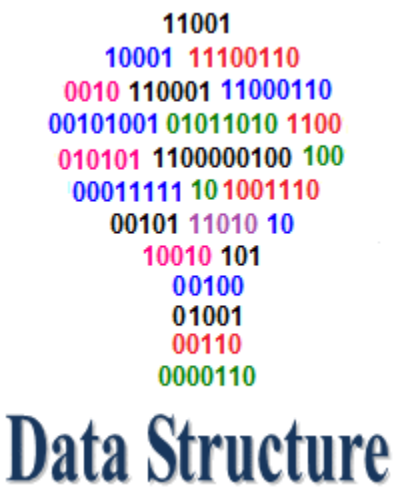
Data Structures Tutorial



Data Structures (DS) tutorial provides basic and advanced concepts of Data Structure. Our Data Structure tutorial is designed for beginners and professionals.

Data Structure is a way to store and organize data so that it can be used efficiently.

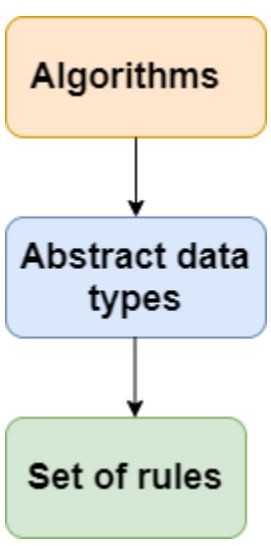
Our Data Structure tutorial includes all topics of Data Structure such as Array, Pointer, Structure, Linked List, Stack, Queue, Graph, Searching, Sorting, Programs, etc.

What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.



Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure

- Non-linear data structure

Linear Data Structure

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Which Data Structure?

A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time. For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the implementation. Therefore, we conclude that we require some data structure to implement a particular ADT.

An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part. Now the question arises: how can one get to know which data structure to be used for a particular ADT?.

As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space. For example, the Stack ADT can be implemented by both Arrays and linked list. Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.

Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structure provides reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Data Structure

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned.

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arrise the following problems:

Processor speed: To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

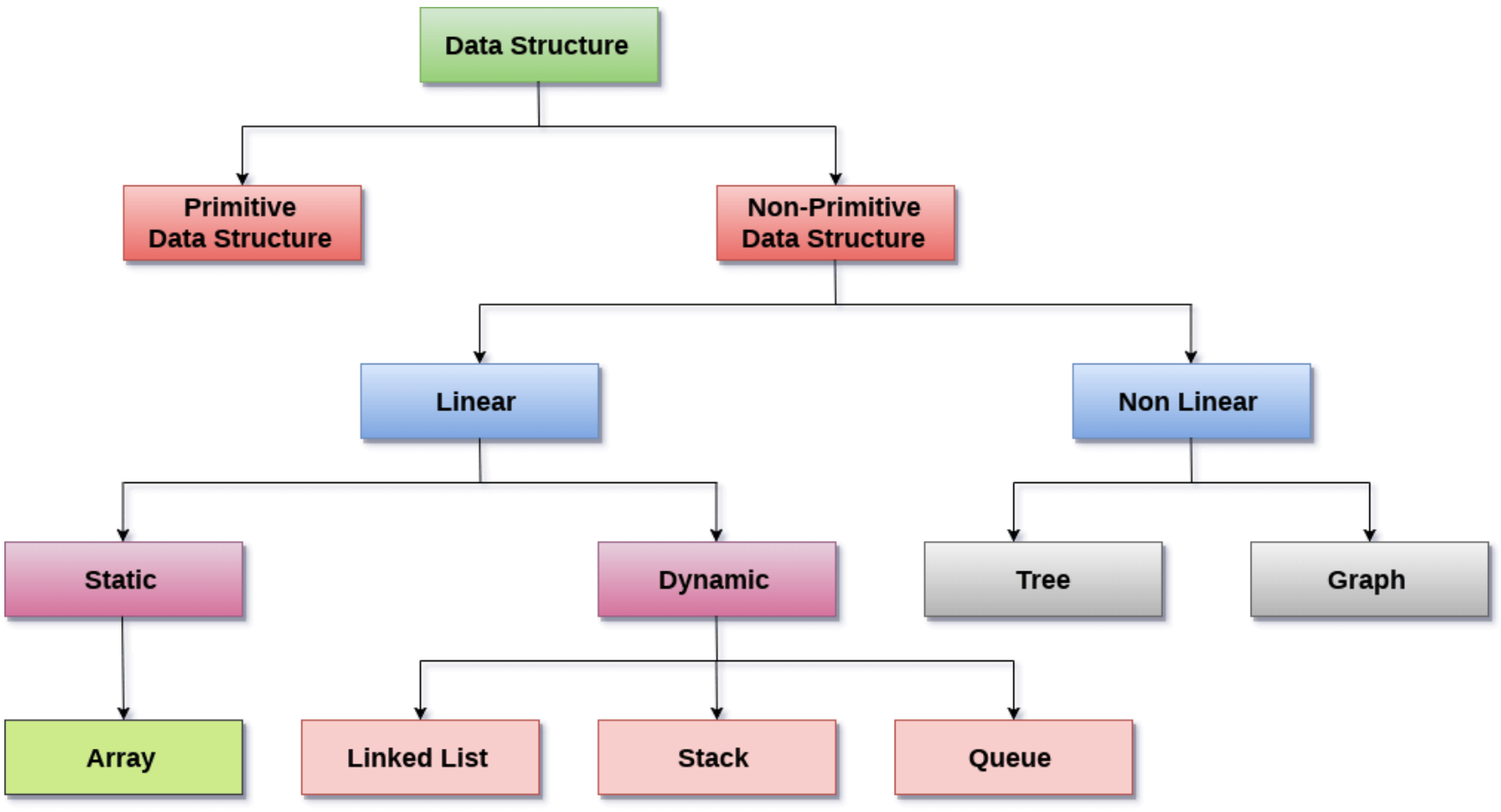
Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

DS Algorithm

What is an Algorithm?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Characteristics of an Algorithm

The following are the characteristics of an algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Why do we need Algorithms?

We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Let's understand the algorithm through a real-world example. Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:

Step 1: First, we will cut the lemon into half.

Step 2: Squeeze the lemon as much you can and take out its juice in a container.

Step 3: Add two tablespoon sugar in it.

Step 4: Stir the container until the sugar gets dissolved.

Step 5: When sugar gets dissolved, add some water and ice in it.

Step 6: Store the juice in a fridge for 5 to minutes.

Step 7: Now, it's ready to drink.

The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Now we will look an example of an algorithm in programming.

We will write an algorithm to add two numbers entered by the user.

The following are the steps required to add two numbers entered by the user:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.

Step 5: Print sum

Step 6: Stop

Factors of an Algorithm

The following are the factors that we need to consider for designing an algorithm:

- **Modularity:** If any problem is given and we can break that problem into small-small modules or small-small steps, which is a basic definition of an algorithm, it means that this feature has been perfectly designed for the algorithm.
- **Correctness:** The correctness of an algorithm is defined as when the given inputs produce the desired output, which means that the algorithm has been designed algorithm. The analysis of an algorithm has been done correctly.

- **Maintainability:** Here, maintainability means that the algorithm should be designed in a very simple structured way so that when we redefine the algorithm, no major change will be done in the algorithm.
- **Functionality:** It considers various logical steps to solve the real-world problem.
- **Robustness:** Robustness means that how an algorithm can clearly define our problem.
- **User-friendly:** If the algorithm is not user-friendly, then the designer will not be able to explain it to the programmer.
- **Simplicity:** If the algorithm is simple then it is easy to understand.
- **Extensibility:** If any other algorithm designer or programmer wants to use your algorithm then it should be extensible.

Importance of Algorithms

1. **Theoretical importance:** When any real-world problem is given to us and we break the problem into small-small modules. To break down the problem, we should know all the theoretical aspects.
2. **Practical importance:** As we know that theory cannot be completed without the practical implementation. So, the importance of algorithm can be considered as both theoretical and practical.

Issues of Algorithms

The following are the issues that come while designing an algorithm:

- **How to design algorithms:** As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.
- **How to analyze algorithm efficiency**

Approaches of Algorithm

The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:

- **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:
    1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
    2. **Sacrificing:** As soon as the best solution is found, then it will stop.
- **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.
- **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But, there are very rare cases in which it provides the optimal solution.
- **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
    1. It breaks down the problem into a subproblem to find the optimal solution.
    2. After breaking down the problem, it finds the optimal solution out of these subproblems.
    3. Stores the result of the subproblems is known as memorization.
    4. Reuse the result so that it cannot be recomputed for the same subproblems.
    5. Finally, it computes the result of the complex program.
- **Branch and Bound Algorithm:** The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.
- **Randomized Algorithm:** As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm?. In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.
- **Backtracking:** Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

The major categories of algorithms are given below:

- **Sort:** Algorithm developed for sorting the items in a certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

Algorithm Analysis

The algorithm can be analyzed in two levels, i.e., first is before creating the algorithm, and second is after creating the algorithm. The following are the two analysis of an algorithm:

