
        }
      }
    while(ch != 4);
    return 0;
}
```

Output :

```
Enter value to be pushed:
9
Enter choice:
1
Enter value to be pushed:
8
Enter choice:
1
Enter value to be pushed:
7
Enter choice:
3
Stack elements are:7 8 9
Enter choice:
2
The popped element is 7
Enter choice:
2
```

```
The popped element is 8

Enter choice:

2

The popped element is 9

Enter choice:

2

Stack Underflow

Enter choice:

3

Stack is empty

Enter choice:

4

Exit
```

QUEUE

Queues are a fundamental data structure operating on the First-In-First-Out (FIFO) principle, meaning the first item added is the first to be removed. They are important for organizing and managing data in many applications, including operating systems, network protocols, and data processing pipelines. Queues are essentially used to manage threads in multithreading and implementing priority queuing systems.

What is a Queue?

A queue is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed. A queue is an Abstract Data Type (ADT) similar to stack, the thing that makes queue different from stack is that a queue is open at both its ends. The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages.

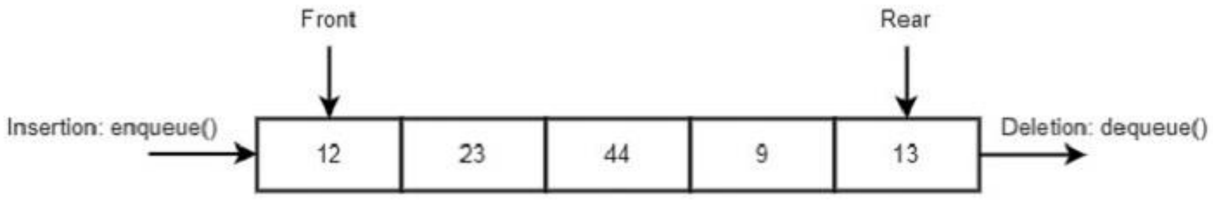
A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Characteristics of a Queue:

1. **Linear order:** Maintains the order of elements.
2. **FIFO Principle:** The first element inserted is the first to be removed.
3. **Operations:**
    - **Enqueue:** Adding an element to the end of the queue.
    - **Dequeue:** Removing an element from the front of the queue.

Representation of Queues

Similar to the stack ADT, a queue ADT can also be implemented using arrays, linked lists, or pointers. As a small example in this tutorial, we implement queues using a one-dimensional array.



Queue: FIFO Operation

Memory Representation

Queues can be represented in memory in two primary ways:

1. Array Representation
    - Array-based implementation (Static Queue)
    - A fixed-size array is used to implement the queue.
    - Two pointers, **front** and **rear**, are used to track the start and end of the queue.
2. Linked List Representation
    - Linked List-based implementation (Dynamic Queue)
    - A dynamic implementation using linked nodes where each node contains the data and a pointer to the next node.
    - Pointers **front** and **rear** keep track of the queue's start and end.

Types of Queues in Data Structure

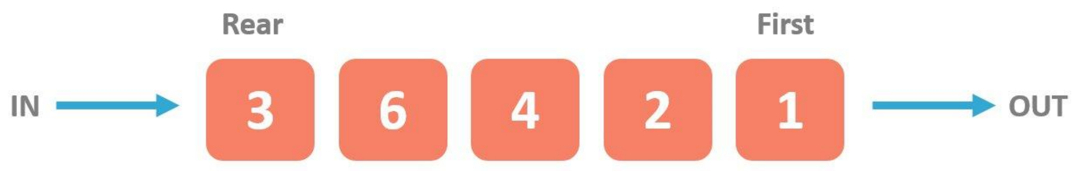
There are four different types of queues in data structures:

- Simple Queue
- Circular Queue
- Priority Queue
- Double-Ended Queue (Deque)

Simple Queue (Linear Queue)

Simple Queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, where elements are added to the rear (back) and removed from the front (head).

- Ordered collection of comparable data kinds.
- Queue structure is FIFO (First in, First Out).
- When a new element is added, all elements added before the new element must be deleted to remove the new element.



.

Applications of Simple Queue

- Resource Allocation: Simple Queues are is useful for resource allocation in operating systems that manage resource requests such as CPU, memory, and I/O devices.
- Batch Processing: Queues accommodate batch jobs, for instance, tasks like data processing or rendering images, which are queued up for sequential execution.
- Message Buffering: It helps to buffer the message in communication systems to ensure a smooth data flow among processes.

Linear Queue Representation Steps

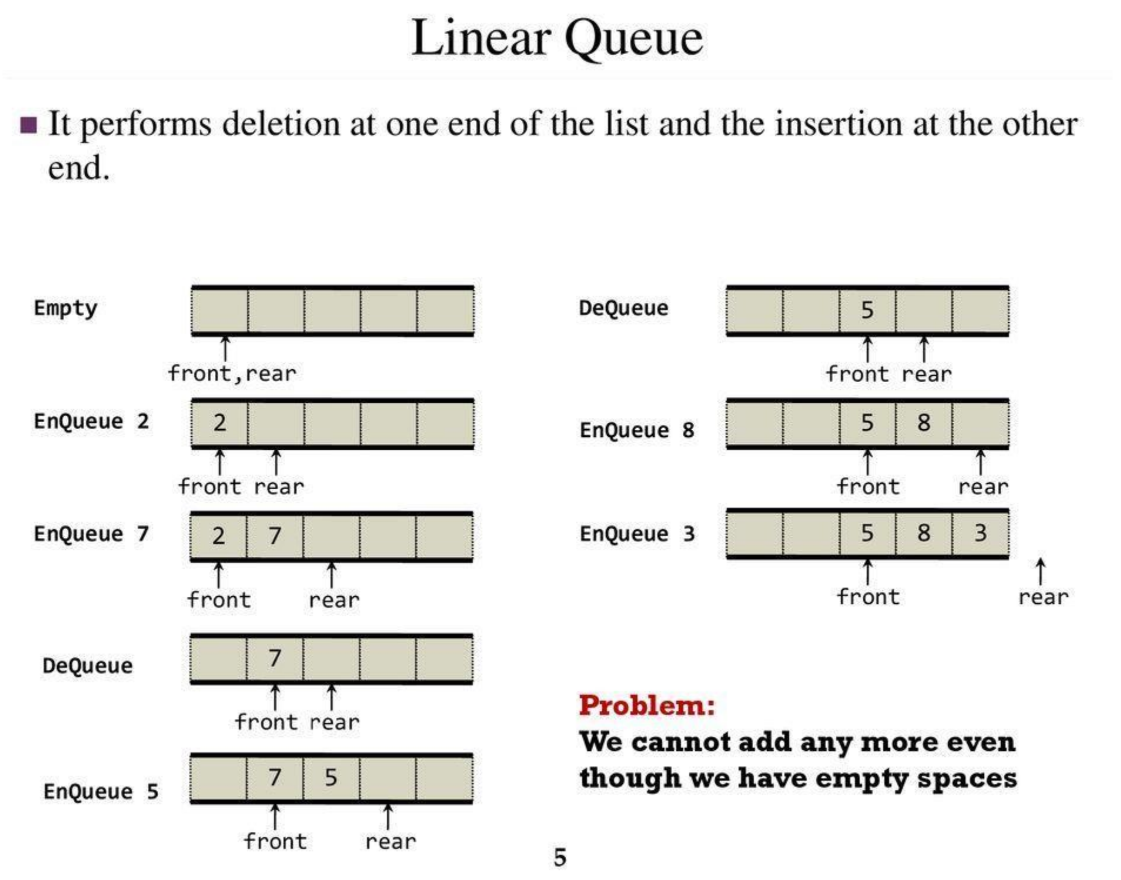
A Linear Queue is a simple queue where elements are inserted at the rear and removed from the front. It follows the FIFO (First In, First Out) principle.

Operations in a Linear Queue

1. **Enqueue (Insertion)**

- Check if the queue is full (rear == SIZE - 1).
- If not full, increment rear and insert the new element.
- If inserting the first element, set front = 0.

2. **Dequeue (Deletion)**
    - Check if the queue is empty (front == -1 or front > rear).
    - If not empty, remove the front element and increment front.
    - If the last element is removed, reset front and rear to -1.



Linear Queue

- It performs deletion at one end of the list and the insertion at the other end.

Problem:
We cannot add any more even though we have empty spaces

Algorithm for Enqueue (Insertion) in Linear Queue

```
void enqueue(int queue[], int Crear, int size, int value) { if (rear
  == size - 1) {
    cout << "Queue is Full (Overflow)\n"; return;
  }
  queue[++rear] = value;
}
```

Algorithm for Dequeue (Deletion) in Linear Queue

```
void dequeue(int queue[], int Cfront, int rear) { if
  (front > rear) {
    cout << "Queue is Empty (Underflow)\n";
    return;
  }
```

```
    cout << "Deleted: " << queue[front++] << endl;
}
```

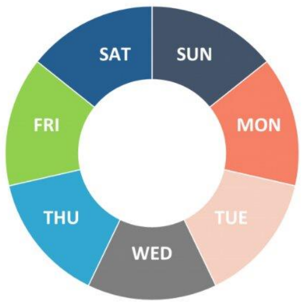
Drawback of Linear Queue

- Even after dequeuing elements, space is not **reused**, leading to inefficiency.
- Requires shifting of elements, which is **costly**.

Circular Queue

A circular queue is a special case of a simple queue in which the last member is linked to the first, forming a circle-like structure.

- The last node is connected to the first node.
- Also known as a **Ring Buffer,** the nodes are connected end to end.
- Insertion takes place at the front of the queue, and deletion at the end of the queue.
- **Example of circular queue application:** Insertion of days in a week.

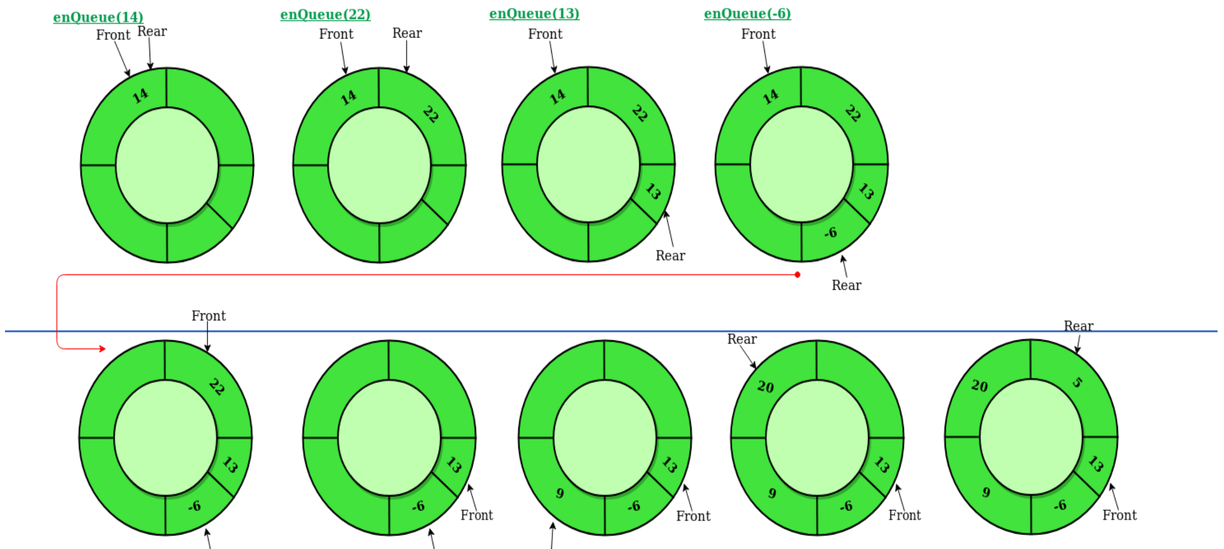


1. Circular Queue Representation Steps

A Circular Queue overcomes the limitation of the linear queue by connecting the last position back to the first position, making use of empty spaces.

Operations in a Circular Queue

1. **Enqueue (Insertion)**
    - Check if the queue is full using the condition: (rear + 1) % SIZE == front
    - If not full, insert the new element at (rear + 1) % SIZE.
    - If inserting the first element, set front = 0.
2. **Dequeue (Deletion)**
    - Check if the queue is empty (front == -1).
    - Remove the front element and move front to (front + 1) % SIZE.
    - If the last element is removed, reset front and rear to -1.



Algorithm for Enqueue in Circular Queue

```
void enqueue(int queue[], int Cfront, int Crear, int size, int value) { if ((rear
   + 1) % size == front) {
      cout << "Queue is Full (Overflow)\n"; return;
   }
   if (front == -1) front = 0;  // First element rear
   = (rear + 1) % size;
   queue[rear] = value;}
```

Algorithm for Dequeue in Circular Queue

```
void dequeue(int queue[], int Cfront, int Crear, int size) { if
   (front == -1) {
      cout << "Queue is Empty (Underflow)\n";
      return;
   }
   cout << "Deleted: " << queue[front] << endl; if
   (front == rear) {
      front = rear = -1; // Queue becomes empty
   } else {
      front = (front + 1) % size;
   }
}
```

Advantages of Circular Queue

- Efficient use of memory as spaces are **reused**.
- No need for shifting elements.

Applications of Circular Queue

- CPU Scheduling: Used in operating systems to manage processes.
- Data Buffering: Frequently used in data streaming and buffering applications.
- Simulation Systems: Helpful in simulating real-world systems and processes, e.g., traffic flow control.

<u>Basic Queue Operations in Queue Data Structure</u>

Below are the basic queue operations in data structure:

Operation	Description
enqueue()	Process of adding or storing an element to the end of the queue

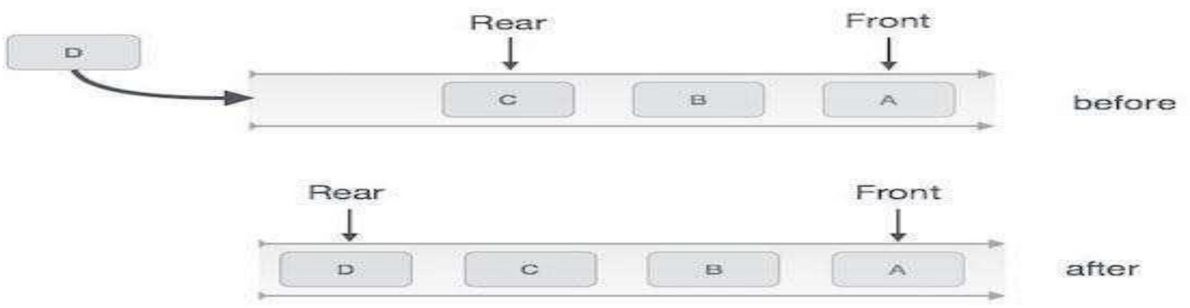
dequeue()	Process of removing or accessing an element from the front of the queue
peek()	Used to get the element at the front of the queue without removing it
initialize()	Creates an empty queue
isfull()	Checks if the queue is full
isempty()	Check if the queue is empty

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

Below are the steps to enqueue (insert) data into a queue

- Check whether the queue is full or not.
- If the queue is full – print the overflow error and exit the program.
- If the queue is not full – increment the rear pointer to point to the next empty space.
- Else add the element in the position pointed by Rear.
- Return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for Enqueue Operation

```
procedure enqueuer (data)
 if queue is full
 return overflow
 endif
  rear ← rear + 1
queue[rear] ← data
 return true
end procedure
```

Implementation of enqueue()

```
int enqueue(int data)
   if(isfull())
      return 0;
   rear = rear + 1;
   queue[rear] = data;
   return 1;
end procedure
```

```
#include <iostream>
#include <string>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
   return itemCount == MAX;
}
bool isEmpty(){
   return itemCount == 0;
}
int removeData(){
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
void insert(int data){
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
int main(){
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);
   insert(15);
   printf("Queue: ");
   while(!isEmpty()) {
      int n = removeData();
      printf("%d ",n);
   }
}
```

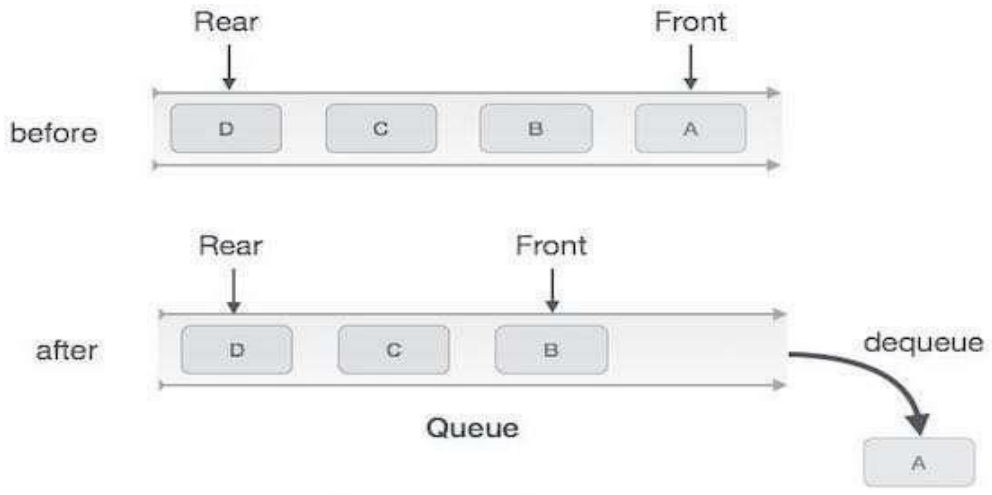
Output

```
Queue: 3 5 9 1 12 15
```

Dequeue Operation

Below are the steps to perform the dequeue operation

- Check whether the queue is full or not.
- If the queue is empty – print the underflow error and exit the program.
- If the queue is not empty – access the data where the front is pointing.
- Else increment the front pointer to point to the next available data element.
- Return success.



Queue Dequeue

Algorithm for Dequeue Operation

```
procedure dequeue
 if queue is empty
    return underflow
 end if  data = queue[front]front ← front + 1return true end procedure
```

Implementation of dequeue()

```
#include <iostream>
```

```
#include <string>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
  return itemCount == MAX;
}
bool isEmpty(){
  return itemCount == 0;
}
void insert(int data){
  if(!isFull()) {
    if(rear == MAX-1) {
      rear = -1;
    }
    intArray[++rear] = data;
    itemCount++;
  }
}
int removeData(){
  int data = intArray[front++];
  if(front == MAX) {
    front = 0;
  }
  itemCount--;
  return data;
}
int main(){
  int i;

  /* insert 5 items */
  insert(3);
  insert(5);
  insert(9);
  insert(1);
  insert(12);
  insert(15);
  printf("Queue: ");
  for(i = 0; i < MAX; i++)
    printf("%d ", intArray[i]);
```

```
// remove one item
int num = removeData();
printf("\nElement removed: %d\n",num);
printf("Updated Queue: ");
while(!isEmpty()) {
  int n = removeData();
  printf("%d ",n); }
```

Output

```
Queue: 3 5 9 1 12 15
Element removed: 3
Updated Queue: 5 9 1 12 15
```

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];


   return data;
}
}
```

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

```
begin procedure peek

   return queue[front]

end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {
   return queue[front];}
```

```
#include <iostream>
#include <string>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
int peek(){
  return intArray[front];
}
bool isFull(){
  return itemCount == MAX;
}
void insert(int data){
  if(!isFull()) {
    if(rear == MAX-1) {
      rear = -1;
    }
    intArray[++rear] = data;
    itemCount++;
  }
}
int main(){
  int i;
  /* insert 5 items */
  insert(3);
  insert(5);
  insert(9);
  insert(1);
  insert(12);
  insert(15);
  printf("Queue: ");
  for(i = 0; i < MAX; i++)
    printf("%d ", intArray[i]);
  printf("\nElement at front: %d\n",peek());
}
```

Output

```
Queue: 3 5 9 1 12 15
Element at front: 3
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ.

```
begin procedure isfull
   if rear equals to MAXSIZE
      return true
   else

      return false
          endif
          end procedure
```

Implementation of isfull()

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;

   else
      return false;
```

```
#include <iostream>
#include <string>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
  return itemCount == MAX;
}
void insert(int data){
  if(!isFull()) {
    if(rear == MAX-1) {
      rear = -1;
    }
    intArray[++rear] = data;
    itemCount++;
  }
}
```

```
int main(){
  int i;

  /* insert 5 items */
  insert(3);
  insert(5);
  insert(9);
  insert(1);
  insert(12);
  insert(15);
  printf("Queue: ");
  for(i = 0; i < MAX; i++)
    printf("%d ", intArray[i]);
  printf("\n");
  if(isFull()) {
    printf("Queue is full!\n");
  }
}
```

Output

```
Queue: 3 5 9 1 12 15
Queue is full!
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

   if front is less than MIN OR front is greater than rear
      return true
   else
      return

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the programming code –

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
```

```
#include <iostream>
#include <string>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isEmpty(){
   return itemCount == 0;
}
int main(){
   int i;
   printf("Queue: ");
   for(i = 0; i < MAX; i++)
      printf("%d ", intArray[i]);
   printf("\n");
   if(isEmpty()) {
      printf("Queue is Empty!\n");
   }
}
```

Output

```
Queue: 0 0 0 0 0 0
Queue is Empty!
```

Implementation of Queue

A queue can be implemented in two ways:

- **Sequential allocation**: It can be implemented using an array. A queue implemented using an array can organize only a limited number of elements.
- **Linked list allocation**: It can be implemented using a linked list. A queue implemented using a linked list can organize unlimited elements.

Queue applications in Data Structure

A queue data structure is generally used in scenarios where the FIFO approach (First In First Out) has to be implemented. The following are some of the most common queue applications in data structure:

- Managing requests on a single shared resource, such as CPU scheduling and disk scheduling
- Handling hardware or real-time systems interrupts
- Handling website traffic
- Routers and switches in networking
- Maintaining the playlist in media players
- **CPU Scheduling:** Used in Round-Robin and other scheduling algorithms.
- **Data Transmission:** Ensures proper order in packets.
- **Printers:** Jobs are managed in a queue.
- **Call Center Systems:** Handles customer requests in a queue.
- **Breadth-First Search (BFS):** In graph traversal algorithms.

Queue Complete Implementation

```
#include <iostream>
using namespace std;
#include <string>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
int peek(){
  return intArray[front];
}
bool isEmpty(){
  return itemCount == 0;
}
bool isFull(){
  return itemCount == MAX;
}
int size(){
  return itemCount;
}
void insert(int data){
  if(!isFull()) {
    if(rear == MAX-1) {
      rear = -1;
    }
    intArray[++rear] = data;
    itemCount++;
  }
}
int removeData(){
  int data = intArray[front++];
  if(front == MAX) {
    front = 0;
  }
  itemCount--;
  return data;
}
int main(){

  /* insert 5 items */
  insert(3);
  insert(5);
  insert(9);
  insert(1);
  insert(12);
  insert(15);
  cout<<"Queue size: "<<size();
  cout<<"\nQueue: ";
  for(int i = 0; i < MAX; i++){
     cout<<intArray[i]<<" ";
  }
  if(isFull()) {
    cout<<"\nQueue is full!";
  }

  // remove one item
  int num = removeData();
  cout<<"\nElement removed: "<<num;
  cout<<"\nQueue size after deletion: "<<size();
  cout<<"\nElement at front: " <<peek();
}
```

Output

```
Queue size: 6
Queue: 3 5 9 1 12 15
Queue is full!
Element removed: 3
Queue size after deletion: 5
Element at front: 5
```

Applications of Queue

A **queue** is a data structure that follows the **First-In-First-Out (FIFO)** principle. It has numerous applications in computing, networking, and real-world scenarios. Here are some key applications:

1. Operating System & Process Scheduling

- **CPU Scheduling:** Queues manage processes in operating systems using scheduling algorithms like **First-Come-First-Serve (FCFS)** and **Round Robin Scheduling**.
- **Disk Scheduling:** Disk I/O requests are handled using queues to optimize data retrieval.

2. Data Structure and Algorithm Applications

- **Breadth-First Search (BFS):** A queue is used to explore nodes level by level in graph and tree traversal.
- **Tree and Graph Traversals:** Used to explore nodes in hierarchical structures.
- **Priority Queues in Dijkstra's Algorithm:** Helps in finding the shortest path in graphs.

3. Network Applications

- **Packet Scheduling:** Routers and switches use queues to manage data packet transfers.
- **Call Handling in Call Centres:** Incoming customer calls are queued and answered in order.

4. Job Scheduling in Printers

- When multiple print jobs are submitted, they are stored in a queue and executed one by one.

5. Real-time Systems & Task Scheduling

- In real-time applications, queues prioritize urgent tasks (e.g., medical monitoring systems).

6. Multi-threading and Inter-Process Communication

- **Producer-Consumer Problem:** One thread produces data, and another consumes it using a queue for synchronization.

7. Banking & Ticket Counters

- Customers are served in a **FIFO** manner at banks, ticket counters, and service centers.

8. Traffic Management Systems

- Traffic signals use queues to manage vehicles at intersections efficiently.

9. Message Queues in Distributed Systems

- Messaging services like **Kafka, RabbitMQ** use queues for asynchronous communication between services.

10. Call Center and Customer Support

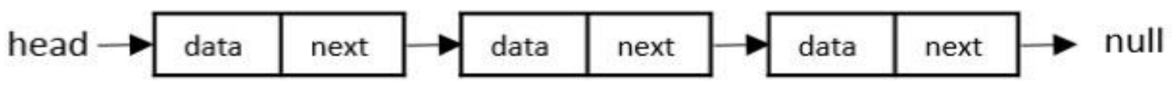
- Calls are queued and assigned to available agents in **First-Come-First-Serve** order.

Linked Lists:

What is Linked List?

A linked list is a linear data structure which can store a collection of "nodes" connected together via links i.e. pointers. Linked lists nodes are not stored at a contiguous location, rather they are linked using pointers to the different memory locations. A node consists of the data value and a pointer to the address of the next node within the linked list.

A linked list is a dynamic linear data structure whose memory size can be allocated or de-allocated at run time based on the operation insertion or deletion, this helps in using system memory efficiently. Linked lists can be used to implement various data structures like a stack, queue, graph, hash maps, etc.



A linked list starts with a **head** node which points to the first node. Every node consists of data which holds the actual data (value) associated with the node and a next pointer which holds the memory address of the next node in the linked list. The last node is called the tail node in the list which points to **null** indicating the end of the list.

LINKED LIST

A linked list is a linear data structure where each node contains:

1. Data
2. Pointer to the next node (Singly Linked List) or both next and previous nodes (Doubly Linked List).

Types of Linked Lists

1. **Singly Linked List:** Each node has one pointer to the next node.
2. **Doubly Linked List:** Each node has two pointers, one to the next and one to the previous node.
3. **Circular Linked List:** The last node connects to the first node.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

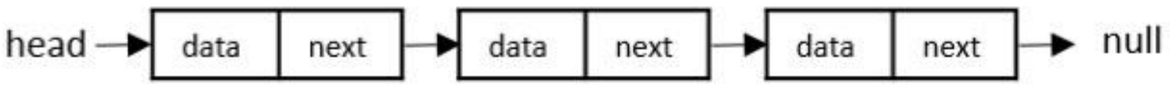
- Singly Linked List
- Doubly Linked List
- Circular Linked List

Singly Linked Lists

Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.

Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are insertion, deletion and traversal.

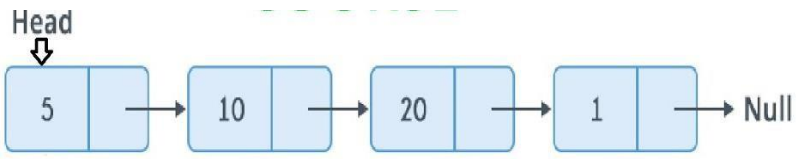


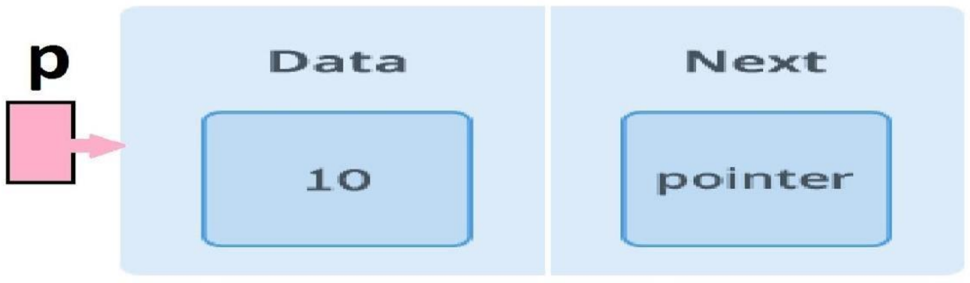
A **Singly Linked List (SLL)** is a linear data structure where each element, known as a node, contains data and a reference (or link) to the next node in the sequence. This structure allows for efficient insertion and deletion operations without the need for contiguous memory allocation.