- Priori Analysis: Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.
- Posterior Analysis: Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
1. sum=0;
2. // Suppose we have to calculate the sum of n numbers.
3. for i=1 to n
4. sum=sum+i;
5. // when the loop ends then sum holds the sum of the n numbers
6. return sum;
```

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

- **Space complexity:** An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

So,

Space complexity = Auxiliary space + Input size.

Types of Algorithms

The following are the types of algorithm:

- **Search Algorithm**

- **Sort Algorithm**

Search Algorithm

On each day, we search for something in our day to day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search the data in an array:

- **Linear search**
- **Binary search**

Linear Search

Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found. It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1. This algorithm can be implemented on the unsorted list.

Binary Search

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

Sorting Algorithms

Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.

Why do we need a sorting algorithm?

- An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
- It produces information in a sorted order, which is a human-readable format.
- Searching a particular element in a sorted list is faster than the unsorted list.

Asymptotic Analysis

As we know that data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space. So, the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space. Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm.

The main question arises in our mind that on what basis should we compare the time complexity of data structures?. The time complexity can be compared based on operations performed on them. Let's consider a simple example.

Suppose we have an array of 100 elements, and we want to insert a new element at the beginning of the array. This becomes a very tedious task as we first need to shift the elements towards the right, and we will add new element at the starting of the array.

Suppose we consider the linked list as a data structure to add the element at the beginning. The linked list contains two parts, i.e., data and address of the next node. We simply add the address of the first node in the new node, and head pointer will now point to the newly added node. Therefore, we conclude that adding the data at the beginning of the linked list is faster than the arrays. In this way, we can compare the data structures and select the best possible data structure for performing the operations.

How to find the Time Complexity or running time for performing the operations?

The measuring of the actual running time is not practical at all. The running time to perform any operation depends on the size of the input. Let's understand this statement through a simple example.

Suppose we have an array of five elements, and we want to add a new element at the beginning of the array. To achieve this, we need to shift each element towards right, and suppose each element takes one unit of time. There are five elements, so five units of time would be taken. Suppose there are 1000 elements in an array, then it takes 1000 units of time to shift. It concludes that time complexity depends upon the input size.

Therefore, if the input size is n, then f(n) is a function of n that denotes the time complexity.

How to calculate f(n)?

Calculating the value of f(n) for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their f(n) values. We can compare the data structures by comparing their f(n) values. We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes. Now, how to find f(n).

Let's look at a simple example.

$f(n) = 5n^2 + 6n + 12$

where n is the number of instructions executed, and it depends on the size of the input.

When n=1

% of running time due to $5n^2 = \frac{5}{5+6+12} * 100 = 21.74\%$

% of running time due to $6n = \frac{6}{5+6+12} * 100 = 26.09\%$

% of running time due to $12 = \frac{12}{5+6+12} * 100 = 52.17\%$

From the above calculation, it is observed that most of the time is taken by 12. But, we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of f(n).

n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

As we can observe in the above table that with the increase in the value of n, the running time of $5n^2$ increases while the running time of 6n and 12 also decreases. Therefore, it is observed that for larger values of n, the squared term consumes almost 99% of the time. As the n^2 term is contributing most of the time, so we can eliminate the rest two terms.

Therefore,

$f(n) = 5n^2$

Here, we are getting the approximate time complexity whose result is very close to the actual result. And this approximate measure of time complexity is known as an Asymptotic complexity. Here, we are not calculating the exact running time, we are eliminating the unnecessary terms, and we are just considering the term which is taking most of the time.

In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.

It is used to mathematically calculate the running time of any operation inside an algorithm.

Example: Running time of one operation is x(n) and for another operation, it is calculated as f(n2). It refers to running time will increase linearly with an increase in 'n' for the first operation, and running time will increase exponentially for the second operation. Similarly, the running time of both operations will be the same if n is significantly small.

Usually, the time required by an algorithm comes under three types:

Worst case: It defines the input for which the algorithm takes a huge time.

Average case: It takes average time for the program execution.

Best case: It defines the input for which the algorithm takes the lowest time

Asymptotic Notations

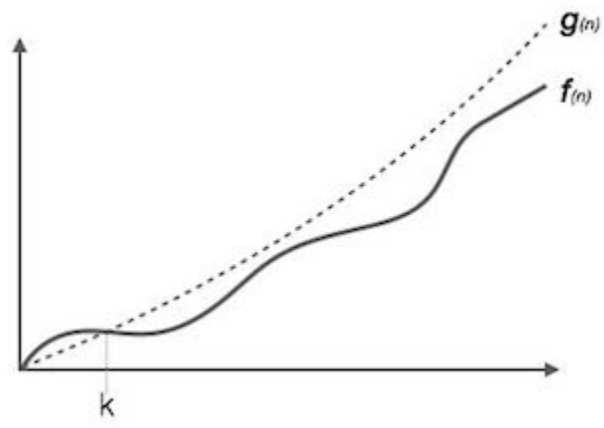
The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (?)
- Omega Notation (Ω)
- Theta Notation (θ)

Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



For example:

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)≤c.g(n) for all n≥no

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

Let's understand through examples

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

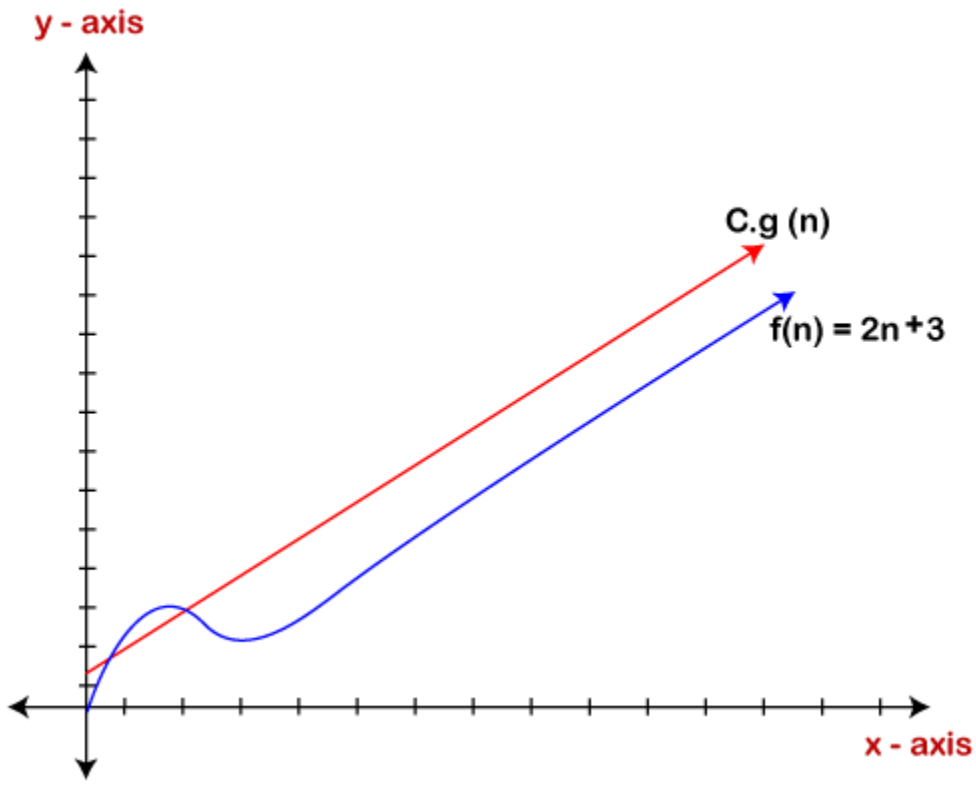
If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)>=c.g(n) for all n≥no and c>0

Let's consider a simple example.

If f(n) = 2n+3, g(n) = n,

Is f(n)= **Ω** (g(n))?

It must satisfy the condition:

f(n)>=c.g(n)

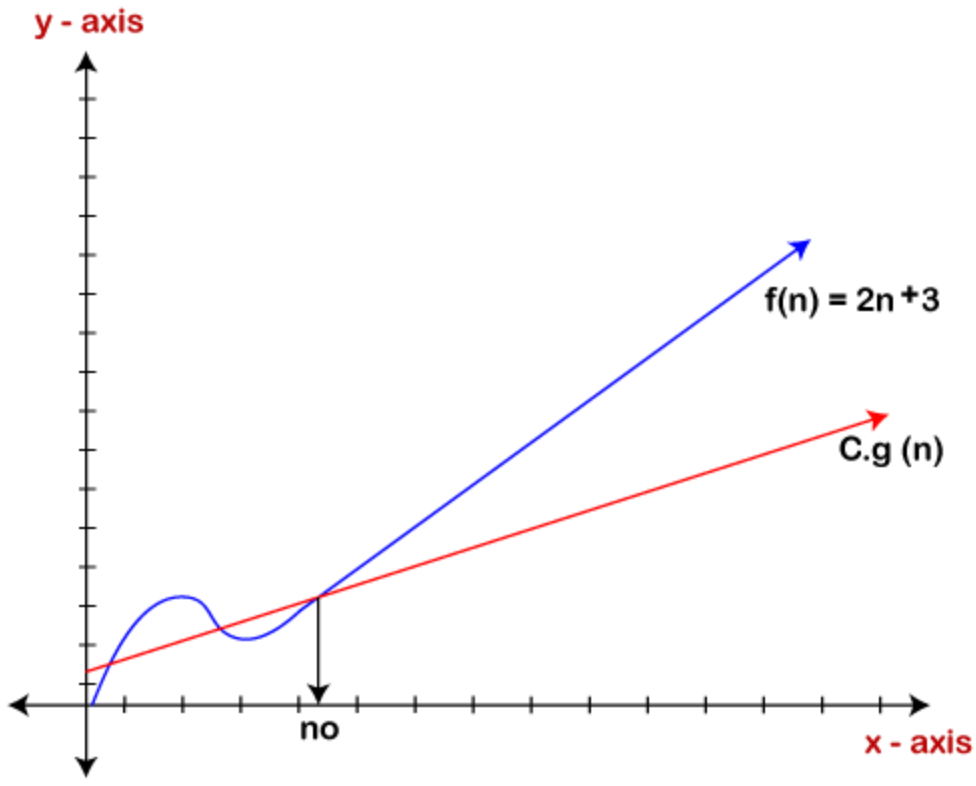
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

2n+3>=c*n

Suppose c=1

2n+3>=n (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.



As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let's understand the big theta notation mathematically:

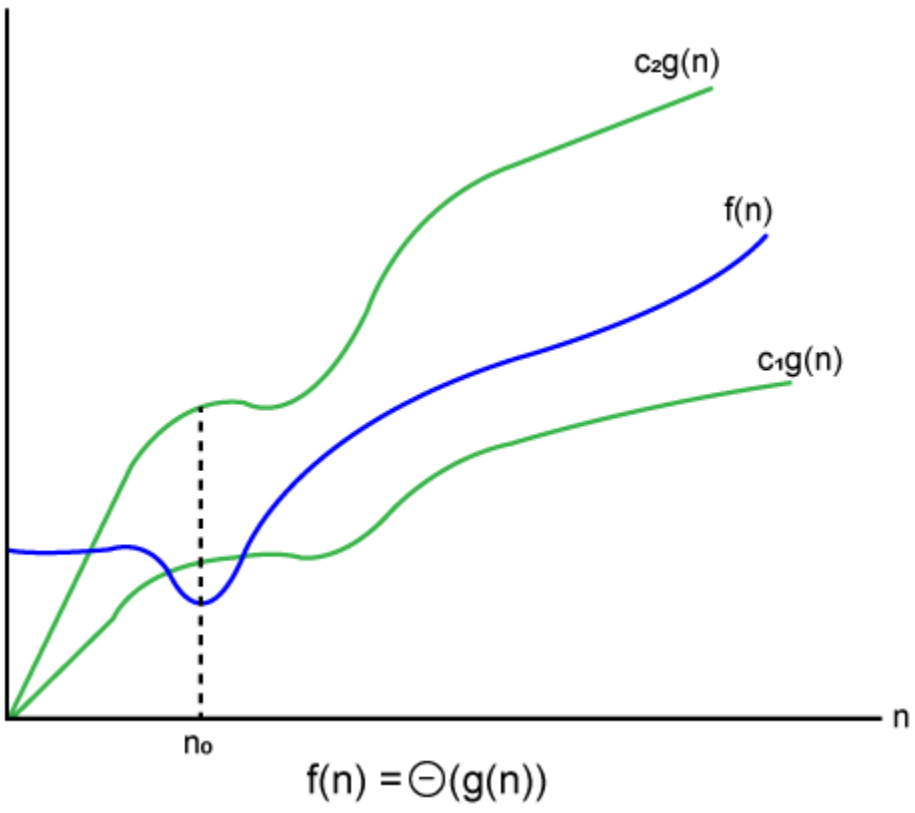
Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

f(n)= θg(n)

The above condition is satisfied only if when

c1.g(n)<=f(n)<=c2.g(n)

where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



Let's consider the same example where
f(n)=2n+3
g(n)=n

As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).

c1.g(n)<=f(n)<=c2.g(n)

Replace g(n) by n and f(n) by 2n+3

c1.n <=2n+3<=c2.n

if c1=1, c2=2, n=1

1*1 <=2*1+3 <=2*1

1 <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

If n=2

1*2<=2*2+3<=2*2

2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

Why we have three different asymptotic analysis?

As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm.

Suppose we have an array of n numbers, and we want to find the particular element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose, if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst case would be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.

So, three different analysis provide the proper bounding between the actual functions. Here, bounding means that we have upper as well as lower limit which assures that the algorithm will behave between these limits only, i.e., it will not go beyond these limits.

Common Asymptotic Notations

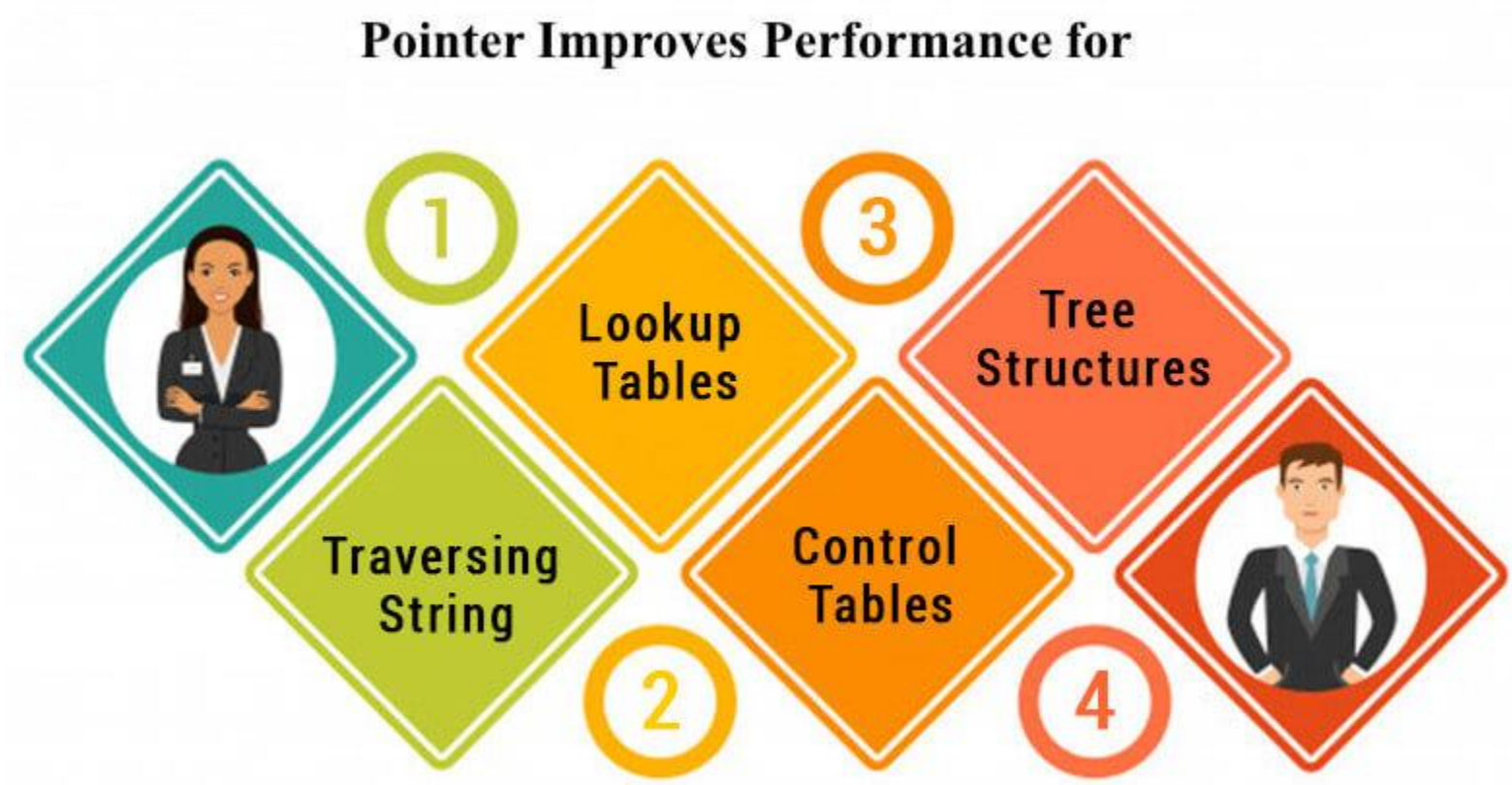
constant	-	?(1)
linear	-	?(n)
logarithmic	-	?(log n)
n log n	-	?(n log n)
exponential	-	2?(n)
cubic	-	?(n3)
polynomial	-	n?(1)
quadratic	-	?(n2)

Pointer

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

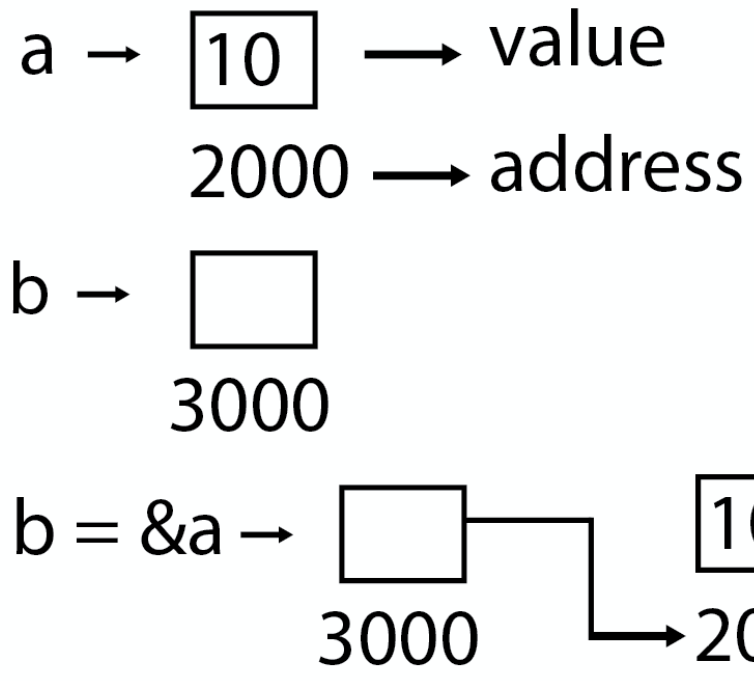
- Traversing String
- Lookup Tables
- Control Tables

- Tree Structures



Pointer Details

- **Pointer arithmetic:** There are four arithmetic operators that can be used in pointers: ++, --, +, -
- **Array of pointers:** You can define arrays to hold a number of pointers.
- **Pointer to pointer:** C allows you to have pointer on a pointer and so on.
- **Passing pointers to functions in C:** Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
- **Return pointer from functions in C:** C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.



Program

Pointer

```
1. #include <stdio.h>
2.
3. int main( )
4. {
```

```
5.  int a = 5;
6.  int *b;
7.  b = &a;
8.
9.  printf ("value of a = %d\n", a);
10. printf ("value of a = %d\n", *(&a));
11. printf ("value of a = %d\n", *b);
12. printf ("address of a = %u\n", &a);
13. printf ("address of a = %d\n", b);
14. printf ("address of b = %u\n", &b);
15. printf ("value of b = address of a = %u", b);
16. return 0;
17. }
```

Output

```
1. value of a = 5
2. value of a = 5
3. address of a = 3010494292
4. address of a = -1284473004
5. address of b = 3010494296
6. value of b = address of a = 3010494292
```

Program

Pointer to Pointer

```
1.  #include <stdio.h>
2.
3.  int main( )
4.  {
5.  int a = 5;
6.  int *b;
7.  int **c;
8.  b = &a;
9.  c = &b;
10. printf ("value of a = %d\n", a);
11. printf ("value of a = %d\n", *(&a));
12. printf ("value of a = %d\n", *b);
13. printf ("value of a = %d\n", **c);
14. printf ("value of b = address of a = %u\n", b);
15. printf ("value of c = address of b = %u\n", c);
16. printf ("address of a = %u\n", &a);
17. printf ("address of a = %u\n", b);
18. printf ("address of a = %u\n", *c);
19. printf ("address of b = %u\n", &b);
20. printf ("address of b = %u\n", c);
21. printf ("address of c = %u\n", &c);
22. return 0;
23. }
```

Pointer to Pointer

```
1. value of a = 5
2. value of a = 5
3. value of a = 5
4. value of a = 5
5. value of b = address of a = 2831685116
6. value of c = address of b = 2831685120
7. address of a = 2831685116
8. address of a = 2831685116
```

```
address of a = 2831685116
address of b = 2831685120
address of b = 2831685120
address of c = 2831685128
```

Structure

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

Syntax

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeber;
};
```

Advantages

- It can hold variables of different data types.
- We can create objects containing different types of attributes.
- It allows us to re-use the data layout across programs.
- It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

Program

```
#include<stdio.h>
#include<conio.h>
void main( )
{
struct employee
{
int id ;
float salary ;
int mobile ;
} ;
struct employee e1,e2,e3 ;
clrscr();
printf ("\nEnter ids, salary & mobile no. of 3 employee\n"
scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);
scanf ("%d%f %d", &e2.id, &e2.salary, &e2.mobile);
scanf ("%d %f %d", &e3.id, &e3.salary, &e3.mobile);
printf ("\n Entered Result ");
printf ("\n%d %f %d", e1.id, e1.salary, e1.mobile);
printf ("\n%d%f %d", e2.id, e2.salary, e2.mobile);
printf ("\n%d %f %d", e3.id, e3.salary, e3.mobile);
getch();
}
```

Array

Definition

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject. instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

for example, in C language, the syntax of declaring an array is like following:

```
1. int arr[10]; char arr[10]; float arr[5]
```

Need of using Array

In computer programming, the most of the cases requires to store the large number of data of similar type. To store such amount of data, we need to define a large number of variables. It would be very difficult to remember names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Following example illustrates, how array can be useful in writing code for a particular problem.

In the following example, we have marks of a student in six different subjects. The problem intends to calculate the average of all the marks of the student.

In order to illustrate the importance of array, we have created two programs, one is without using array and other involves the use of array to store marks.

Program without array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int marks_1 = 56, marks_2 = 78, marks_3 = 88, marks_4 = 76, marks_5 = 56, marks_6 = 89;
5.     float avg = (marks_1 + marks_2 + marks_3 + marks_4 + marks_5 +marks_6) / 6 ;
6.     printf(avg);
7. }
```

Program by using array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int marks[6] = {56,78,88,76,56,89);
5.     int i;
```

```
6.      float avg;
7.      for (i=0; i<6; i++ )
8.      {
9.          avg = avg + marks[i];
10.     }
11.     printf(avg);
12. }
```

Complexity of Array operations

Time and space complexity of various array operations are described in the following table.

Time Complexity

Algorithm	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

In array, space complexity for worst case is **O(n)**.

Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

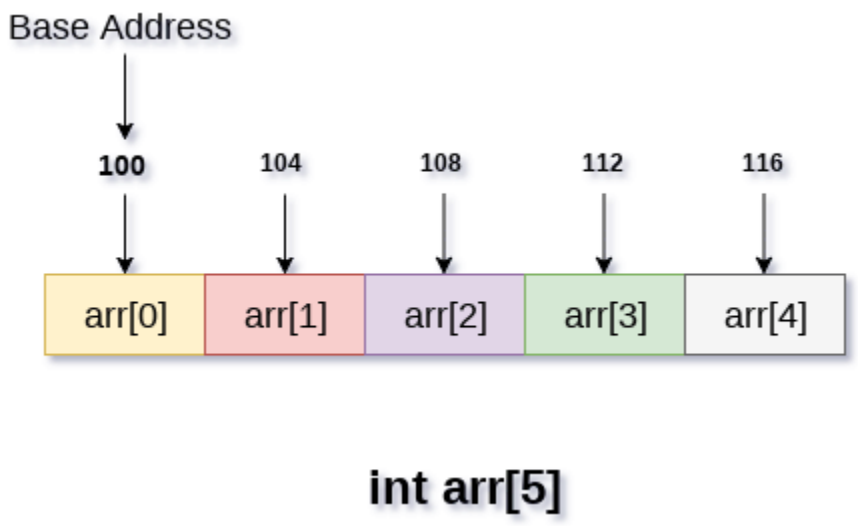
Memory Allocation of the array

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

The indexing of the array can be defined in three ways.

1. 0 (zero - based indexing) : The first element of the array will be arr[0].
2. 1 (one - based indexing) : The first element of the array will be arr[1].
3. n (n - based indexing) : The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is **n-1**. However, it will be n if we use **1** based indexing.

Accessing Elements of an array

To access any random element of an array we need the following information:

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Address of any element of a 1D array can be calculated by using the following formula:

1. Byte address of element A[i] = base address + size * (i - first index)

Example :

1. In an array, A[-10 ..... +2], Base address (BA) = 999, size of an element = 2 bytes,
2. find the location of A[-1].
3. L(A[-1]) = 999 + [(-1) - (-10)] x 2
4. = 999 + 18
5. = 1017

Passing array to the function :

As we have mentioned earlier that, the name of the array represents the starting address or the address of the first element of the array. All the elements of the array can be traversed by using the base address.

The following example illustrate, how the array can be passed to a function.

Example:

```
1. #include <stdio.h>
2. int summation(int[]);
3. void main ()
4. {
5.     int arr[5] = {0,1,2,3,4};
6.     int sum = summation(arr);
7.     printf("%d",sum);
8. }
9.
10. int summation (int arr[])
11. {
12.     int sum=0,i;
13.     for (i = 0; i<5; i++)
14.     {
15.         sum = sum + arr[i];
16.     }
```

```
17.     return sum;
18. }
```

The above program defines a function named as summation which accepts an array as an argument. The function calculates the sum of all the elements of the array and returns it.

2D Array

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

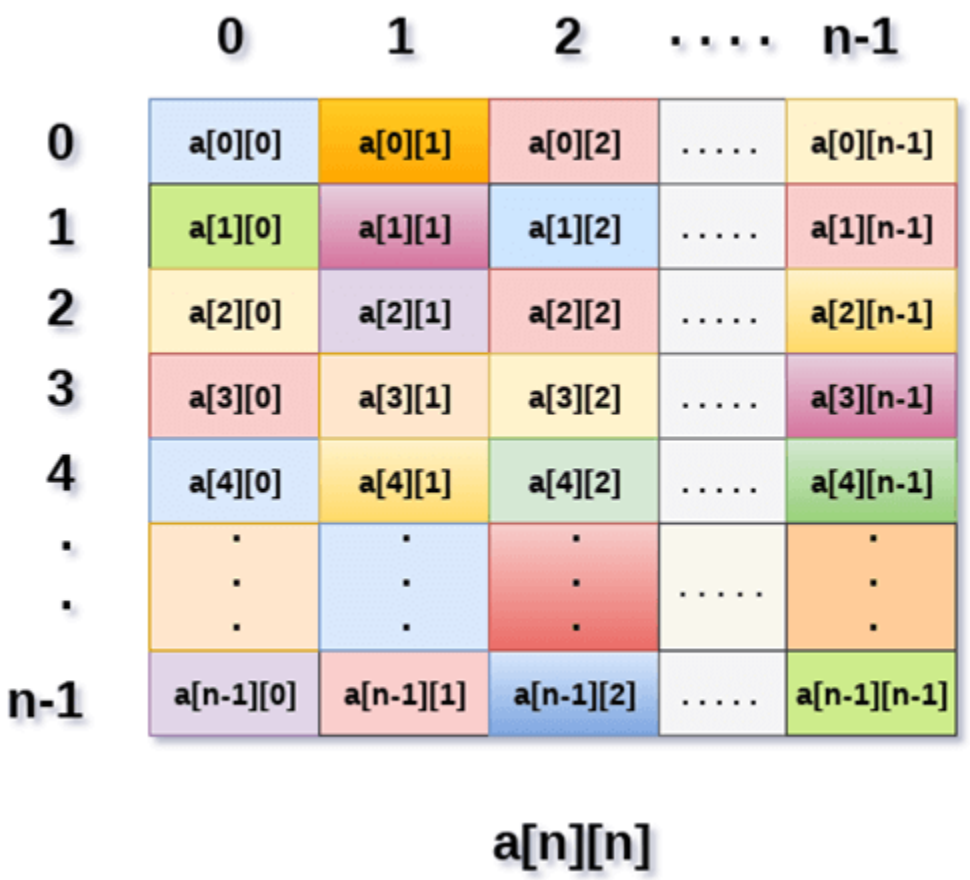
However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

How to declare 2D Array

The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

```
1.  int arr[max_rows][max_columns];
```

however, It produces the data structure which looks like following.



Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

How do we access data in a 2D array

Due to the fact that the elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

However, we can store the value stored in any particular cell of a 2D array to some variable x by using the following syntax.

```
1.  int x = a[i][j];
```

where i and j is the row and column number of the cell respectively.

We can assign each cell of a 2D array to 0 by using the following code:

```
1.  for ( int i=0; i<n ;i++)
2.  {
3.      for (int j=0; j<n; j++)
```

```
4.      {
5.          a[i][j] = 0;
6.      }
7.  }
```

Initializing 2D Arrays

We know that, when we declare and initialize one dimensional array in C programming simultaneously, we don't need to specify the size of the array. However this will not work with 2D arrays. We will have to define at least the second dimension of the array.

The syntax to declare and initialize the 2D array is given as follows.

```
1.  int arr[2][2] = {0,1,2,3};
```

The number of elements that can be present in a 2D array will always be equal to (**number of rows * number of columns**).

Example : Storing User's data into a 2D array and printing it.

C Example :

```
1.  #include <stdio.h>
2.  void main ()
3.  {
4.      int arr[3][3],i,j;
5.      for (i=0;i<3;i++)
6.      {
7.          for (j=0;j<3;j++)
8.          {
9.              printf("Enter a[%d][%d]: ",i,j);
10.             scanf("%d",&arr[i][j]);
11.         }
12.     }
13.     printf("\n printing the elements ....\n");
14.     for(i=0;i<3;i++)
15.     {
16.         printf("\n");
17.         for (j=0;j<3;j++)
18.         {
19.             printf("%d\t",arr[i][j]);
20.         }
21.     }
22. }
```

Java Example

```
1.  import java.util.Scanner;
2.  publicclass TwoDArray {
3.  publicstaticvoid main(String[] args) {
4.      int[][] arr = newint[3][3];
5.      Scanner sc = new Scanner(System.in);
6.      for (inti =0;i<3;i++)
7.      {
8.          for(intj=0;j<3;j++)
9.          {
10.             System.out.print("Enter Element");
11.              arr[i][j]=sc.nextInt();
12.             System.out.println();
13.         }
14.     }
15.     System.out.println("Printing Elements...");
```

```
16.     for(inti=0;i<3;i++)
17.     {
18.         System.out.println();
19.          for(intj=0;j<3;j++)
20.         {
21.             System.out.print(arr[i][j]+"\t");
22.         }
23.     }
24. }
25. }
```

C# Example

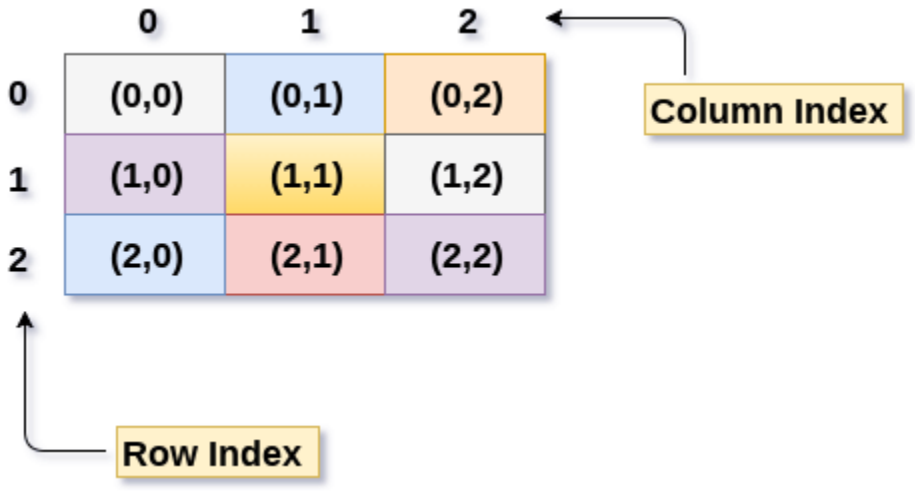
```
1.  using System;
2.
3.  public class Program
4.  {
5.      public static void Main()
6.      {
7.          int[,] arr = new int[3,3];
8.          for (int i=0;i<3;i++)
9.          {
10.             for (int j=0;j<3;j++)
11.             {
12.                 Console.WriteLine("Enter Element");
13.                  arr[i,j]= Convert.ToInt32(Console.ReadLine());
14.             }
15.         }
16.         Console.WriteLine("Printing Elements...");
17.         for (int i=0;i<3;i++)
18.         {
19.             Console.WriteLine();
20.             for (int j=0;j<3;j++)
21.             {
22.                 Console.Write(arr[i,j]+" ");
23.             }
24.         }
25.     }
26. }
```

Mapping 2D array to 1D array

When it comes to map a 2 dimensional array, most of us might think that why this mapping is required. However, 2 D arrays exists from the user point of view. 2D arrays are created to implement a relational database table lookalike data structure, in computer memory, the storage technique for 2D array is similar to that of an one dimensional array.

The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

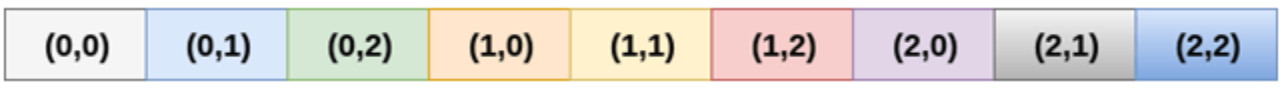
A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.



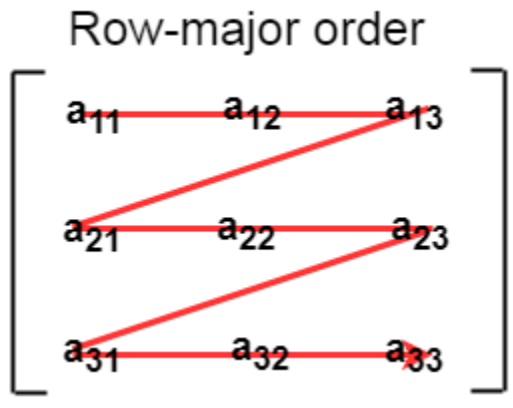
There are two main techniques of storing 2D array elements into memory

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

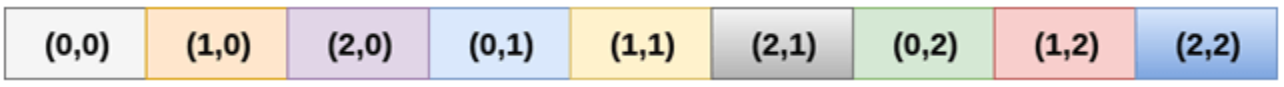


first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.

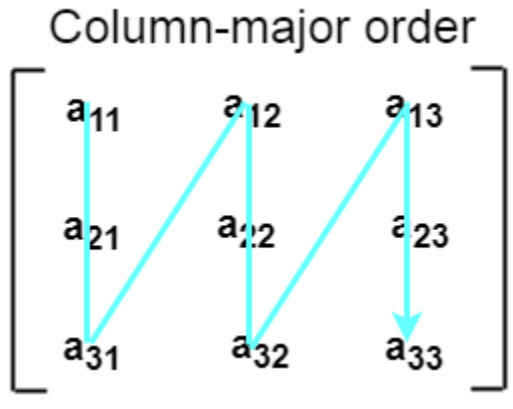


2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in in the above image is given as follows.



first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



Calculating the Address of the random element of a 2D array

Due to the fact that, there are two different techniques of storing the two dimensional array into the memory, there are two different formulas to calculate the address of a random element of the 2D array.

By Row Major Order

If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

1. Address(a[i][j]) = B. A. + (i * n + j) * size

where, B. A. is the base address or the address of the first element of the array a[0][0] .

Example :

1. a[10...30, 55...75], base address of the array (BA) = 0, size of an element = 4 bytes .
2. Find the location of a[15][68].
3.
4. Address(a[15][68]) = 0 +
5. ((15 - 10) x (68 - 55 + 1) + (68 - 55)) x 4
6.
7. = (5 x 14 + 13) x 4
8. = 83 x 4
9. = 332 answer

By Column major order

If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

1. Address(a[i][j]) = ((j*m)+i)*Size + BA

where BA is the base address of the array.

Example:

1. A [-5 ... +20][20 ... 70], BA = 1020, Size of element = 8 bytes. Find the location of a[0][30].
2.
3. Address [A[0][30]) = ((30-20) x 24 + 5) x 8 + 1020 = 245 x 8 + 1020 = 2980 bytes

Linked List

Before understanding the linked list concept, we first look at ***why there is a need for a linked list.***

If we want to store the value in a memory, we need a memory manager that manages the memory for every variable. For example, if we want to create a variable of integer type like:

```
1. int x;
```

In the above example, we have created a variable 'x' of type integer. As we know that integer variable occupies 4 bytes, so 'x' variable will occupy 4 bytes to store the value.

Suppose we want to create an array of integer type like:

```
1. int x[3];
```

In the above example, we have declared an array of size 3. As we know, that all the values of an array are stored in a continuous manner, so all the three values of an array are stored in a sequential fashion. The total memory space occupied by the array would be **3*4 = 12 bytes**.

There are two major drawbacks of using array:

- We cannot insert more than 3 elements in the above example because only 3 spaces are allocated for 3 elements.
- In the case of an array, lots of wastage of memory can occur. For example, if we declare an array of 50 size but we insert only 10 elements in an array. So, in this case, the memory space for other 40 elements will get wasted and cannot be used by another variable as this whole space is occupied by an array.

In array, we are providing the fixed-size at the compile-time, due to which wastage of memory occurs. The solution to this problem is to use the **linked list**.

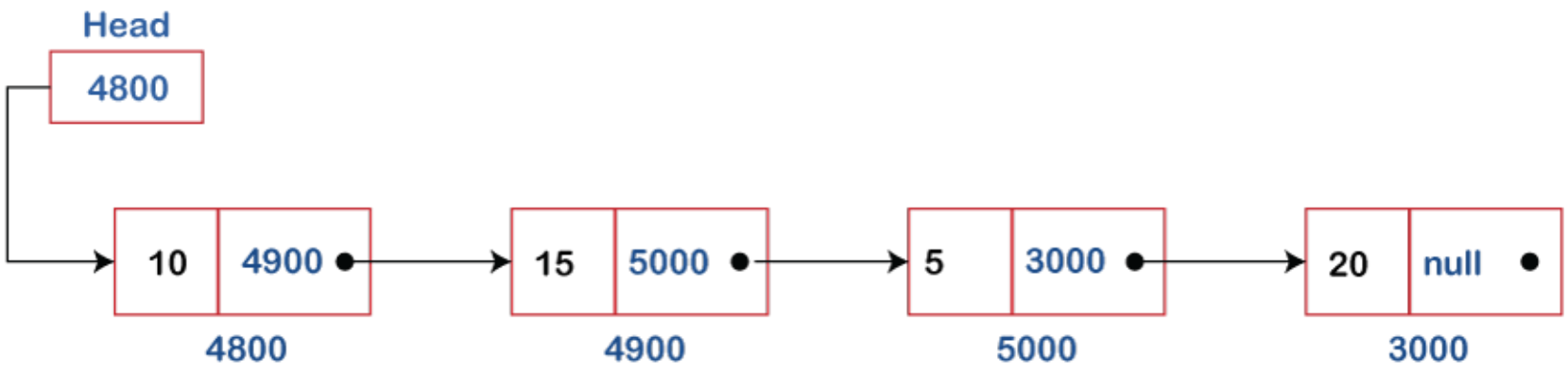
What is Linked List?

A linked list is also a collection of elements, but the elements are not stored in a consecutive location.

Suppose a programmer made a request for storing the integer value then size of 4-byte memory block is assigned to the integer value. The programmer made another request for storing 3 more integer elements; then, three different memory blocks are assigned to these three elements but the memory blocks are available in a random location. So, how are the elements connected?.

These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is **the data element**, and the other is the **pointer**. The pointer variable will occupy 4 bytes which is pointing to the next element.

A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:



In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the **NULL** value in the address part.

How can we declare the Linked list?

The declaration of an array is very simple as it is of single type. But the linked list contains two parts, which are of two different types, i.e., one is a simple variable, and the second one is a pointer variable. We can declare the linked list by using the user-defined data type known as structure.