Graphical Representation of a Singly Linked List:

Each node in a singly linked list comprises two components:

1. **Data Field:** Stores the actual data.
2. **Next Pointer:** Holds the reference to the next node in the list.





Algorithm for Insertion at Beginning (SLL)

```
struct Node { int
   data;
   Node* next;
};
void insertAtBeginning(Node* Chead, int value) {
   Node* newNode = new Node();
   newNode->data  =  value;
   newNode->next  =  head;
   head = newNode;
}
```

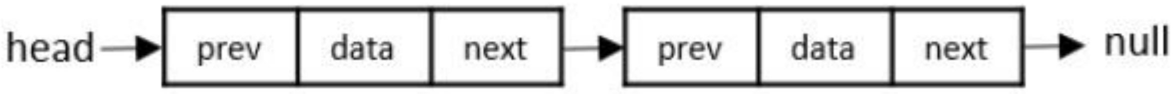
Algorithm for Deletion at End (SLL)

```
void deleteAtEnd(Node* Chead) { if
   (!head) return;
   if (!head->next) {
      delete head; head
      = NULL; return;
   }
   Node* temp = head;
   while (temp->next->next) temp = temp->next; delete
   temp->next;
   temp->next = NULL;
}
```

Doubly Linked Lists

Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

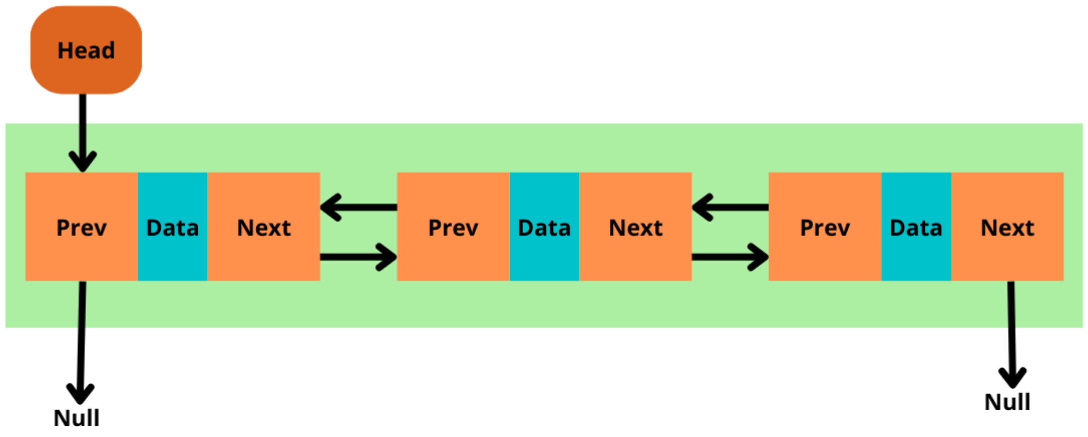
In a doubly linked list, each node contains a data part and two addresses, one for the previous node and one for the next node.



A **Doubly Linked List (DLL)** is a type of linked list in which each node contains three components:

1. **Data Field:** Stores the actual data.
2. **Next Pointer:** References the next node in the sequence.
3. **Previous Pointer:** References the previous node in the sequence.

This bidirectional linkage allows traversal in both forward and backward directions, enhancing flexibility over singly linked lists.



Doubly Linked List

Algorithm for Insertion at Beginning (DLL)

```
struct DNode {
   int data;
   DNode* prev;
   DNode*  next;
};
```

```
void insertAtBeginning(DNode* Chead, int value) { DNode*
  newNode = new DNode();
  newNode->data = value;
  newNode->prev = NULL;
  newNode->next = head;
  if (head) head->prev = newNode;
  head = newNode;
}
```

Algorithm for Deletion at End (DLL)

```
void deleteAtEnd(DNode* Chead) { if
  (!head) return;
  if (!head->next) {
    delete head; head
    = NULL; return;
  }
  DNode* temp = head;
  while (temp->next) temp = temp->next;
  temp->prev->next = NULL;
  delete temp;
}
```

Advantages of Linked Lists

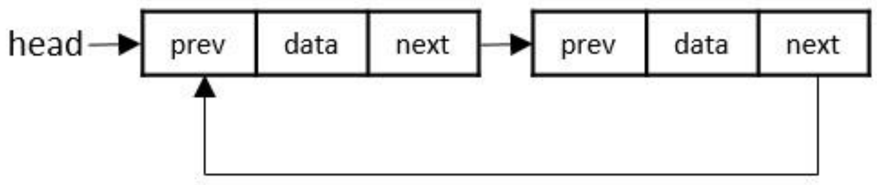
Feature	**Singly Linked List**	**Doubly Linked List**
Insertion/Deletion	Faster at head	Faster at head C tail
Memory Usage	Less (one pointer)	More (two pointers)
Traversal	One direction	Both directions

Circular Linked Lists

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.

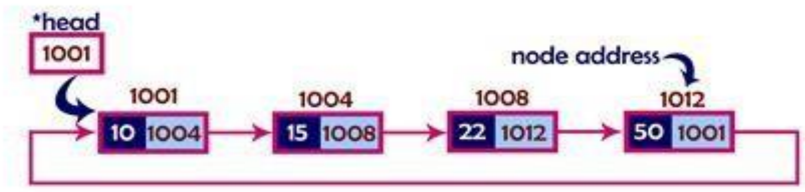


A **Circular Linked List** is a variation of the linked list where the last node points back to the first node, forming a circle. This structure allows for continuous traversal without a defined beginning or end.

Graphical Representation:

1. **Circular Singly Linked List:** In this type, each node contains data and a reference to the next node. The last node's reference points back to the first node.
2. **Circular Doubly Linked List:** Here, each node has three components:

- **Data:** The value stored in the node.
- **Next Pointer:** Reference to the next node.



Difference between Arrays and Linked List

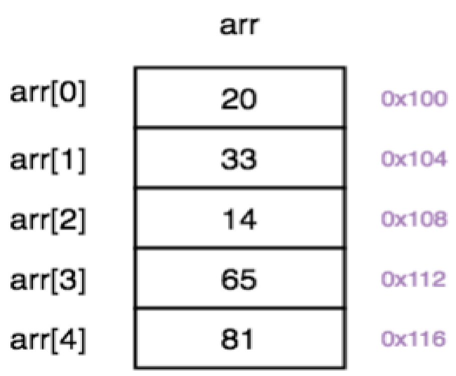
What is Array?

An array is a grouping of data elements or data items stored in contiguous memory. An array is one of the most simple data structures where we can easily access the data element by only using its index number.

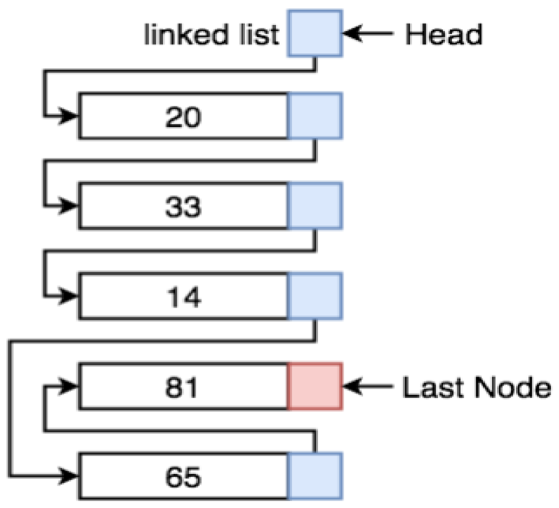
What is a Linked List?

A linked list is a linear and a non-primitive data structure in which each element is allocated dynamically, and each element points to the next element. In other words, we can say that it is a data structure consisting of a group of nodes that concurrently represent a sequence.

S.No.	**ARRAY**	**LINKED LIST**
1.	An array is a grouping of data elements of equivalent data type.	A linked list is a group of entities called a node. The node includes two segments: data and address.
2.	It stores the data elements in a contiguous memory zone.	It stores elements randomly, or we can say anywhere in the memory zone.
3.	In the case of an array, memory size is fixed, and it is not possible to change it during the run time.	In the linked list, the placement of elements is allocated during the run time.
4.	The elements are not dependent on each other.	The data elements are dependent on each other.
5.	The memory is assigned at compile time.	The memory is assigned at run time.
6.	It is easier and faster to access the element in an array.	In a linked list, the process of accessing elements takes more time.
7.	In the case of an array, memory utilization is ineffective.	In the case of the linked list, memory utilization is effective.
8	When it comes to executing any operation like insertion, deletion, array takes more time.	When it comes to executing any operation like insertion, deletion, the linked list takes less time.



Array representation



Basic Operations in Linked List

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below −

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

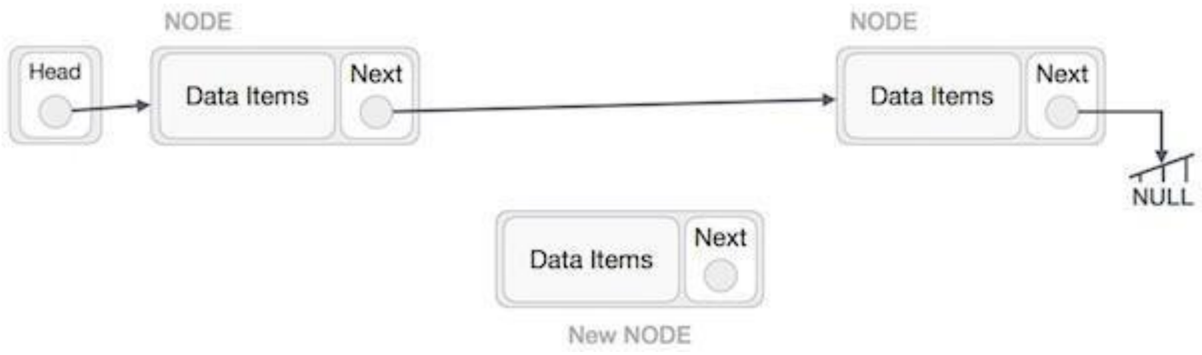
Singly Linked List
What is a Node?

• A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.

Node Implementation

```
// A linked list node struct Node
 {
 int data;
 struct Node *next;
};
 typedef struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
struct Books *add;
} Book;
```

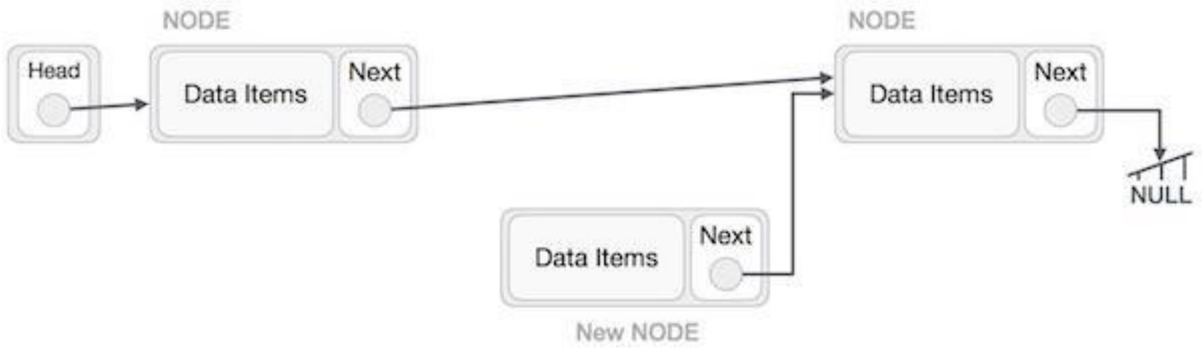
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C –

```
NewNode.next -> RightNode;
```

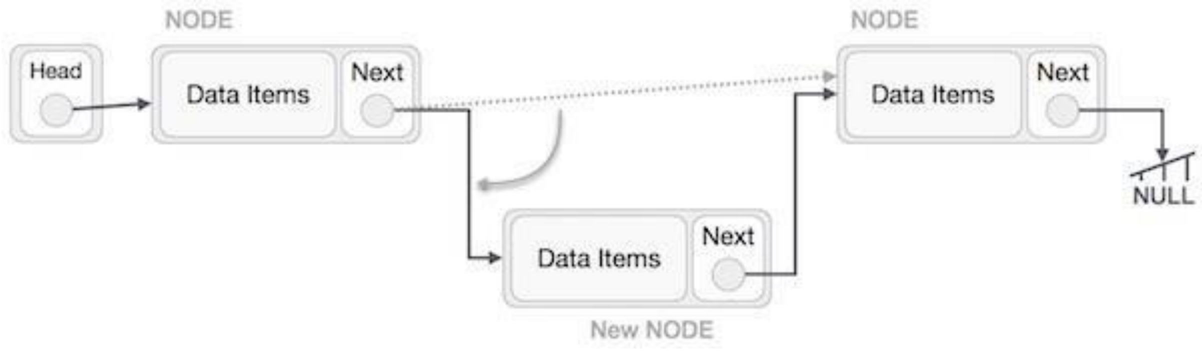
It should look like this –



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```

This will put the new node in the middle of the two. The new list should look like this –



Insertion in linked list can be done in three different ways.

Inserting a node

• A node can be added in three ways
1) At the front of the linked list
2) After a given node.
3) At the end of the linked list.