The structure of a linked list can be defined as:

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. }
```

In the above declaration, we have defined a structure named as ***a node*** consisting of two variables: an integer variable (data), and the other one is the pointer (next), which contains the address of the next node.

Advantages of using a Linked list over Array

The following are the advantages of using a linked list over an array:

- **Dynamic data structure:**
  The size of the linked list is not fixed as it can vary according to our requirements.
- **Insertion and Deletion:**
  Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is O(1) in the linked list, while in the case of an array, the complexity would be O(n). If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.
- **Memory efficient**
  Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.
- **Implementation**
  Both the stacks and queues can be implemented using a linked list.

Disadvantages of Linked list

The following are the disadvantages of linked list:

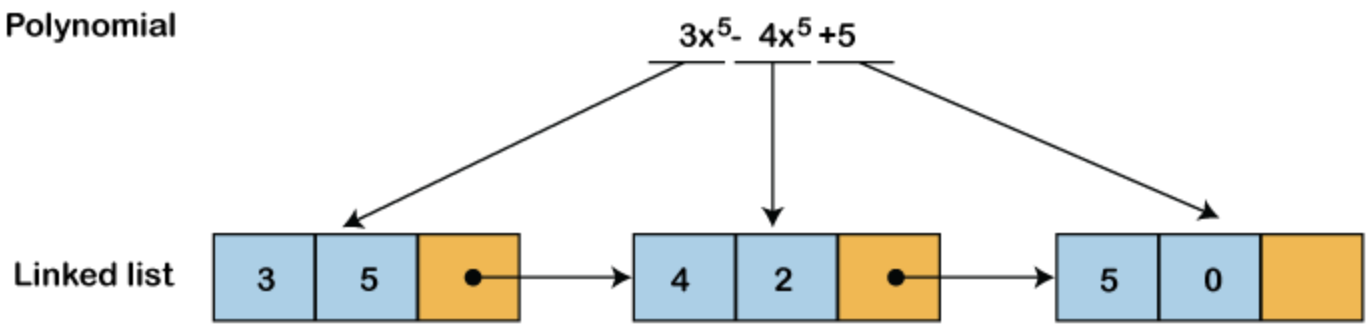
- **Memory usage**
  The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.
- **Traversal**
  In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing**
  In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked List

The applications of the linked list are given below:

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial. We know that polynomial is a collection of terms in which each term contains coefficient and power. The coefficients and power of each term are stored as node and link pointer points to the next element in a linked list, so

linked list can be used to create, delete and display the polynomial.



- A sparse matrix is used in scientific computation and numerical analysis. So, a linked list is used to represent the sparse matrix.
- The various operations like student's details, employee's details or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Stack, Queue, tree and various other data structures can be implemented using a linked list.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- To implement hashing, we require hash tables. The hash table contains entries that are implemented using linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Types of Linked List

Before knowing about the types of a linked list, we should know what is ***linked list***. So, to know about the linked list, click on the link given below:

Types of Linked list

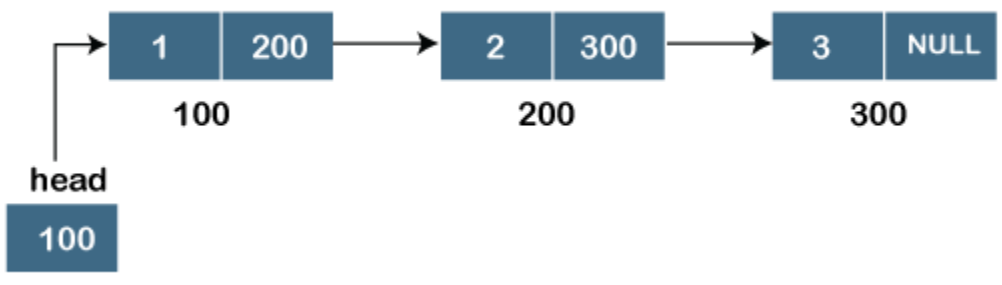
The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

Singly Linked list

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a *pointer*.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a ***head pointer***.

The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

1. struct node

```
2.  {
3.      int data;
4.      struct node *next;
5.  }
```

In the above representation, we have defined a user-defined structure named a *node* containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

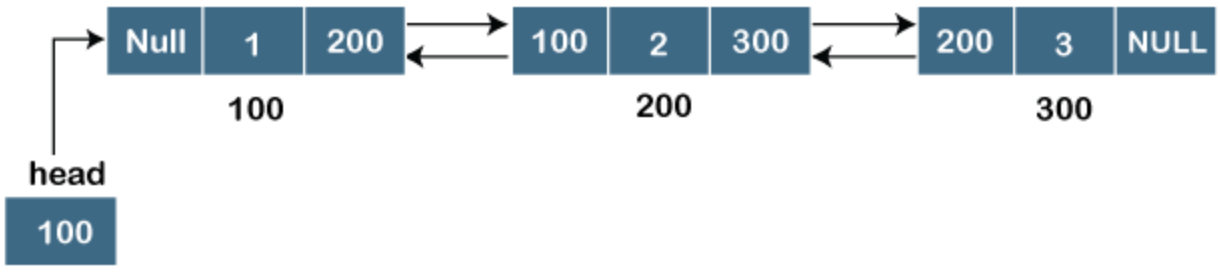
To know more about a singly linked list, click on the link given below:

https://www.javatpoint.com/singly-linked-list

Doubly linked list

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***. The initial node in the doubly linked list has the *NULL* value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

```
1.  struct node
2.  {
3.     int data;
4.     struct node *next;
5.    struct node *prev;
6.  }
```

In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is *data* of integer type, and the other two are the pointers, i.e., *next and prev* of the node type. The *next pointer* variable holds the address of the next node, and the *prev pointer* holds the address of the previous node. The type of both the pointers, i.e., *next and prev* is *struct node* as both the pointers are storing the address of the node of the ***struct node*** type.

To know more about doubly linked list, click on the link given below:

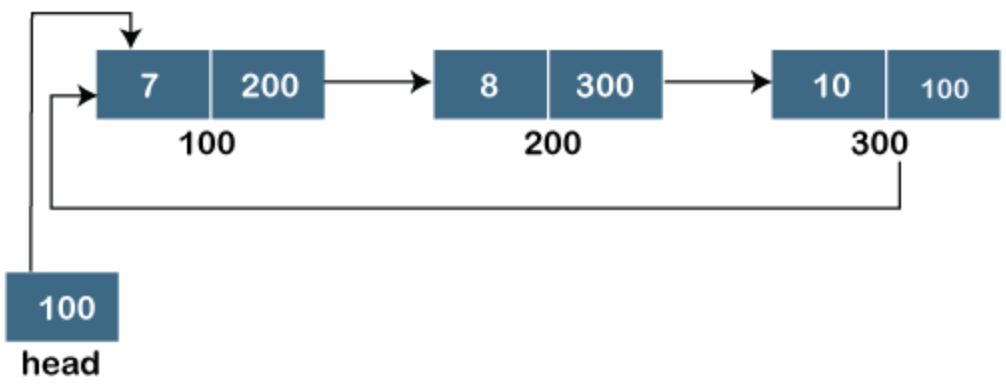
https://www.javatpoint.com/doubly-linked-list

Circular linked list

A circular linked list is a variation of a singly linked list. The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

```
1.  struct node
2.  {
3.     int data;
4.     struct node *next;
5.  }
```

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:

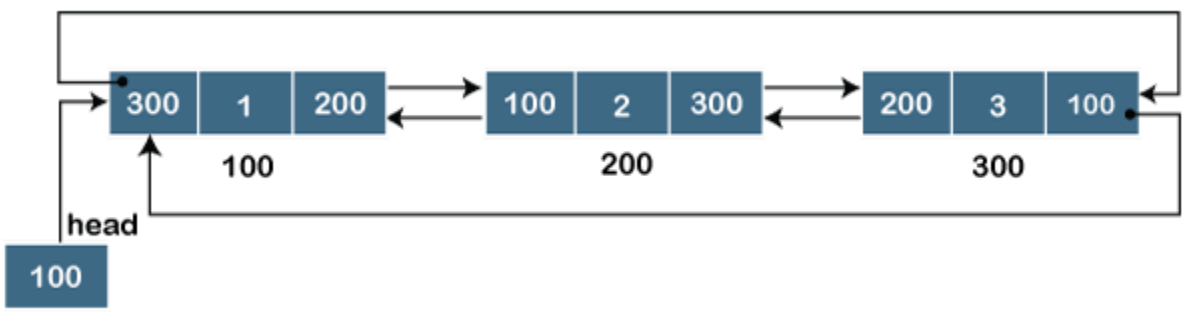


To know more about the circular linked list, click on the link given below:

https://www.javatpoint.com/circular-singly-linked-list

Doubly Circular linked list

The doubly circular linked list has the features of both the ***circular linked list*** and ***doubly linked list***.



The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

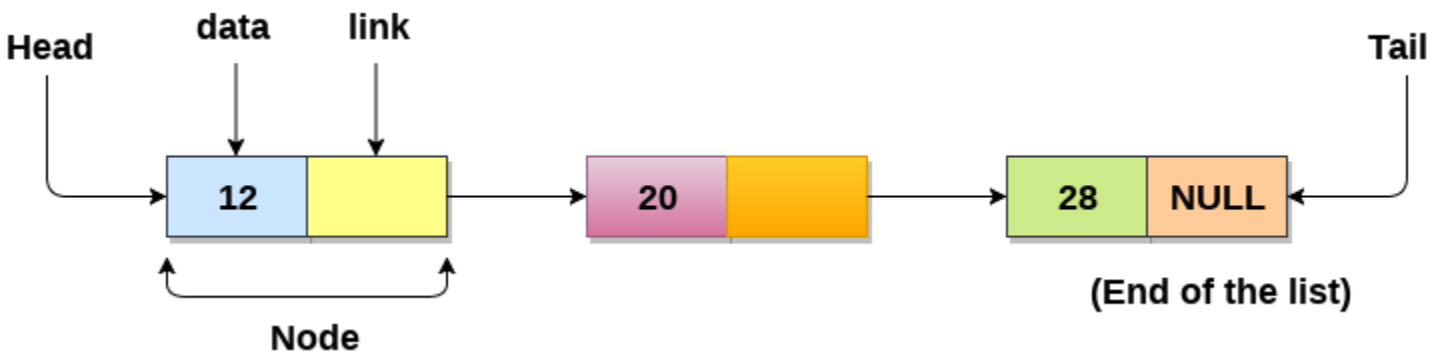
```
1. struct node
2. {
3.   int data;
4.   struct node *next;
5.  struct node *prev;
6. }
```

To know more about the doubly circular linked list, click on the link given below:

https://www.javatpoint.com/circular-doubly-linked-list

Linked List

- Linked List can be defined as collection of objects called *nodes* that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.

- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

Data Structure	Time Complexity								Space Compleity
	Average				*Worst*				*Worst*
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. };
```

```
6. struct node *head, *ptr;
7. ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Linked List in C: Menu Driven Program

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
8. struct node *head;
9.
```

```c
void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
    int choice =0;
    while(choice != 9)
    {
        printf("\n\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
        5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            randominsert();
            break;
            case 4:
            begin_delete();
            break;
            case 5:
            last_delete();
            break;
            case 6:
            random_delete();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr;
```

```
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
                    ptr -> next = NULL;
                    head = ptr;
                    printf("\nNode inserted");
                }
                else
                {
                    temp = head;
                    while (temp -> next != NULL)
                    {
                        temp = temp -> next;
                    }
                    temp->next = ptr;
                    ptr->next = NULL;
                    printf("\nNode inserted");

                }
            }
        }
        void randominsert()
        {
            int i,loc,item;
            struct node *ptr, *temp;
            ptr = (struct node *) malloc (sizeof(struct node));
            if(ptr == NULL)
```

```
124.            {
125.                printf("\nOVERFLOW");
126.            }
127.            else
128.            {
129.                printf("\nEnter element value");
130.                scanf("%d",&item);
131.                ptr->data = item;
132.                printf("\nEnter the location after which you want to insert ");
133.                scanf("\n%d",&loc);
134.                temp=head;
135.                for(i=0;i<loc;i++)
136.                {
137.                    temp = temp->next;
138.                    if(temp == NULL)
139.                    {
140.                        printf("\ncan't insert\n");
141.                        return;
142.                    }
143.
144.                }
145.                ptr ->next = temp ->next;
146.                temp ->next = ptr;
147.                printf("\nNode inserted");
148.            }
149.        }
150.        void begin_delete()
151.        {
152.            struct node *ptr;
153.            if(head == NULL)
154.            {
155.                printf("\nList is empty\n");
156.            }
157.            else
158.            {
159.                ptr = head;
160.                head = ptr->next;
161.                free(ptr);
162.                printf("\nNode deleted from the begining ...\n");
163.            }
164.        }
165.        void last_delete()
166.        {
167.            struct node *ptr,*ptr1;
168.            if(head == NULL)
169.            {
170.                printf("\nlist is empty");
171.            }
172.            else if(head -> next == NULL)
173.            {
174.                head = NULL;
175.                free(head);
176.                printf("\nOnly node of the list deleted ...\n");
177.            }
178.
179.            else
180.            {
```

```
            ptr = head;
            while(ptr->next != NULL)
            {
                ptr1 = ptr;
                ptr = ptr ->next;
            }
            ptr1->next = NULL;
            free(ptr);
            printf("\nDeleted Node from the last ...\n");
        }
    }
    void random_delete()
    {
        struct node *ptr,*ptr1;
        int loc,i;
        printf("\n Enter the location of the node after which you want to perform deletion \n");
        scanf("%d",&loc);
        ptr=head;
        for(i=0;i<loc;i++)
        {
            ptr1 = ptr;
            ptr = ptr->next;

            if(ptr == NULL)
            {
                printf("\nCan't delete");
                return;
            }
        }
        ptr1 ->next = ptr ->next;
        free(ptr);
        printf("\nDeleted node %d ",loc+1);
    }
    void search()
    {
        struct node *ptr;
        int item,i=0,flag;
        ptr = head;
        if(ptr == NULL)
        {
            printf("\nEmpty List\n");
        }
        else
        {
            printf("\nEnter item which you want to search?\n");
            scanf("%d",&item);
            while (ptr!=NULL)
            {
                if(ptr->data == item)
                {
                    printf("item found at location %d ",i+1);
                    flag=0;
                }
                else
                {
                    flag=1;
                }
```

```
                i++;
                ptr = ptr -> next;
            }
            if(flag==1)
            {
                printf("Item not found\n");
            }
        }

    }

    void display()
    {
        struct node *ptr;
        ptr = head;
        if(ptr == NULL)
        {
            printf("Nothing to print");
        }
        else
        {
            printf("\nprinting values . . . . .\n");
            while (ptr!=NULL)
            {
                printf("\n%d",ptr->data);
                ptr = ptr -> next;
            }
        }
    }
```