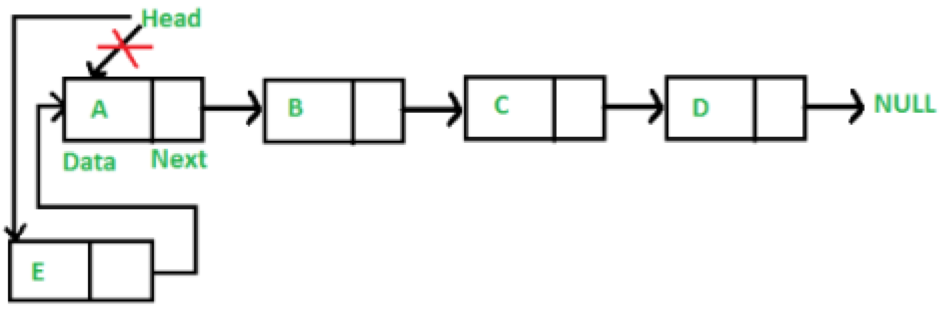
Insertion at Beginning

In this operation, we are adding an element at the beginning of the list.

Algorithm

```
1. START
2. Create a node to store the data
3. Check if the list is empty
4. If the list is empty, add the data to the node and
   assign the head pointer to it.
5. If the list is not empty, add the data to a node and link to the
   current head. Assign the head to the newly added node.
6. END
```

```
void push(struct Node** head_ref, int new_data)
 {
/* 1. allocate node */
struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
 /* 2. put in the data */
 new_node->data = new_data;
 /* 3. Make next of new node as head */
new_node->next = (*head_ref);
/* 4. move the head to point to the new node */
(*head_ref) = new_node;
 }
```



```
#include <bits/stdc++.h>
#include <string>
using namespace std;
struct node {
   int data;
```

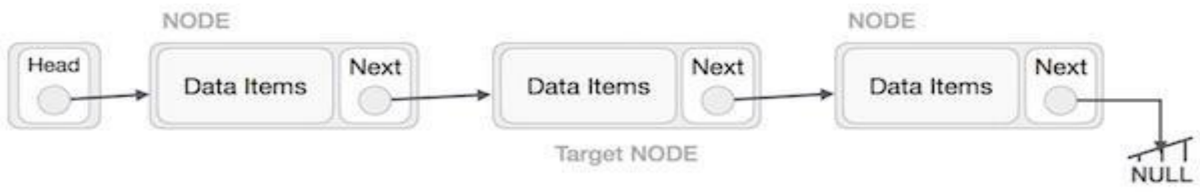
```
  struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
  struct node *p = head;
  cout << "\n[";

  //start from the beginning
  while(p != NULL) {
    cout << " " << p->data << " ";
    p = p->next;
  }
  cout << "]";
}

//insertion at the beginning
void insertatbegin(int data){

  //create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;

  // point it to old first node
  lk->next = head;

  //point first to new first node
  head = lk;
}
int main(){
  insertatbegin(12);
  insertatbegin(22);
  insertatbegin(30);
  insertatbegin(44);
  insertatbegin(50);
  cout << "Linked List: ";

  // print list
  printList();
}
```

Output

```
Linked List:
[ 50  44  30  22  12 ]
```

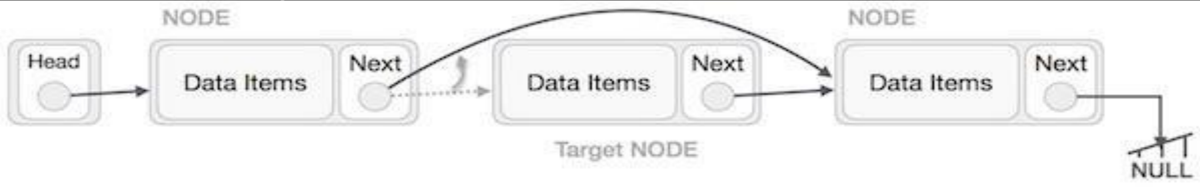
Linked List - Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



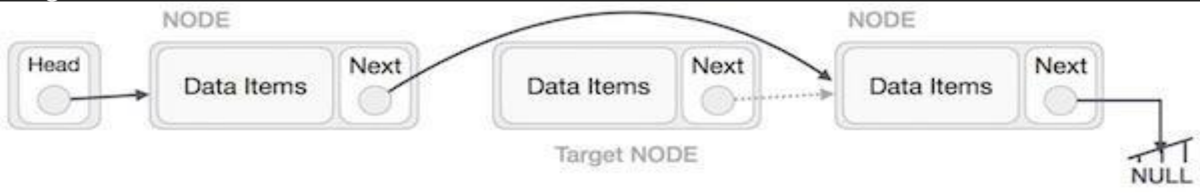
The left (previous) node of the target node now should point to the next node of the target node −

```
LeftNode.next -> TargetNode.next;
```

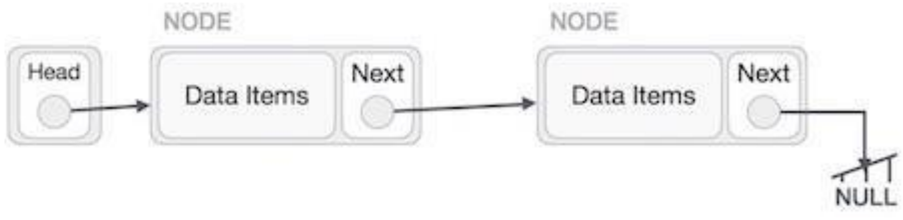


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```



We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node



e completely.

NODE
NODE
Head
Data Items
Next
Data Items
Next
Data Items
Next
New NODE
NULL

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion in linked lists is also performed in three different ways. They are as follows −

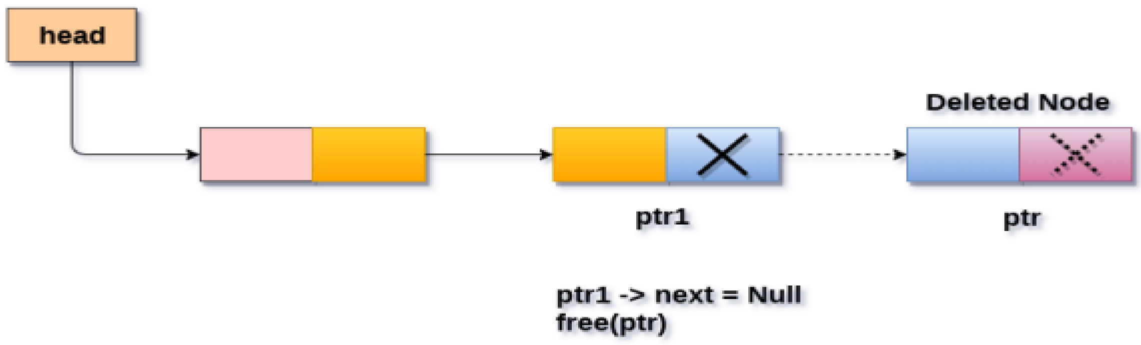
1) At the front of the linked list
2) After a given node/specified position
3) At the end of the linked list.

Deletion at Ending

In this deletion operation of the linked, we are deleting an element from the ending of the list.

```
1. START
2. Iterate until you find the second last element in the list.
3. Assign NULL to the second last element in the list.
4. END
```

```
void end_delete()
 { struct node *ptr,*ptr1;
if(head == NULL)
 {
 Cout<<"\nlist is empty";
 }
 else if(head -> next == NULL)
  {
 free(head);
  head = NULL;
 cout<<"\nOnly node of the list deleted ...";
  }
 else
 {
 ptr = head;
 while(ptr->next != NULL)
  {
  ptr1 = ptr;
 ptr = ptr ->next;
 }
 ptr1->next = NULL;
 free(ptr);
 cout<<"\n Deleted Node from the last ...";
 }
  }
```



```
#include <bits/stdc++.h>
#include <string>
using namespace std;
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// Displaying the list
void printList(){
   struct node *p = head;
   while(p != NULL) {
      cout << " " << p->data << " ";
      p = p->next;
   }
}

// Insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void deleteatend(){
   struct node *linkedlist = head;
   while (linkedlist->next->next != NULL)
      linkedlist = linkedlist->next;
   linkedlist->next = NULL;
}
int main(){
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(44);
   insertatbegin(50);
   cout << "Linked List: ";

   // print list
   printList();
   deleteatend();
   cout << "\nLinked List after deletion: ";
   printList();
}
```

Output

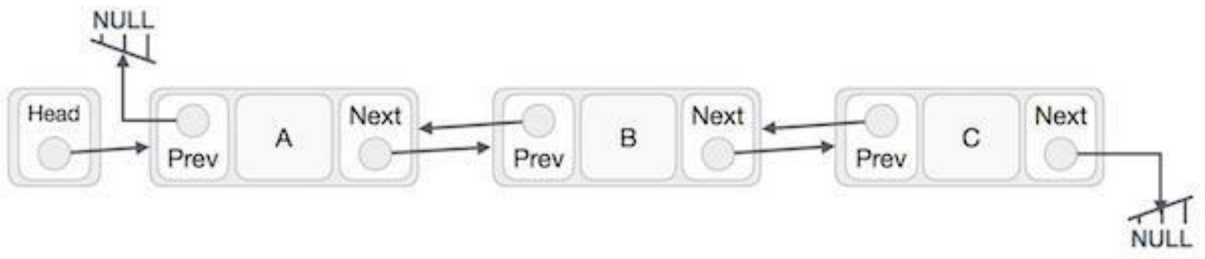
```
Linked List:  50  44  30  22  12
Linked List after deletion:  50  44  30  22
```

What is Doubly Linked List?

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, forward as well as backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **Linked List** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations in Doubly Linked List

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner.

Applications of Linked Lists

1. ***Singly Linked List Applications***

- **Dynamic Memory Allocation:** Used to manage dynamic memory in operating systems.
- **Implementation of Stacks and Queues:** Basis for implementing stack (LIFO) and queue (FIFO) data structures.
- **Polynomial Representation:** Efficiently represent sparse polynomials where each term is stored in a node.
- **Hashing:** Used in chaining methods to resolve hash collisions.
- **Symbol Tables in Compilers:** Efficiently manage dynamic symbol tables.

2. ***Doubly Linked List Applications***

- **Navigation Systems:** Enable efficient forward and backward traversal, such as browser history or playlist navigation.
- **Undo/Redo Functionality:** Used in applications like text editors for undo and redo operations.
- **Dynamic Memory Management:** Helps implement techniques like garbage collection.
- **Complex Data Structures:** Used in implementing advanced structures like binary trees, Fibonacci heaps, or LRU cache.
- **Music/Video Playlists:** Enable easy movement between previous and next items.

Both types of linked lists offer flexibility in memory allocation, making them suitable for applications requiring dynamic memory management and non-contiguous memory use.

Applications of Linked Lists

Linked lists are widely used in various domains due to their **dynamic memory allocation**, **efficient insertion/deletion**, and **non-contiguous storage**. Below are some key applications:

1. Implementation of Data Structures

Stacks and Queues: Implemented using linked lists for efficient dynamic storage.

Graphs: Adjacency lists use linked lists to represent graph edges.

Hash Tables: Collision resolution using chaining (linked list implementation).

2. Dynamic Memory Allocation

Operating Systems: Memory management uses linked lists for **heap allocation** (free and allocated memory blocks).

3. Undo/Redo Functionality

Text Editors & Software Applications: Maintain a history of actions using a **doubly linked list** for undo and redo operations.

4. Web Browsers – Forward & Backward Navigation

Doubly Linked List stores browsing history, allowing users to move **back and forward** between pages.

5. Music & Video Playlists

Circular Linked Lists are used in media players to **loop through songs or videos continuously**.

6. CPU Scheduling (Operating Systems)

Circular Linked Lists implement **Round Robin Scheduling**, where processes are executed in a cyclic order.

7. File Systems

Linked allocation in file systems stores file data across non-contiguous blocks to minimize fragmentation.

8. Polynomial Arithmetic

Linked Lists store and process polynomials efficiently for mathematical computations.ww

9. Social Media & Recommendation Systems

Graph-based models using linked lists connect users, friends, or recommended products dynamically.

10. Network Packet Transmission

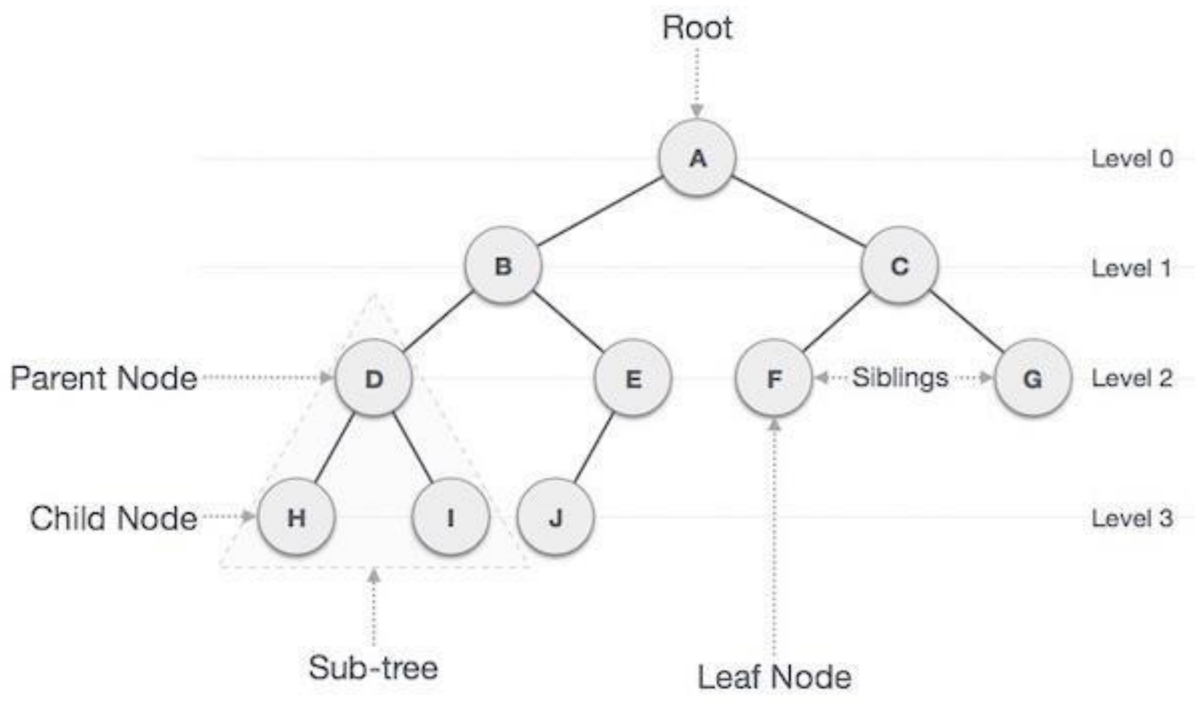
Linked Lists store incoming and outgoing packets efficiently in **network routers**.

Tree Data Structrue

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root.

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

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



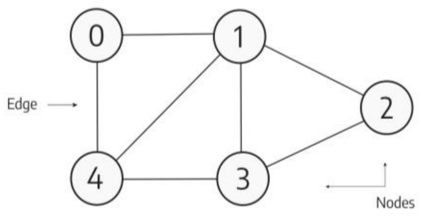
Properties of Trees

- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has n-1 edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and n-1 edges is a tree.

Nodes and Edges in a Tree:

In a Tree Data Structure, the basic components are:

1. **Nodes:**
   - Each element in the tree is called a node.
   - The root node is the topmost node (e.g., A).
   - Each node can have child nodes.



2. **Edges:**
   - The connection between two nodes is called an edge.
   - It represents the relationship (parent-child) between nodes.

Graphical Representation of Nodes and Edges

```
        (A)   <-- Root Node
       /   \
     (B)    (C)
     / \       \
   (D) (E)    (F)
```

Nodes: {A, B, C, D, E, F}

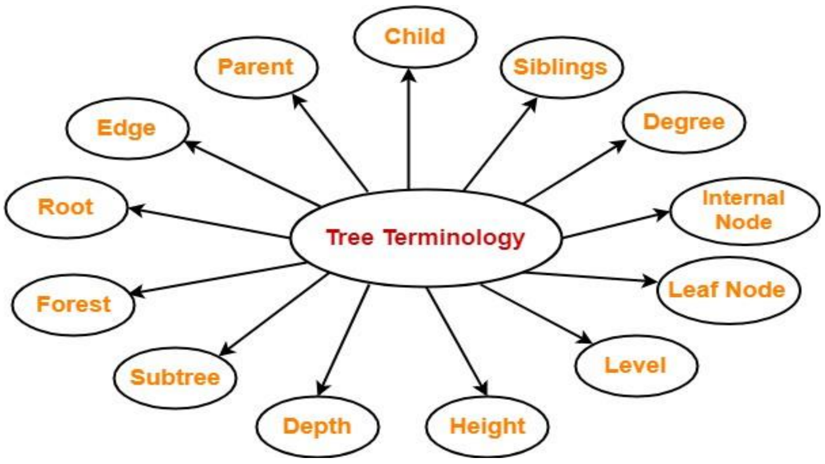
Edges:

- A → B
- A → C
- B → D
- B → E
- C → F

Tree Applications

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

Tree Terminologies



<u>Important Terms</u>

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.

- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

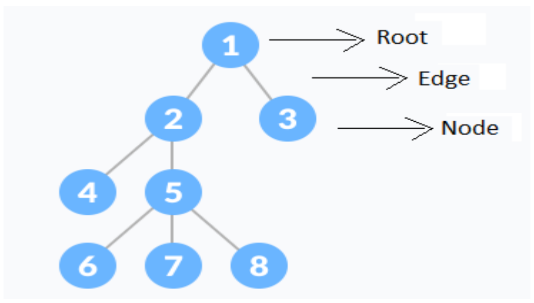
Explanation

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

- The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.
- The node having at least a child node is called an internal node.

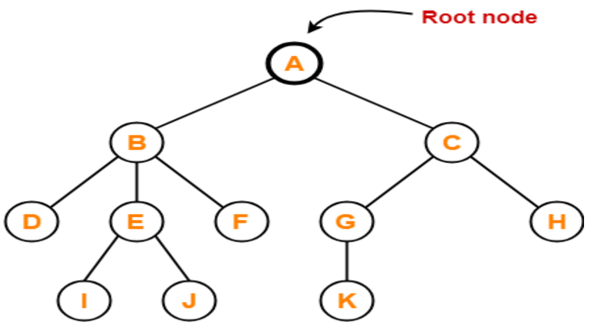
Edge -It is the link between any two nodes.

Root - It is the topmost node of a tree.



Root

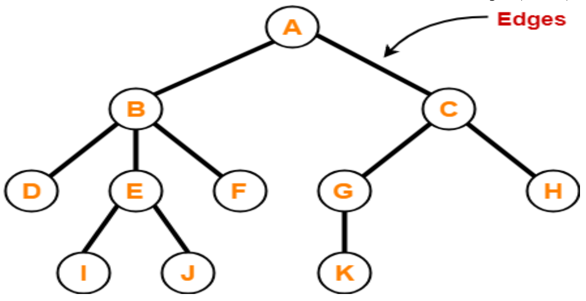
- The first node from where the tree originates is called as a root node.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



Edge

- The connecting link between any two nodes is called as an edge.

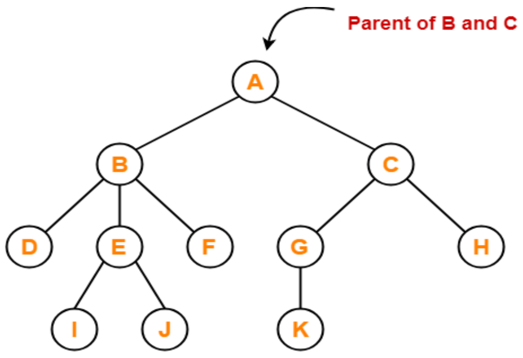
- In a tree with n number of nodes, there are exactly (n-1) number of



edges.

Parent

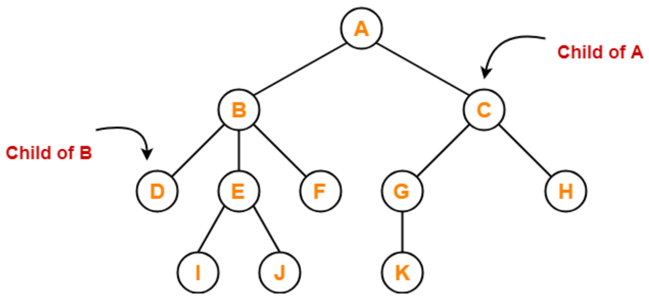
- The node which has a branch from it to any other node is called as a parent node.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

Child

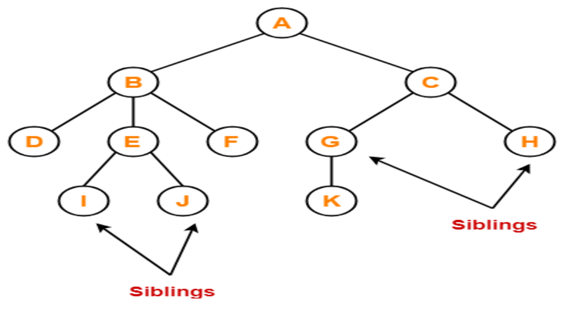
- The node which is a descendant of some node is called as a child node.
- All the nodes except root node are child nodes.



- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

Siblings

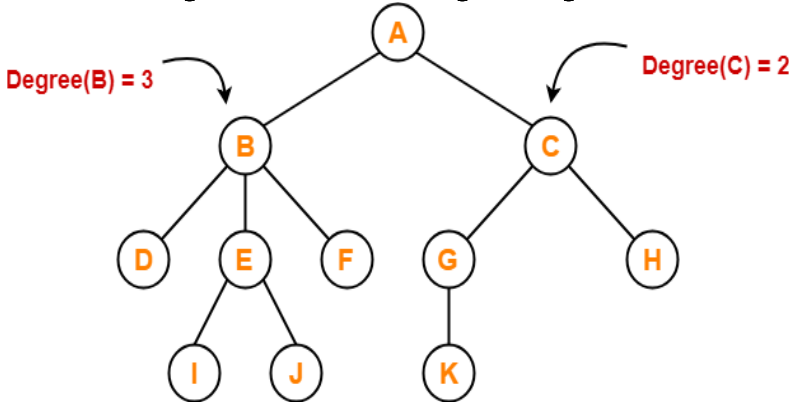
- Nodes which belong to the same parent are called as siblings.
- In other words, nodes with the same parent are sibling nodes.



- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

Degree

- Degree of a node is the total number of children of that node.
- Degree of a tree is the highest degree of a node among all the nodes in the tree

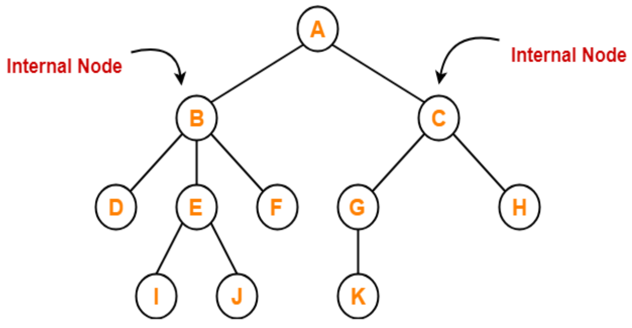


- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

.

Internal node

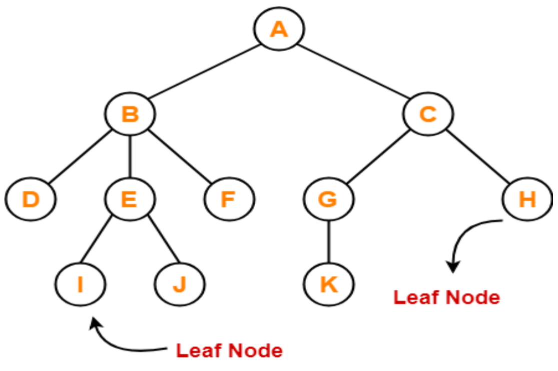
- The node which has at least one child is called as an internal node.
- Internal nodes are also called as non-terminal nodes.
- Every non-leaf node is an internal node.



Here, nodes A, B, C, E and G are internal nodes.

Leaf node

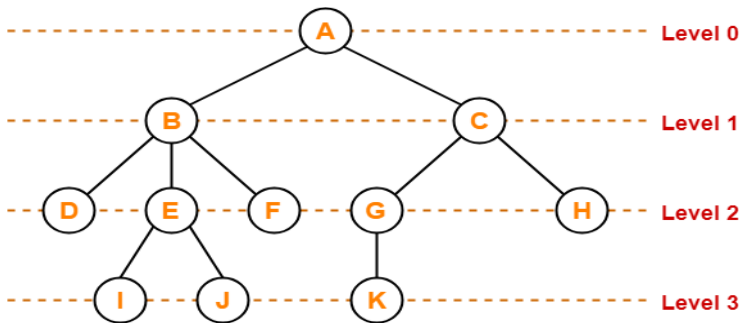
- The node which does not have any child is called as a leaf node.
- Leaf nodes are also called as external nodes or terminal nodes.



Here, nodes D, I, J, F, K and H are leaf nodes.

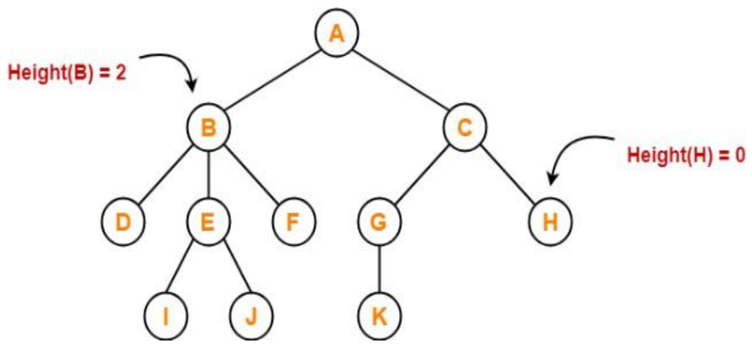
Level

- In a tree, each step from top to bottom is called as level of a tree.
- The level count starts with 0 and increments by 1 at each level or step.



Height

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
- Height of a tree is the height of root node.
- Height of all leaf nodes = 0



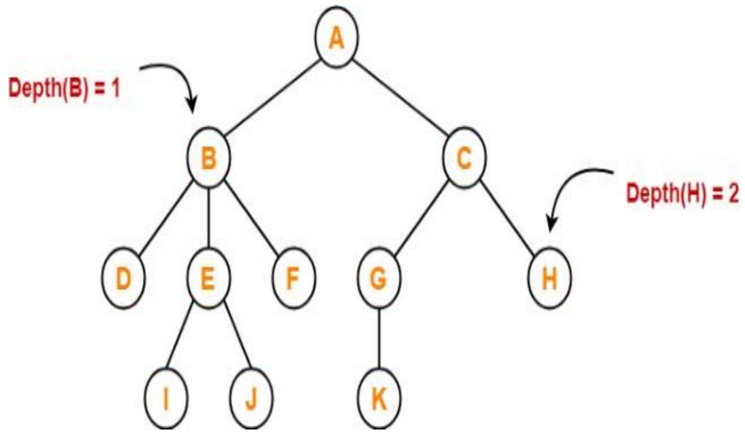
- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

Depth

- Total number of edges from root node to a particular node is called as depth of that node.
- Depth of a tree is the total number of edges from root node to a leaf node in the longest

path.

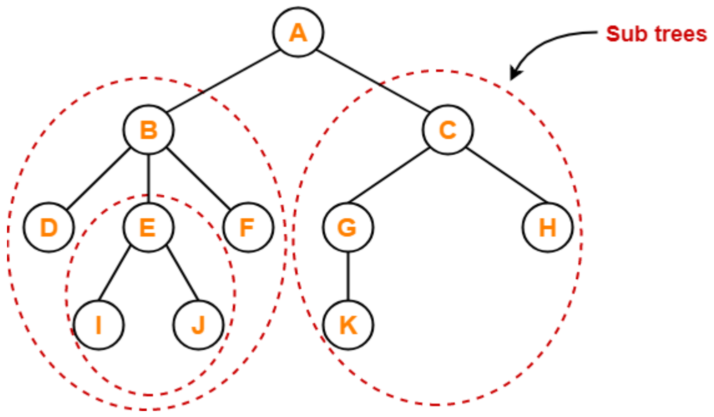
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

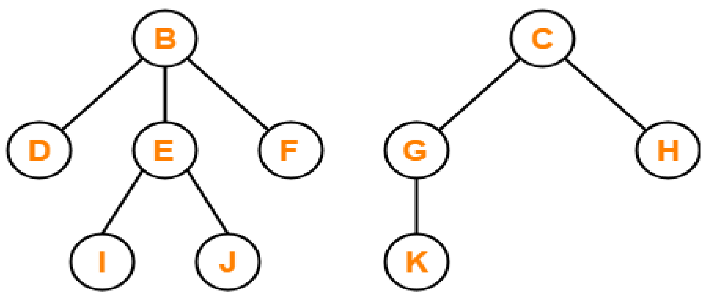
Subtree

- In a tree, each child from a node forms a subtree recursively.
- Every child node forms a subtree on its parent node.