Output:

```
                    *********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
1

Enter value
```

```
1


Node inserted


*********Main Menu*********


Choose one option from the following list ...


==============================================


1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit


Enter your choice?

2


Enter value?

2


Node inserted


*********Main Menu*********


Choose one option from the following list ...


==============================================


1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit


Enter your choice?

3


Enter element value1
```

```
Enter the location after which you want to insert 1

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

8

printing values . . . . .

1

2

1

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

2
```

```
Enter value?
123

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
1

Enter value
1234

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
4
```

```
Node deleted from the begining ...

*********Main Menu*********

Choose one option from the following list ...

==========================================

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

5

Deleted Node from the last ...

*********Main Menu*********

Choose one option from the following list ...

==========================================

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

6

Enter the location of the node after which you want to perform deletion

1

Deleted node 2
```

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

8

printing values . . . . .

1

1

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

7

Enter item which you want to search?

1

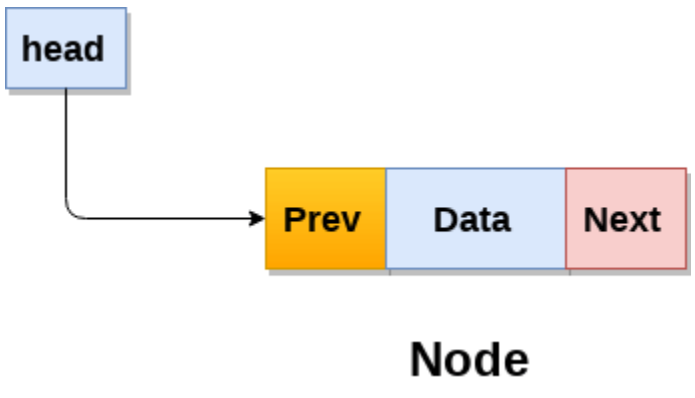
item found at location 1

item found at location 2

```
*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
9
```

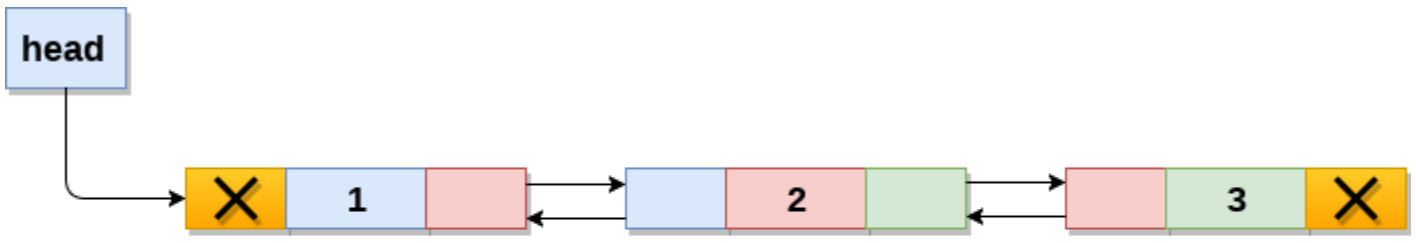
Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
1. struct node
2. {
3.     struct node *prev;
4.     int data;
5.     struct node *next;
6. }
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

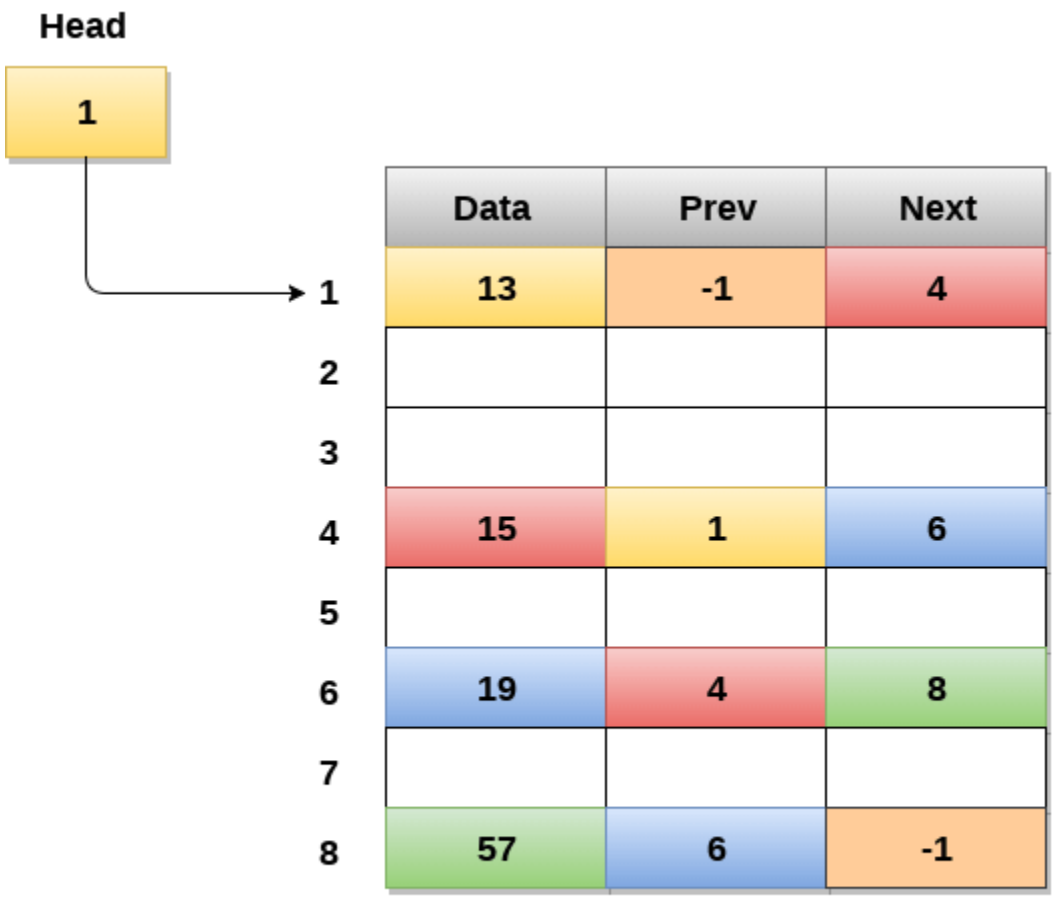
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

```
1. struct node
2. {
3.     struct node *prev;
4.     int data;
5.     struct node *next;
6. };
7. struct node *head;
```

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description

1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Menu Driven Program in C to implement all the operations of doubly linked list

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     struct node *prev;
6.     struct node *next;
7.     int data;
8. };
9. struct node *head;
10. void insertion_beginning();
11. void insertion_last();
12. void insertion_specified();
13. void deletion_beginning();
14. void deletion_last();
15. void deletion_specified();
16. void display();
17. void search();
18. void main ()
19. {
20. int choice =0;
21.   while(choice != 9)
22.   {
23.     printf("\n*********Main Menu*********\n");
24.     printf("\nChoose one option from the following list ...\n");
25.     printf("\n===============================================\n");
26.     printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
27.     5.Delete from last\n6.Delete the node after the given data\n7.Search\n8.Show\n9.Exit\n");
28.     printf("\nEnter your choice?\n");
29.     scanf("\n%d",&choice);
30.     switch(choice)
31.     {
32.       case 1:
33.         insertion_beginning();
34.       break;
```

```
35.          case 2:
36.                insertion_last();
37.          break;
38.         case 3:
39.          insertion_specified();
40.         break;
41.          case 4:
42.         deletion_beginning();
43.          break;
44.         case 5:
45.          deletion_last();
46.         break;
47.          case 6:
48.         deletion_specified();
49.          break;
50.         case 7:
51.          search();
52.         break;
53.           case 8:
54.           display();
55.           break;
56.           case 9:
57.           exit(0);
58.           break;
59.           default:
60.           printf("Please enter valid choice..");
61.        }
62.    }
63. }
64. void insertion_beginning()
65. {
66.    struct node *ptr;
67.    int item;
68.    ptr = (struct node *)malloc(sizeof(struct node));
69.    if(ptr == NULL)
70.    {
71.        printf("\nOVERFLOW");
72.    }
73.    else
74.    {
75.     printf("\nEnter Item value");
76.     scanf("%d",&item);
77.
78.    if(head==NULL)
79.    {
80.        ptr->next = NULL;
81.        ptr->prev=NULL;
82.        ptr->data=item;
83.        head=ptr;
84.    }
85.    else
86.    {
87.        ptr->data=item;
88.        ptr->prev=NULL;
89.        ptr->next = head;
90.        head->prev=ptr;
91.        head=ptr;
```

```
92.    }
93.    printf("\nNode inserted\n");
94. }
95.
96. }
97. void insertion_last()
98. {
99.    struct node *ptr,*temp;
100.            int item;
101.            ptr = (struct node *) malloc(sizeof(struct node));
102.            if(ptr == NULL)
103.            {
104.                printf("\nOVERFLOW");
105.            }
106.            else
107.            {
108.                printf("\nEnter value");
109.                scanf("%d",&item);
110.                ptr->data=item;
111.                if(head == NULL)
112.                {
113.                    ptr->next = NULL;
114.                    ptr->prev = NULL;
115.                    head = ptr;
116.                }
117.                else
118.                {
119.                    temp = head;
120.                    while(temp->next!=NULL)
121.                    {
122.                        temp = temp->next;
123.                    }
124.                    temp->next = ptr;
125.                    ptr ->prev=temp;
126.                    ptr->next = NULL;
127.                    }
128.
129.                }
130.            printf("\nnode inserted\n");
131.            }
132.        void insertion_specified()
133.        {
134.            struct node *ptr,*temp;
135.            int item,loc,i;
136.            ptr = (struct node *)malloc(sizeof(struct node));
137.            if(ptr == NULL)
138.            {
139.                printf("\n OVERFLOW");
140.            }
141.            else
142.            {
143.                temp=head;
144.                printf("Enter the location");
145.                scanf("%d",&loc);
146.                for(i=0;i<loc;i++)
147.                {
148.                    temp = temp->next;
```

```
149.                 if(temp == NULL)
150.                 {
151.                     printf("\n There are less than %d elements", loc);
152.                     return;
153.                 }
154.             }
155.             printf("Enter value");
156.             scanf("%d",&item);
157.             ptr->data = item;
158.             ptr->next = temp->next;
159.             ptr -> prev = temp;
160.             temp->next = ptr;
161.             temp->next->prev=ptr;
162.             printf("\nnode inserted\n");
163.         }
164.     }
165.     void deletion_beginning()
166.     {
167.         struct node *ptr;
168.         if(head == NULL)
169.         {
170.             printf("\n UNDERFLOW");
171.         }
172.         else if(head->next == NULL)
173.         {
174.             head = NULL;
175.             free(head);
176.             printf("\nnode deleted\n");
177.         }
178.         else
179.         {
180.             ptr = head;
181.             head = head -> next;
182.             head -> prev = NULL;
183.             free(ptr);
184.             printf("\nnode deleted\n");
185.         }
186.
187.     }
188.     void deletion_last()
189.     {
190.         struct node *ptr;
191.         if(head == NULL)
192.         {
193.             printf("\n UNDERFLOW");
194.         }
195.         else if(head->next == NULL)
196.         {
197.             head = NULL;
198.             free(head);
199.             printf("\nnode deleted\n");
200.         }
201.         else
202.         {
203.             ptr = head;
204.             if(ptr->next != NULL)
205.             {
```

```
206.                 ptr = ptr -> next;
207.              }
208.              ptr -> prev -> next = NULL;
209.              free(ptr);
210.              printf("\nnode deleted\n");
211.           }
212.        }
213.        void deletion_specified()
214.        {
215.           struct node *ptr, *temp;
216.           int val;
217.           printf("\n Enter the data after which the node is to be deleted : ");
218.           scanf("%d", &val);
219.           ptr = head;
220.           while(ptr -> data != val)
221.           ptr = ptr -> next;
222.           if(ptr -> next == NULL)
223.           {
224.              printf("\nCan't delete\n");
225.           }
226.           else if(ptr -> next -> next == NULL)
227.           {
228.              ptr ->next = NULL;
229.           }
230.           else
231.           {
232.              temp = ptr -> next;
233.              ptr -> next = temp -> next;
234.              temp -> next -> prev = ptr;
235.              free(temp);
236.              printf("\nnode deleted\n");
237.           }
238.        }
239.        void display()
240.        {
241.           struct node *ptr;
242.           printf("\n printing values...\n");
243.           ptr = head;
244.           while(ptr != NULL)
245.           {
246.              printf("%d\n",ptr->data);
247.              ptr=ptr->next;
248.           }
249.        }
250.        void search()
251.        {
252.           struct node *ptr;
253.           int item,i=0,flag;
254.           ptr = head;
255.           if(ptr == NULL)
256.           {
257.              printf("\nEmpty List\n");
258.           }
259.           else
260.           {
261.              printf("\nEnter item which you want to search?\n");
262.              scanf("%d",&item);
```

```
263.             while (ptr!=NULL)
264.             {
265.                 if(ptr->data == item)
266.                 {
267.                     printf("\nitem found at location %d ",i+1);
268.                    flag=0;
269.                     break;
270.                 }
271.                 else
272.                 {
273.                     flag=1;
274.                 }
275.                 i++;
276.                 ptr = ptr -> next;
277.             }
278.             if(flag==1)
279.             {
280.                 printf("\nItem not found\n");
281.             }
282.         }
283.
284.     }
```

Output

```
*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
1

Enter Item value12

Node inserted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
```

```
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
1

Enter Item value123

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
1

Enter Item value1234

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...
1234
123
12

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
2

Enter value89

node inserted

*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
```

```
8.Show
9.Exit

Enter your choice?
3
Enter the location1
Enter value12345

node inserted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...
1234
123
12345
12
89

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
4

node deleted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
5

node deleted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
```

```
8

 printing values...
123
12345

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
6

 Enter the data after which the node is to be deleted : 123

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
8

 printing values...
123

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
7

Enter item which you want to search?
123

item found at location 1
*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
6

 Enter the data after which the node is to be deleted : 123

Can't delete
```

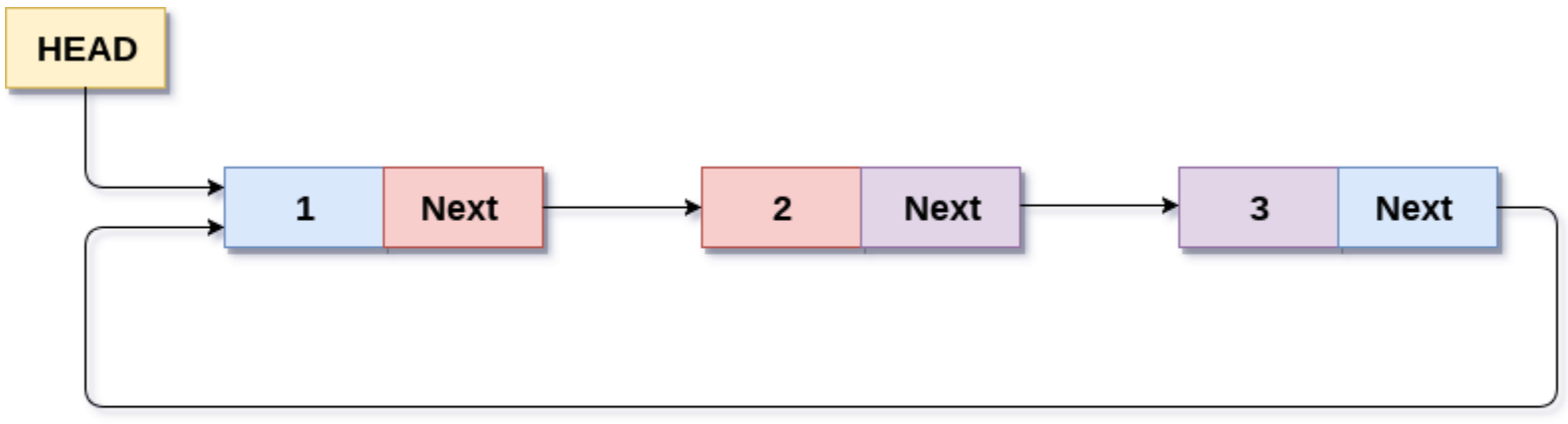
```
*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
9

Exited..
```

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



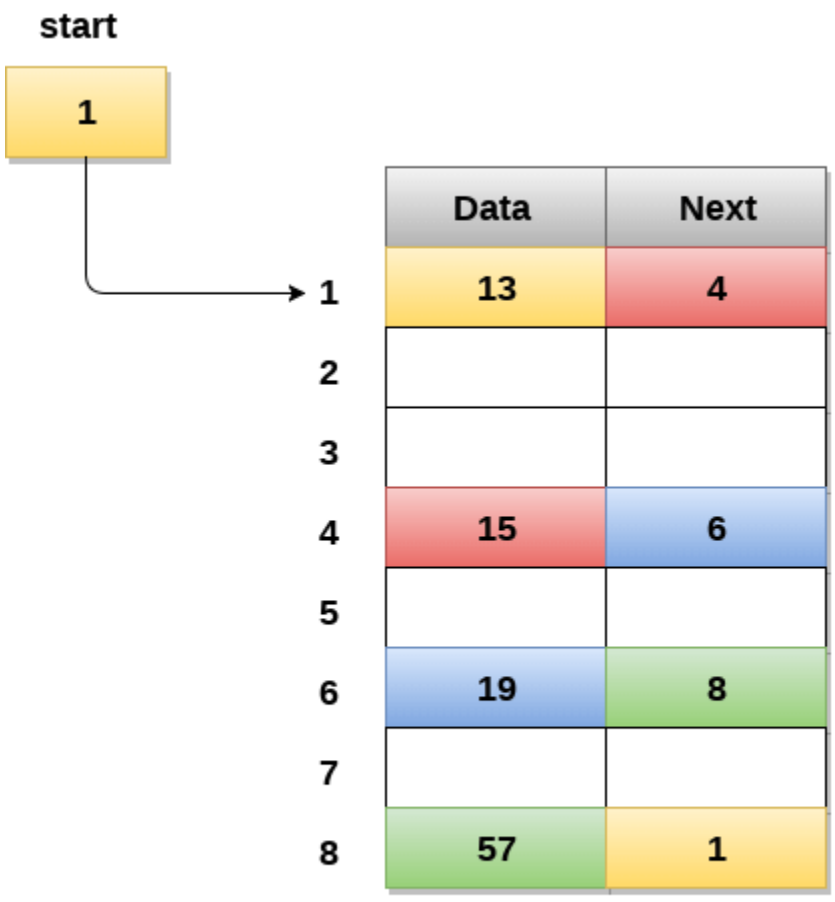
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Menu-driven program in C implementing all operations

on circular singly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 7)
   {
       printf("\n*********Main Menu*********\n");
       printf("\nChoose one option from the following list ...\n");
       printf("\n==============================================\n");
       printf("\n1.Insert in begining\n2.Insert at last\n3.Delete from Beginning\n4.Delete from last\n5.Search for an element\n6.
Show\n7.Exit\n");
       printf("\nEnter your choice?\n");
       scanf("\n%d",&choice);
       switch(choice)
       {
           case 1:
           beginsert();
           break;
           case 2:
           lastinsert();
           break;
           case 3:
           begin_delete();
           break;
           case 4:
           last_delete();
           break;
           case 5:
           search();
           break;
           case 6:
           display();
           break;
           case 7:
           exit(0);
           break;
           default:
```

```
53.             printf("Please enter valid choice..");
54.         }
55.     }
56. }
57. void beginsert()
58. {
59.     struct node *ptr,*temp;
60.     int item;
61.     ptr = (struct node *)malloc(sizeof(struct node));
62.     if(ptr == NULL)
63.     {
64.         printf("\nOVERFLOW");
65.     }
66.     else
67.     {
68.         printf("\nEnter the node data?");
69.          scanf("%d",&item);
70.         ptr -> data = item;
71.          if(head == NULL)
72.         {
73.              head = ptr;
74.             ptr -> next = head;
75.         }
76.         else
77.         {
78.             temp = head;
79.              while(temp->next != head)
80.                 temp = temp->next;
81.              ptr->next = head;
82.             temp -> next = ptr;
83.              head = ptr;
84.         }
85.         printf("\nnode inserted\n");
86.     }
87.
88. }
89. void lastinsert()
90. {
91.     struct node *ptr,*temp;
92.     int item;
93.     ptr = (struct node *)malloc(sizeof(struct node));
94.     if(ptr == NULL)
95.     {
96.         printf("\nOVERFLOW\n");
97.     }
98.     else
99.     {
100.                 printf("\nEnter Data?");
101.                  scanf("%d",&item);
102.                 ptr->data = item;
103.                  if(head == NULL)
104.                 {
105.                      head = ptr;
106.                     ptr -> next = head;
107.                 }
108.                 else
109.                 {
```

```
                temp = head;
                 while(temp -> next != head)
                {
                    temp = temp -> next;
                }
                 temp -> next = ptr;
                ptr -> next = head;
              }

              printf("\nnode inserted\n");
           }

        }

        void begin_delete()
        {
           struct node *ptr;
           if(head == NULL)
           {
               printf("\nUNDERFLOW");
           }
           else if(head->next == head)
           {
               head = NULL;
              free(head);
               printf("\nnode deleted\n");
           }

           else
           {   ptr = head;
              while(ptr -> next != head)
                  ptr = ptr -> next;
              ptr->next = head->next;
               free(head);
              head = ptr->next;
               printf("\nnode deleted\n");

           }
        }
        void last_delete()
        {
           struct node *ptr, *preptr;
           if(head==NULL)
           {
              printf("\nUNDERFLOW");
           }
           else if (head ->next == head)
           {
              head = NULL;
               free(head);
              printf("\nnode deleted\n");

           }
           else
           {
               ptr = head;
              while(ptr ->next != head)
```

```
167.                {
168.                   preptr=ptr;
169.                    ptr = ptr->next;
170.                }
171.                preptr->next = ptr -> next;
172.                free(ptr);
173.                printf("\nnode deleted\n");
174.
175.            }
176.        }
177.
178.        void search()
179.        {
180.            struct node *ptr;
181.            int item,i=0,flag=1;
182.            ptr = head;
183.            if(ptr == NULL)
184.            {
185.                printf("\nEmpty List\n");
186.            }
187.            else
188.            {
189.                printf("\nEnter item which you want to search?\n");
190.                scanf("%d",&item);
191.                if(head ->data == item)
192.                {
193.                printf("item found at location %d",i+1);
194.                flag=0;
195.                }
196.                else
197.                {
198.                while (ptr->next != head)
199.                {
200.                   if(ptr->data == item)
201.                    {
202.                       printf("item found at location %d ",i+1);
203.                        flag=0;
204.                       break;
205.                    }
206.                   else
207.                    {
208.                       flag=1;
209.                    }
210.                   i++;
211.                    ptr = ptr -> next;
212.                }
213.                }
214.                if(flag != 0)
215.                {
216.                   printf("Item not found\n");
217.                }
218.            }
219.
220.        }
221.
222.        void display()
223.        {
```

```
224.            struct node *ptr;
225.            ptr=head;
226.            if(head == NULL)
227.            {
228.               printf("\nnothing to print");
229.            }
230.            else
231.            {
232.               printf("\n printing values ... \n");
233.
234.               while(ptr -> next != head)
235.               {
236.
237.                  printf("%d\n", ptr -> data);
238.                  ptr = ptr -> next;
239.               }
240.               printf("%d\n", ptr -> data);
241.            }
242.
243.         }
```

Output:

```
*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
1

Enter the node data?10

node inserted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
2

Enter Data?20

node inserted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
2
```

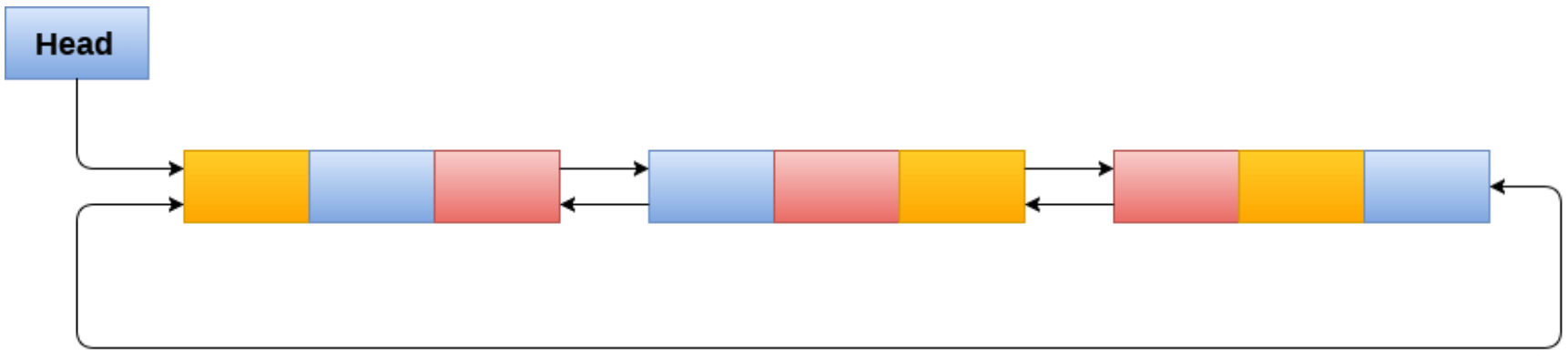
```
Enter Data?30

node inserted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
3

node deleted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
4

node deleted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
5

Enter item which you want to search?
20
item found at location 1
*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
6

 printing values ...
20

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
```

```
7.Exit

Enter your choice?
7
```

Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.