Forest

- A forest is a set of disjoint trees.



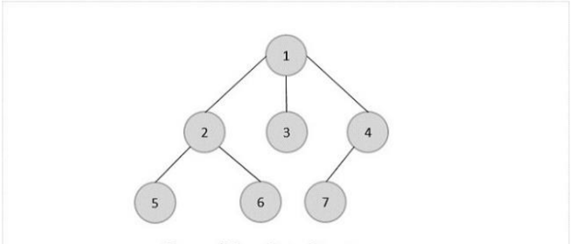
Forest

Types of Tree

- General Tree

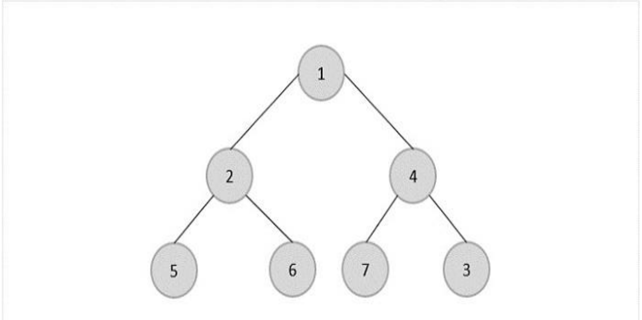
- Binary Tree
- Binary Search Tree
- AVL Tree
- Red-Black Tree
- N-ary Tree

General Trees
General trees are unordered tree data structures where the root node has minimum 0 or maximum 'n' subtrees.
The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.



General Tree Data Structure

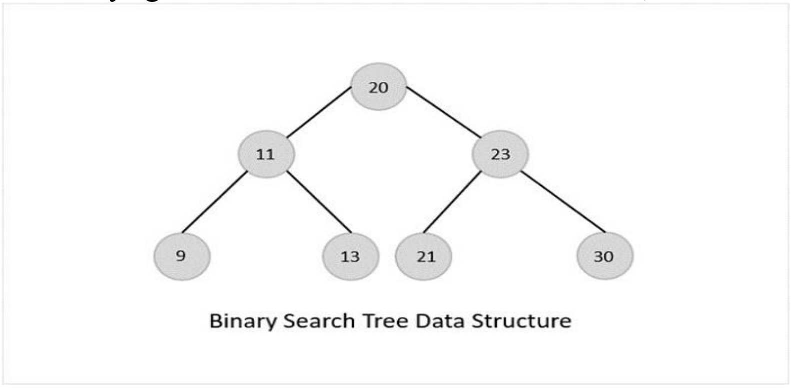
Binary Trees
Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree. Based on the number of children, binary trees are divided into three types.
Full Binary Tree
A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.
Complete Binary Tree
A complete binary tree is a binary tree type where all the leaf nodes must be on the same level. However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.
Perfect Binary Tree
A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.



Binary Tree Data Structure

Binary Search Trees
Binary Search Trees possess all the properties of Binary Trees including some extra properties of their own, based on some constraints, making them more efficient than binary trees.

The data in the Binary Search Trees (BST) is always stored in such a way that the values in the left subtree are always less than the values in the root node and the values in the right subtree are always greater than the values in the root node, i.e. left subtree < root node ≤ right subtree.



Advantages of BST

- Binary Search Trees are more efficient than Binary Trees since time complexity for performing various operations reduces.
- Since the order of keys is based on just the parent node, searching operation becomes simpler.
- The alignment of BST also favors Range Queries, which are executed to find values existing between two keys. This helps in the Database Management System.

Disadvantages of BST

The main disadvantage of Binary Search Trees is that if all elements in nodes are either greater than or lesser than the root node, the tree becomes skewed. Simply put, the tree becomes slanted to one side completely.
This skewness will make the tree a linked list rather than a BST, since the worst case time complexity for searching operation becomes O(n).
To overcome this issue of skewness in the Binary Search Trees, the concept of Balanced Binary Search Trees was introduced.

Memory Representation of a Tree

Trees can be represented in two common ways: **using arrays** and **using linked lists**.
1. **Array Representation**

In the array representation of a tree, nodes are stored sequentially in an array such that their relationships can be determined using index calculations.

(a) Binary Tree Representation

For a binary tree:

- The **root node** is stored at index 0 (or 1 for 1-based indexing).
- For a node at index i:
  - **Left child** is stored at index 2i + 1.
  - **Right child** is stored at index 2i + 2.
  - **Parent** is stored at index (i - 1) // 2.

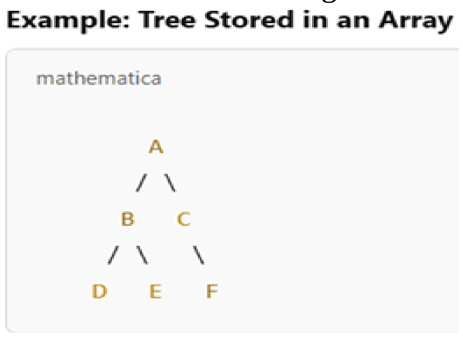
Example

For a binary tree like this:
markdown

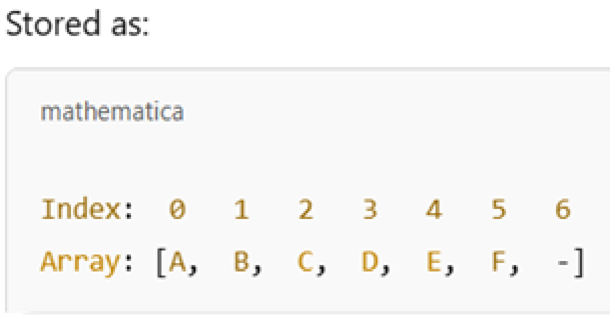
```
CopyEdit
      1
     / \
    2   3
   / \
  4   5
```

The array representation is:
`[1, 2, 3, 4, 5]`

- Trees (especially binary trees) can be stored in arrays using level order traversal.
- For a node at index **i**:
  - Left child is at index **2i + 1**.
  - Right child is at index **2i + 2**.


Example: Tree Stored in an Array

```
mathematica

        A
       / \
      B   C
     / \   \
    D   E   F
```


Stored as:

```
mathematica

Index:  0   1   2   3   4   5   6
Array: [A,  B,  C,  D,  E,  F,  -]
```

- B (index 1) → Left child of A (index 0)
- C (index 2) → Right child of A (index 0)
- D (index 3) → Left child of B (index 1)
- E (index 4) → Right child of B (index 1)
- F (index 5) → Right child of C (index 2)

(b) General Tree Representation

For a general tree (not binary), each node can store its children sequentially in the array, but you might need additional structures to track children (e.g., an adjacency list).

2. **Linked List Representation**

In this representation, each node is represented as an object or structure with pointers to its children.

(a) Binary Tree Representation

Each node in a binary tree contains:

- **Data**: The value of the node.
- **Left Pointer**: A pointer to the left child.
- **Right Pointer**: A pointer to the right child.

Structure for a binary tree node:

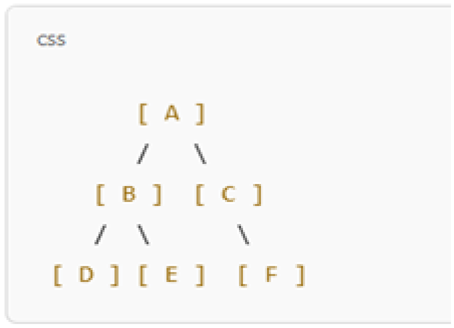
```
struct Node {
   int data;
   struct Node* left;
```

```
    struct Node* right;
};
```

Each node contains:

- Data
- Pointer to the left child
- Pointer to the right child (for binary trees)

Graphical Representation



```
        [ A ]
        /   \
     [ B ]  [ C ]
     /  \      \
  [ D ] [ E ]  [ F ]
```

(b) General Tree Representation

For a general tree, each node contains:

- **Data**: The value of the node.
- **Child Pointer**: A pointer to the first child.
- **Sibling Pointer**: A pointer to the next sibling.

Structure for a general tree node:

```
struct Node {
    int data;
    struct Node* firstChild;
    struct Node* nextSibling;
};
```

Example

For the same tree:
markdown
CopyEdit

```
        1
       / \
      2   3
     / \
    4   5
```

Linked list representation would look like:

- 1 → left(2) → right(3)
- 2 → left(4) → right(5)

Comparison

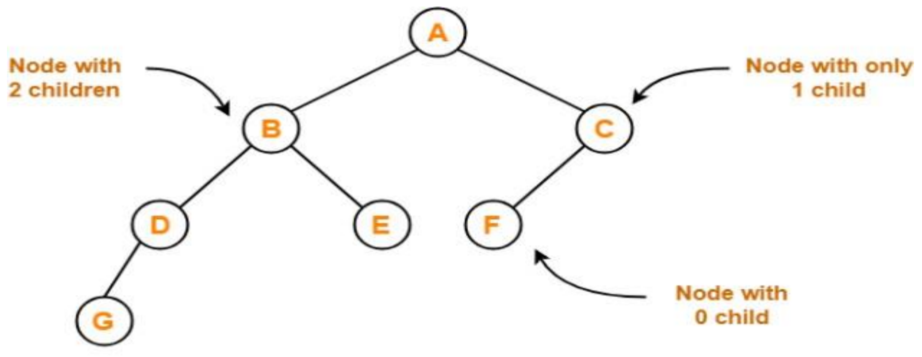
Aspect	**Array Representation**	**Linked List Representation**
Space Efficiency	Uses contiguous memory; size must be known in advance.	Efficient memory usage; grows dynamically.

Aspect	**Array Representation**	**Linked List Representation**
Ease of Access	Direct access using index calculations.	Traversal required for access.
Flexibility	Difficult to handle dynamic trees (insertion/deletion).	Easier to handle dynamic structures.
Use Cases	Suitable for complete binary trees.	Suitable for general and sparse trees.

Binary Tree:
A binary tree 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 a type of tree structure commonly used in computer science for efficient searching, sorting, and hierarchical data representation.
Binary tree is a special tree data structure in which each node can have at most 2 children.
Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children

Key Terms:

1. **Root Node**: The topmost node of the tree.
2. **Leaf Node**: A node with no children.
3. **Height**: The longest path from the root to a leaf.
4. **Depth**: The level of a node in the tree, starting from the root (depth = 0).
5. **Subtree**: A tree formed by any node and its descendants.
6. **Binary Tree Types**:
   - **Full Binary Tree**: Every node has 0 or 2 children.
   - **Complete Binary Tree**: All levels, except possibly the last, are completely filled, and nodes are as far left as possible.
   - **Perfect Binary Tree**: All internal nodes have two children, and all leaf nodes are at the same level.
   - **Skewed Binary Tree**: A tree where all nodes have only one child (either left or



Binary Tree Example

right).

Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal

- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

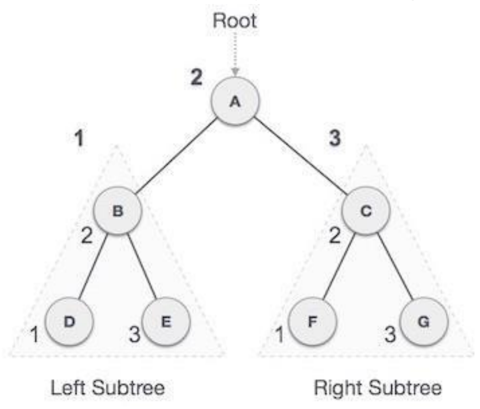
In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

Algorithm

Until all nodes are traversed −

```
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```

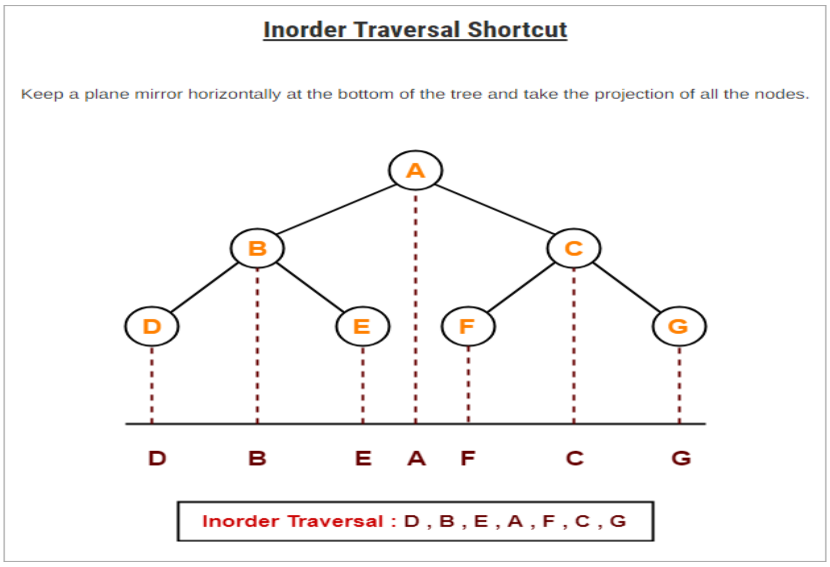


We start from A, and following in-order traversal, we move to its left subtree B.B is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −
D → B → E → A → F → C → G

```
#include <iostream>
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
struct node *root = NULL;
void insert(int data){
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;
   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;
   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;
      while(1) {
         parent = current;
         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;
            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }//go to right of the tree
         else {
            current = current->rightChild;
            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
void inorder_traversal(struct node* root){
   if(root != NULL) {
      inorder_traversal(root->leftChild);
      printf("%d ",root->data);
      inorder_traversal(root->rightChild);
   }
}
int main(){
   int i;
   int array[7] = { 27, 14, 35, 10, 19, 31, 42 };
   for(i = 0; i < 7; i++)
      insert(array[i]);
   printf("Inorder traversal: ");
   inorder_traversal(root);
   return 0;
}
```

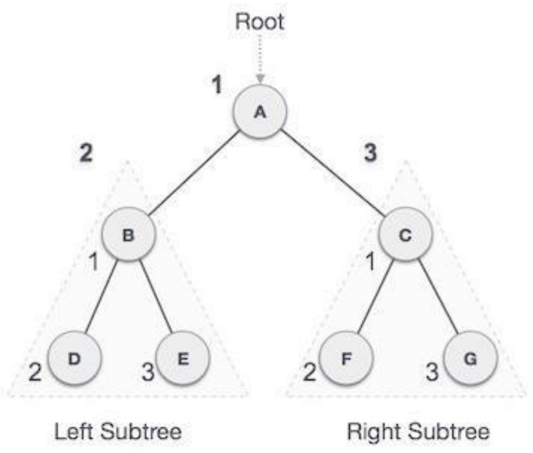
Output

Inorder traversal: 10 14 19 27 31 35 42


Inorder Traversal Shortcut
Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.
A
B
C
D
E
F
G
D B E A F C G
Inorder Traversal : D , B , E , A , F , C , G

Pre-order Traversal

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



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

A → B → D → E → C → F → G

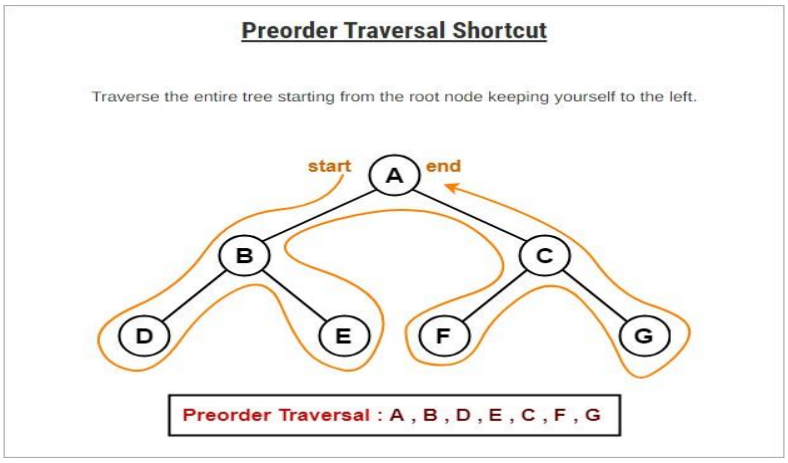
Algorithm

```
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

```
#include <iostream>
struct node {
  int data;
  struct node *leftChild;
  struct node *rightChild;
};
struct node *root = NULL;
void insert(int data){
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;
  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;
    while(1) {
      parent = current;
      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left
        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }//go to right of the tree
      else {
        current = current->rightChild;
        //insert to the right
        if(current == NULL) {
          parent->rightChild = tempNode;
          return;
        }
      }
    }
  }
}
void pre_order_traversal(struct node* root){
  if(root != NULL) {
    printf("%d ",root->data);
    pre_order_traversal(root->leftChild);
    pre_order_traversal(root->rightChild);
  }
}
int main(){
  int i;
  int array[7] = { 27, 14, 35, 10, 19, 31, 42 };
  for(i = 0; i < 7; i++)
     insert(array[i]);
  printf("Preorder traversal: ");
  pre_order_traversal(root);
  return 0;
}
```

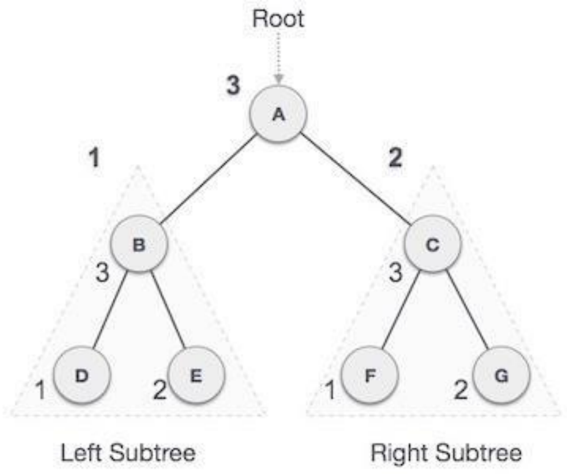
Output

```
Preorder traversal: 27 14 10 19 35 31 42
```


Preorder Traversal Shortcut
Traverse the entire tree starting from the root node keeping yourself to the left.
start
end
A
B
C
D
E
F
G
Preorder Traversal : A , B , D , E , C , F , G

Post-order Traversal

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



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

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.

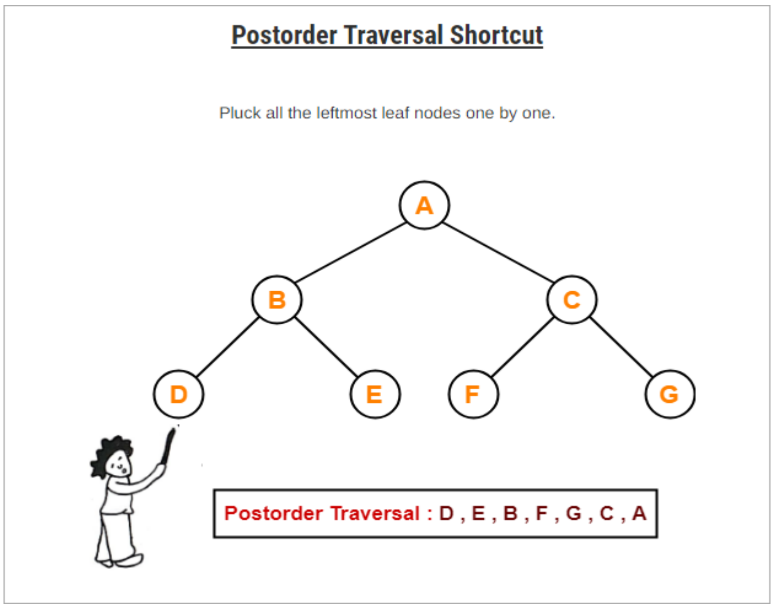
Step 3 − Visit root node.

```
#include <iostream>
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
struct node *root = NULL;
void insert(int data){
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;
   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;
      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }//go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

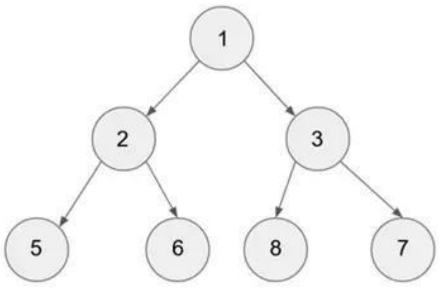
```
void post_order_traversal(struct node* root){
   if(root != NULL) {
      post_order_traversal(root->leftChild);
      post_order_traversal(root->rightChild);
      printf("%d ", root->data);
   }
}
int main(){
   int i;
   int array[7] = { 27, 14, 35, 10, 19, 31, 42 };
   for(i = 0; i < 7; i++)
      insert(array[i]);
   printf("Post order traversal: ");
   post_order_traversal(root);
   return 0;
}
```

Output

```
Post order traversal: 10 19 14 31 42 35 27
```


Postorder Traversal Shortcut
Pluck all the leftmost leaf nodes one by one.
A
B
C
D
E
F
G
Postorder Traversal : D , E , B , F , G , C , A

Traversal Algorithms

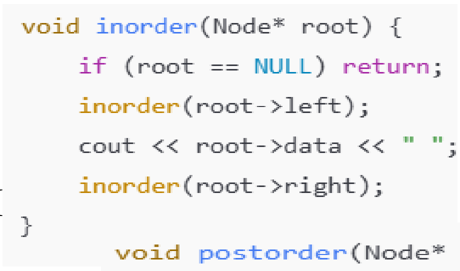


1. **Pre-Order (Root → Left → Right): [Preorder: 1 → 2 → 3 → 4 → 5 → 6 → 7]**
   - Visit root.

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

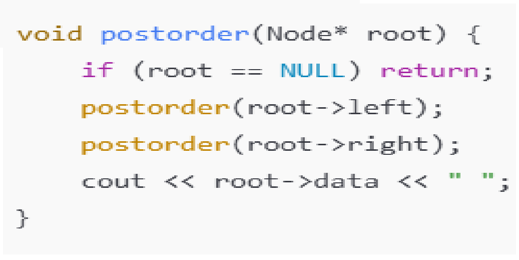
- Traverse left subtree.
- Traverse right subtree.

2. In-Order (Left → Root → Right): [In Order: 5 → 2 → 6 → 1 → 8 → 3 → 7]
   - Traverse left subtree.
   - Visit root.
   - Traverse right subtree.



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

3. Post-Order (Left → Right → Root): [
   - Traverse left subtree.
   - Traverse right subtree.
   - Visit root.



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

Construction of Binary Tree

Binary trees can be constructed using traversal sequences such as **In-Order and Pre-Order** or **In-Order and Post-Order**. These sequences uniquely define a binary tree.

1. **Construction from In-Order and Pre-Order Traversals**

- **In-Order**: Left Subtree → Root → Right Subtree.
- **Pre-Order**: Root → Left Subtree → Right Subtree.
- Steps:
  1. Identify the root node (first element of the Pre-Order sequence).
  2. Locate the root in the In-Order sequence. Elements to the left of the root in In-Order form the left subtree, and elements to the right form the right subtree.
  3. Recursively repeat the process for the left and right subtrees.
- Example: In-Order: `D B E A F C` Pre-Order: `A B D E C F`
  Steps:
  - Root: `A` (first in Pre-Order)
  - Left Subtree (In-Order): `D B E`, Right Subtree (In-Order): `F C`
  - Recursively construct left and right subtrees.
-

```
def build_tree(in_order, pre_order):
    if not in_order or not pre_order:
        return None

    root_data = pre_order.pop(0)
    root = TreeNode(root_data)
    root_index = in_order.index(root_data)

    root.left = build_tree(in_order[:root_index], pre_order)
    root.right = build_tree(in_order[root_index+1:], pre_order)
    return root
```

2. **Construction from In-Order and Post-Order Traversals**

- **In-Order**: Left Subtree → Root → Right Subtree.
- **Post-Order**: Left Subtree → Right Subtree → Root.
- Steps:
    1. Identify the root node (last element of the Post-Order sequence).
    2. Locate the root in the In-Order sequence. Elements to the left of the root in In-Order form the left subtree, and elements to the right form the right subtree.
    3. Recursively repeat the process for the left and right subtrees.
- Example: In-Order: `D B E A F C` Post-Order: `D E B F C A`
  Steps:
    - Root: `A` (last in Post-Order)
    - Left Subtree (In-Order): `D B E`, Right Subtree (In-Order): `F C`
    - Recursively construct left and right subtrees.
- Code:

```
def build_tree(in_order, post_order):
    if not in_order or not post_order:
        return None

    root_data = post_order.pop()
    root = TreeNode(root_data)
    root_index = in_order.index(root_data)

    root.right = build_tree(in_order[root_index+1:], post_order)
    root.left = build_tree(in_order[:root_index], post_order)

    return root
```

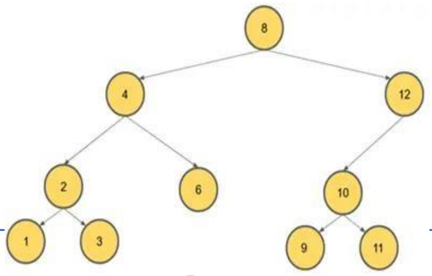
NOTE :

Traversal Type	Order
Pre-Order	Root → Left Subtree → Right Subtree
In-Order	Left Subtree → Root → Right Subtree
Post-Order	Left Subtree → Right Subtree → Root

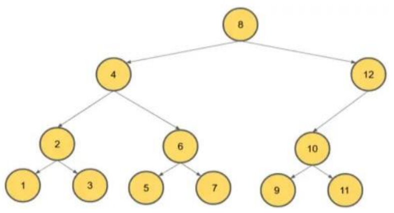
Types of Binary Tree

Here are some important types of Binary Tree:

Full Binary Tree: Each node can have 0 or 2 child nodes in this binary tree. Only one child node is not allowed in this type of binary tree. So, except for the leaf node, all nodes will have 2 children.

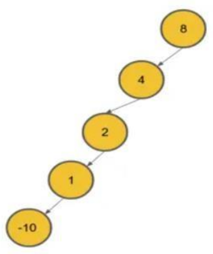


Complete Binary Tree: Each node can have 0 or 2 nodes. It seems like the Full Binary Tree, but all the leaf elements are lean to the left subtree, whereas in the full binary tree node can be in the right or left subtree

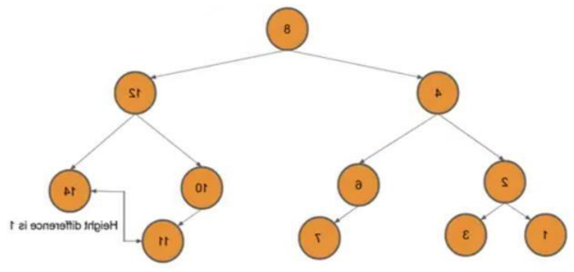


Perfect Binary Tree: All the nodes must have 0 or 2 nodes, and all the leaf nodes should be at the same level or height. The above example of a full binary tree structure is not a Perfect Binary Tree because node 6 and node 1,2,3 are not in the same height. But the example of the Complete Binary Tree is a perfect binary tree.

Degenerate Binary Tree: Every node can have only a single child. All the operations like searching, inserting, and deleting take O(N) time



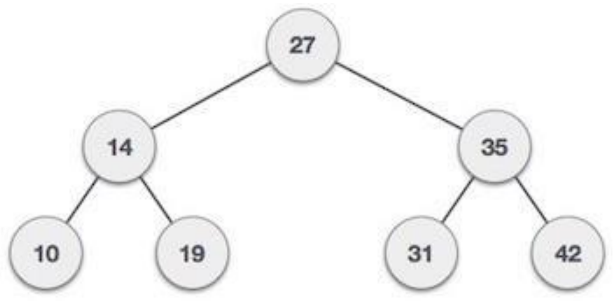
Balanced Binary Tree: Here this binary tree, the height difference of left and right subtree is at most 1. So, while adding or deleting a node, we need to balance the tree's height again. This type of Self-Balanced Binary Tree is



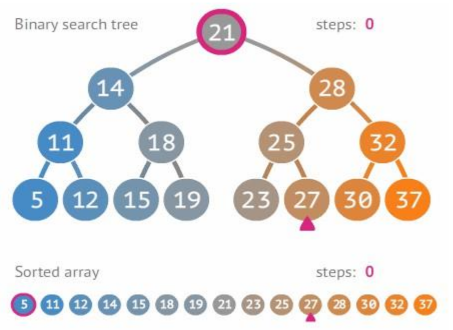
Binary Tree Representation

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

Following is a pictorial representation of BST −



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



A **BST** is a binary tree where:

- Left subtree contains nodes with values **less than** the root.
- Right subtree contains nodes with values **greater than** the root.

Defining a Node

Define a node that stores some data, and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Binary Search Tree (BST)

A **Binary Search Tree** is a binary tree where:

1. Each node contains a key.
2. The key in the left subtree of a node is less than the node's key.
3. The key in the right subtree of a node is greater than the node's key.
4. Both the left and right subtrees must also be binary search trees.

Operations

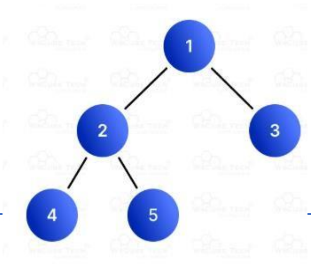
Binary trees support several fundamental operations, including insertion, deletion, searching, and traversal:

1. Insertion

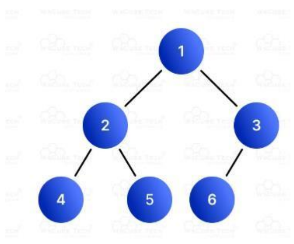
Insertion involves adding a new node to the binary tree. In a binary tree, a new node is usually inserted at the first available position in level order to maintain the completeness of the tree.

Example:

Let's insert the value 6 into the following binary tree:



After insertion, the binary tree will look like this:

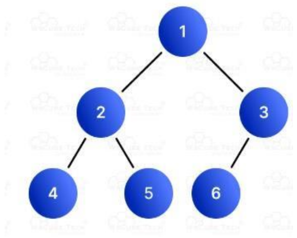


2. Deletion

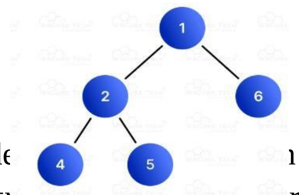
Deletion involves removing a node from the binary tree. In a binary tree, the node to be deleted is replaced by the deepest and rightmost node to maintain the tree's structure.

Example:

Let's delete the value 3 from the following binary tree:



After deletion, the binary tree will look like this:

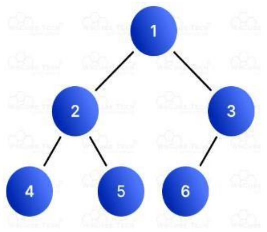


3. Search

Searching involves finding a node [illegible] the binary tree. The search operation can be implemented using any traversal method (in-order, pre-order, post- order, or level-order).

Example:

Let's search for the value 5 in the following binary tree:
Using level-order traversal:



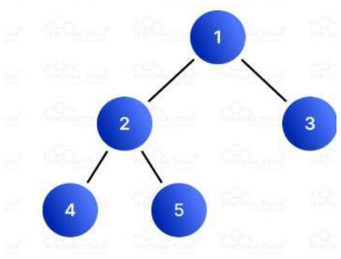
- Visit node 1
- Visit node 2
- Visit node 3
- Visit node 4
- Visit node 5 (found)

4. Traversal

Traversal involves visiting all the nodes in the binary tree in a specific order. The main traversal methods are in-order, pre-order, post-order, and level-order.

Example:

Consider the following binary tree:



- In-order Traversal (Left, Root, Right): 4, 2, 5, 1, 3
- Pre-order Traversal (Root, Left, Right): 1, 2, 4, 5, 3
- Post-order Traversal (Left, Right, Root): 4, 5, 2, 3, 1
- Level-order Traversal (Breadth-First): 1, 2, 3, 4, 5

Insertion of a Node in a BST
Steps to Insert a Node:

1. **Start at the Root**:
   - Compare the key to be inserted with the root node's key.
2. **Traverse Left or Right**:
   - If the key is smaller, move to the left subtree.
   - If the key is larger, move to the right subtree.
3. **Find the Empty Spot**:
   - Repeat the above steps until you find a null (empty) position.
4. **Insert the Node**:
   - Insert the new node at the found position.

Algorithm

1. Start at the root node.
2. Compare the key to be inserted with the current node's key:
   - If the key is **less**, move to the left subtree.
   - If the key is **greater**, move to the right subtree.
3. When you reach a null pointer (empty spot), insert the new node at that position.
4. Repeat recursively until the new node is placed.

Example

Insert the key `10` into the following BST:
markdown
CopyEdit
```
     15
    / \
  10   20
 /
8
```

If the tree is empty, `10` becomes the root. Otherwise, traverse as follows:

- Compare 10 with 15: Move left.
- Compare 10 with 10: The left child exists, continue traversal to find an empty spot.

```
#include <iostream>
using namespace std;
// Definition of the Node
struct Node {
    int key;
    Node* left;
    Node* right;

    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};
// Function to insert a node in a BST
Node* insert(Node* root, int key) {
    // If the tree is empty, return a new node
    if (root == nullptr)
        return new Node(key);
    // Otherwise, traverse the tree
    if (key < root->key) {
        root->left = insert(root->left, key);  // Go to the left subtree
    } else if (key > root->key) {
        root->right = insert(root->right, key);  // Go to the right subtree
    }

    // Return the unchanged root node
    return root;
}
// Function for in-order traversal of the BST
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
int main() {
    Node* root = nullptr;
    int keys[] = {15, 10, 20, 8, 12, 17, 25};

    for (int key : keys) {
        root = insert(root, key);
    }
    cout << "In-order traversal after insertion: ";
    inorder(root);
    cout << endl;
    return 0;
}
```

Deletion of a Node in a BST

Cases for Deletion:

1. **Node is a Leaf (No Children)**:
    - Simply remove the node.
2. **Node has One Child**:
    - Replace the node with its child.
3. **Node has Two Children**:
    - Find the **in-order successor** (the smallest value in the right subtree) or the **in-order predecessor** (the largest value in the left subtree).
    - Replace the node's key with the successor's key.
    - Recursively delete the successor.

Example

Delete the key `15` from the following BST:

markdown

CopyEdit

```
      15
     /  \
   10    20
  /
 8
```

- 15 has two children.
- Find the **in-order successor**: The smallest value in the right subtree is 20.
- Replace 15 with 20 and delete 20 from the right subtree.

Resulting BST:

```
     20
    /  \
   10   null
  /
 8
```

Algorithm

1. Start at the root node and search for the node to be deleted.
2. Three cases arise:
    - **Node is a Leaf (No Children)**:
        - Remove the node by setting the pointer to nullptr.
    - **Node has One Child**:
        - Replace the node with its child.
    - **Node has Two Children**:
        - Find the **in-order successor** (smallest value in the right subtree).
        - Replace the node's key with the successor's key.
        - Recursively delete the successor.

```
// Function to find the in-order successor (smallest value in the right subtree)
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr) {
        current = current->left;
    }
    return current;
}
// Function to delete a node in a BST
Node* deleteNode(Node* root, int key) {
    // Base case: the tree is empty
    if (root == nullptr) {
        return root;
    }
    // Traverse the tree to find the node to delete
    if (key < root->key) {
        root->left = deleteNode(root->left, key);  // Go to the left subtree
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);  // Go to the right subtree
    } else {
        // Node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        // Node with two children: Get the in-order successor
        Node* temp = minValueNode(root->right);
        // Copy the in-order successor's key to this node
        root->key = temp->key;
        // Delete the in-order successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
int main() {
    Node* root = nullptr;
    int keys[] = {15, 10, 20, 8, 12, 17, 25};

    for (int key : keys) {
        root = insert(root, key);
    }
    cout << "In-order traversal after insertion: ";
    inorder(root);
    cout << endl;
    // Deleting a node
    cout << "Deleting node 10..." << endl;
    root = deleteNode(root, 10);
    cout << "In-order traversal after deletion: ";
    inorder(root);
    cout << endl;
    return 0;
}
```

Example Output

For the insertion example with keys `[15, 10, 20, 8, 12, 17, 25]`:

After insertion (in-order traversal):

```
8 10 12 15 17 20 25
```

After deletion (key `10`):

```
8 12 15 17 20 25
```

<u>Advanced Tree Structures: AVL and B-Trees</u>

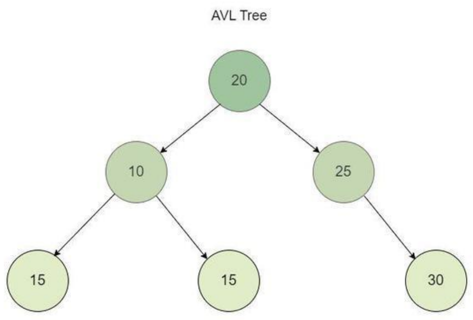
1. AVL Trees

Definition

An **AVL Tree** is a self-balancing binary search tree (BST). It ensures that the height difference (balance factor) between the left and right subtrees of any node is at most 1. Named after its inventors **Adelson-Velsky** and **Landis**.

<u>**AVL Tree (Adelson-Velsky and Landis Tree)**</u>

- A **self-balancing BST** where the **height difference** of left and right subtrees (Balance



- **Rotations** are used to maintain balance:
    1. Left-Left (LL) Rotation
    2. Right-Right (RR) Rotation
    3. Left-Right (LR) Rotation
    4. Right-Left (RL) Rotation

Properties

1. **Binary Search Tree Property**:
    - For each node, keys in the left subtree are smaller, and keys in the right subtree are larger.
2. **Balance Factor**:
    - Defined as: Balance Factor=Height of Left Subtree−Height of Right Subtree\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}Balance Factor=Height of Left Subtree−Height of Right Subtree
    - The balance factor must be **-1, 0, or 1** for all nodes.
3. **Height**:
    - The height of an AVL tree is logarithmic: Height∝log⁡(n)\text{Height} \propto \log(n)Height∝log(n)

Rotations for Balancing

When the balance factor of a node becomes **outside the range [-1, 1]**, rotations are performed:

1. **Single Rotations**:
    - **Right Rotation (LL Imbalance)**: Performed when the left subtree of the left child is too tall.
    - **Left Rotation (RR Imbalance)**: Performed when the right subtree of the right child is too tall.
2. **Double Rotations**:
    - **Left-Right Rotation (LR Imbalance)**: Performed when the left subtree of the right child is too tall.

- **Right-Left Rotation (RL Imbalance)**: Performed when the right subtree of the left child is too tall.

Operations

1. **Insertion**:
   - Insert as in a normal BST.
   - Update balance factors and rebalance using rotations if necessary.
2. **Deletion**:
   - Delete as in a normal BST.
   - Update balance factors and rebalance using rotations if necessary.

Applications

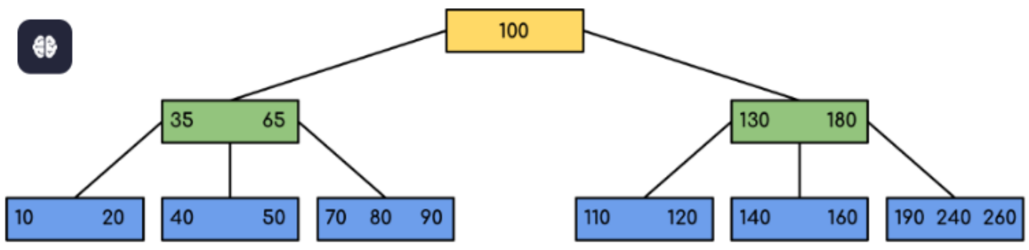
- Databases requiring fast searches and insertions.
- File systems for managing structured data.
- Memory management in operating systems.
- Database Indexing
- Memory Management
- File Systems
- Network Routing
- Priority Queues
- Compiler Design
- Geospatial Databases
- Event Scheduling Systems
- Artificial Intelligence and Machine Learning
- Telecommunication Systems

3. B-Trees

Definition

A **B-Tree** is a self-balancing **m-ary search tree** where nodes can have multiple keys and children. It is optimized for minimizing disk I/O operations, making it ideal for databases and file systems.

- A **self-balancing search tree** used for large data storage (e.g., databases, file systems).
- Each node has multiple children and stores multiple keys.
- **Balanced** because all leaves are at the same level.



Properties

1. **M-ary Tree**:
   - A node can have at most **m children** and contain **m-1 keys**.
2. **Balance**:
   - All leaf nodes are at the same level, ensuring the tree remains balanced.

3. **Key Order**:
    - Keys are stored in sorted order, and child pointers split the key ranges.
4. **Node Capacity**:
    - A node contains at least ⌈m/2⌉−1\lceil m/2 \rceil - 1⌈m/2⌉−1 keys and at most m−1m-1m−1 keys (except the root, which can have fewer keys).

Operations

1. **Search**:
    - Similar to binary search but extended to multiple keys per node.
2. **Insertion**:
    - Insert the key into the appropriate leaf node.
    - If the node overflows (more than m−1m-1m−1 keys), split the node into two and promote the middle key to the parent.
3. **Deletion**:
    - Delete the key from the appropriate node.
    - If a node underflows (fewer than ⌈m/2⌉−1\lceil m/2 \rceil - 1⌈m/2⌉−1 keys), redistribute keys or merge nodes.
4. **Splitting and Merging**:
    - Ensure balance during insertion and deletion by splitting or merging nodes.

Applications

- Databases (e.g., relational databases like MySQL use B+ trees).
- File systems (e.g., NTFS, ext4).
- Indexing large datasets.
- 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.

Comparison of AVL and B-Trees

Feature	AVL Trees	B-Trees
Structure	Binary tree (each node has ≤2 children)	M-ary tree (each node has ≤m children)
Balance Factor	Ensures balance by height	Ensures balance by filling nodes
Storage	Each node stores 1 key	Each node stores multiple keys
Applications	In-memory data structures	Disk-based storage and databases
Rebalancing	Uses rotations	Uses splitting and merging
Efficiency	Better for smaller datasets	Better for large datasets