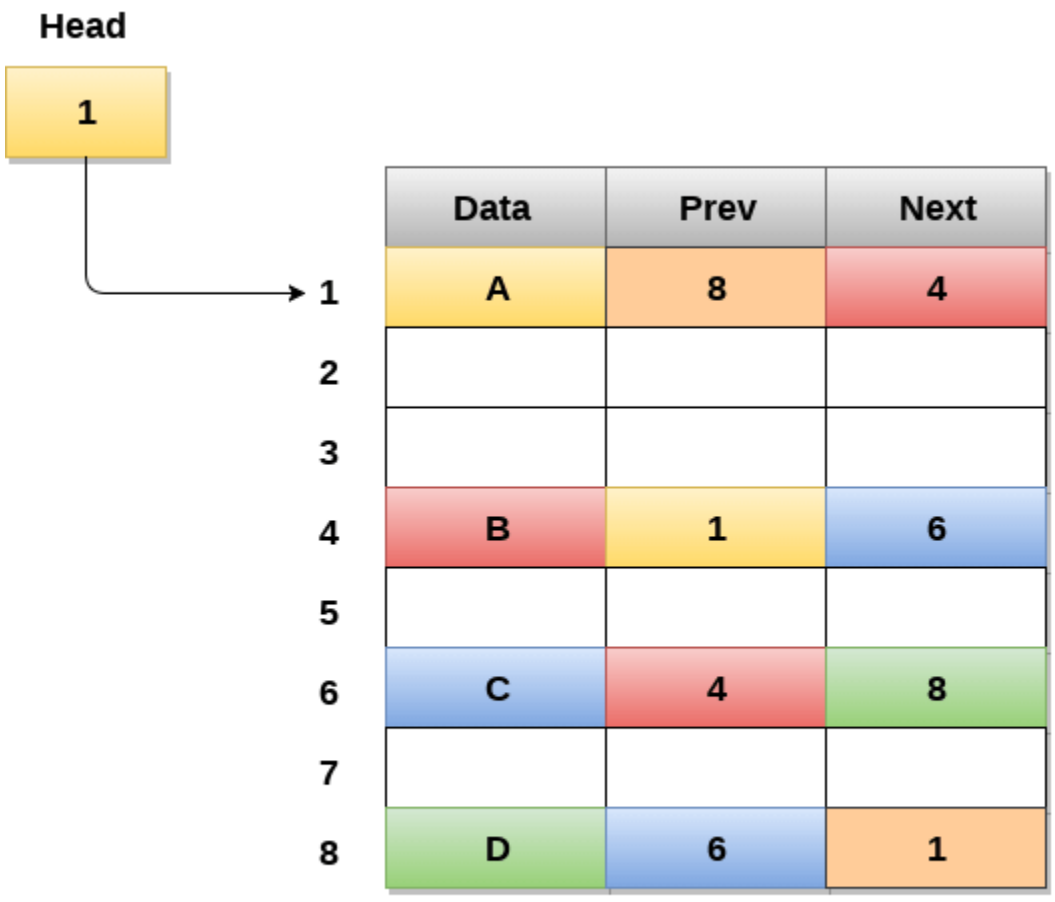


Circular Doubly Linked List

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Memory Management of Circular Doubly linked list

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1. In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.



Memory Representation of a Circular Doubly linked list

Operations on circular doubly linked list :

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list is described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

C program to implement all the operations on circular doubly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *prev;
    struct node *next;
    int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void deletion_beginning();
void deletion_last();
```

```
14. void display();
15. void search();
16. void main ()
17. {
18. int choice =0;
19.    while(choice != 9)
20.    {
21.        printf("\n*********Main Menu*********\n");
22.       printf("\nChoose one option from the following list ...\n");
23.        printf("\n==============================================\n");
24.       printf("\n1.Insert in Beginning\n2.Insert at last\n3.Delete from Beginning\n4.Delete from last\n5.Search\n6.Show\n7.Exit\
   n");
25.        printf("\nEnter your choice?\n");
26.       scanf("\n%d",&choice);
27.        switch(choice)
28.       {
29.            case 1:
30.           insertion_beginning();
31.            break;
32.          case 2:
33.                  insertion_last();
34.          break;
35.           case 3:
36.          deletion_beginning();
37.           break;
38.          case 4:
39.           deletion_last();
40.          break;
41.           case 5:
42.          search();
43.           break;
44.          case 6:
45.           display();
46.          break;
47.           case 7:
48.          exit(0);
49.           break;
50.          default:
51.           printf("Please enter valid choice..");
52.       }
53.    }
54. }
55. void insertion_beginning()
56. {
57.   struct node *ptr,*temp;
58.   int item;
59.   ptr = (struct node *)malloc(sizeof(struct node));
60.   if(ptr == NULL)
61.   {
62.       printf("\nOVERFLOW");
63.   }
64.   else
65.   {
66.    printf("\nEnter Item value");
67.    scanf("%d",&item);
68.    ptr->data=item;
69.   if(head==NULL)
```

```
70.  {
71.     head = ptr;
72.     ptr -> next = head;
73.     ptr -> prev = head;
74.  }
75.  else
76.  {
77.      temp = head;
78.   while(temp -> next != head)
79.   {
80.       temp = temp -> next;
81.   }
82.   temp -> next = ptr;
83.   ptr -> prev = temp;
84.   head -> prev = ptr;
85.   ptr -> next = head;
86.   head = ptr;
87.  }
88.   printf("\nNode inserted\n");
89. }
90.
91. }
92. void insertion_last()
93. {
94.   struct node *ptr,*temp;
95.   int item;
96.   ptr = (struct node *) malloc(sizeof(struct node));
97.   if(ptr == NULL)
98.   {
99.       printf("\nOVERFLOW");
100.          }
101.          else
102.          {
103.              printf("\nEnter value");
104.              scanf("%d",&item);
105.               ptr->data=item;
106.              if(head == NULL)
107.              {
108.                 head = ptr;
109.                  ptr -> next = head;
110.                  ptr -> prev = head;
111.              }
112.              else
113.              {
114.                temp = head;
115.                 while(temp->next !=head)
116.                {
117.                    temp = temp->next;
118.                }
119.                 temp->next = ptr;
120.                ptr ->prev=temp;
121.                 head -> prev = ptr;
122.             ptr -> next = head;
123.               }
124.          }
125.            printf("\nnode inserted\n");
126.       }
```

```
127.
128.        void deletion_beginning()
129.        {
130.            struct node *temp;
131.            if(head == NULL)
132.            {
133.                printf("\n UNDERFLOW");
134.            }
135.            else if(head->next == head)
136.            {
137.                head = NULL;
138.               free(head);
139.                printf("\nnode deleted\n");
140.            }
141.            else
142.            {
143.                temp = head;
144.               while(temp -> next != head)
145.                {
146.                   temp = temp -> next;
147.                }
148.               temp -> next = head -> next;
149.                head -> next -> prev = temp;
150.               free(head);
151.                head = temp -> next;
152.            }
153.
154.        }
155.        void deletion_last()
156.        {
157.            struct node *ptr;
158.            if(head == NULL)
159.            {
160.                printf("\n UNDERFLOW");
161.            }
162.            else if(head->next == head)
163.            {
164.                head = NULL;
165.                free(head);
166.                printf("\nnode deleted\n");
167.            }
168.            else
169.            {
170.                ptr = head;
171.                if(ptr->next != head)
172.                {
173.                    ptr = ptr -> next;
174.                }
175.                ptr -> prev -> next = head;
176.                head -> prev = ptr -> prev;
177.                free(ptr);
178.                printf("\nnode deleted\n");
179.            }
180.        }
181.
182.        void display()
183.        {
```

```
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }

}

void search()
{
    struct node *ptr;
    int item,i=0,flag=1;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        if(head ->data == item)
        {
        printf("item found at location %d",i+1);
        flag=0;
        }
        else
    {
     while (ptr->next != head)
    {
        if(ptr->data == item)
        {
            printf("item found at location %d ",i+1);
            flag=0;
            break;
        }
        else
        {
            flag=1;
        }
        i++;
        ptr = ptr -> next;
    }
    }
```

```
241.                if(flag != 0)
242.                {
243.                    printf("Item not found\n");
244.                }
245.            }
246.
247.        }
```

Output:

```
*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
1

Enter Item value123

Node inserted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
2

Enter value234

node inserted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
1

Enter Item value90

Node inserted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
2
```

```
Enter value80

node inserted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
3

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
4

node deleted

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
6

 printing values ...
123

*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit

Enter your choice?
5

Enter item which you want to search?
123
item found at location 1
*********Main Menu*********

Choose one option from the following list ...

================================================

1.Insert in Beginning
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search
6.Show
7.Exit
```

```
Enter your choice?
7
```

Skip list in Data structure

What is a skip list?

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

Complexity table of the Skip list

S. No	Complexity	Average case	Worst case
1.	Access complexity	O(logn)	O(n)
2.	Search complexity	O(logn)	O(n)
3.	Delete complexity	O(logn)	O(n)
4.	Insert complexity	O(logn)	O(n)
5.	Space complexity	-	O(nlogn)

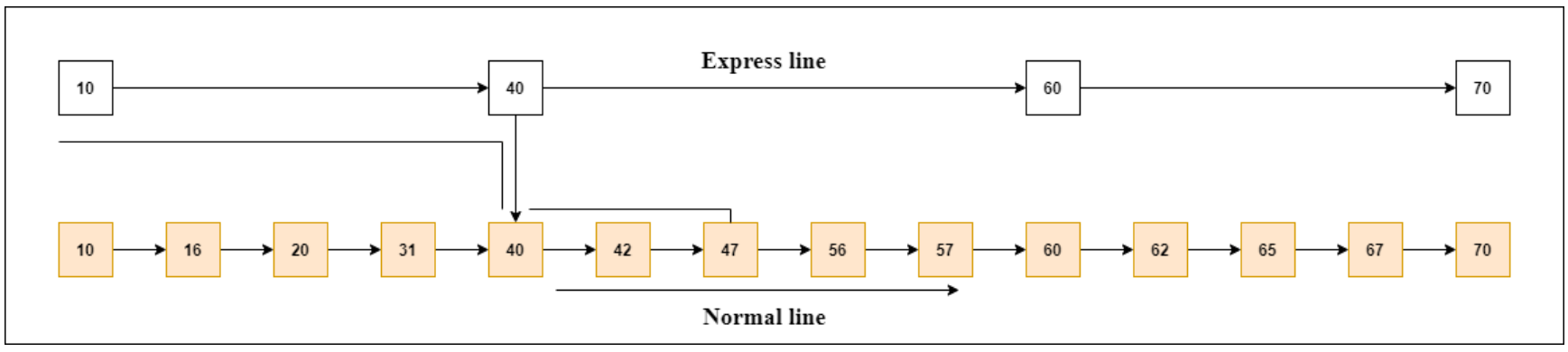
Working of the Skip list

Let's take an example to understand the working of the skip list. In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.

The lower layer is a common line that links all nodes, and the top layer is an express line that links only the main nodes, as you can see in the diagram.

Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal a 47 or more than 47.

You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.



Note: Once you find a node like this on the "express line", you go from this node to a "normal lane" using a pointer, and when you search for the node in the normal line.

Skip List Basic Operations

There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.

- **Search Operation:** The search operation is used to search a particular node in a skip list.

Algorithm of the insertion operation

```
1. Insertion (L, Key)
2. local update [0...Max_Level + 1]
3. a = L → header
4. for i = L → level down to 0 do.
5.         while a → forward[i] → key  forward[i]
6. update[i] = a
7.
8. a = a → forward[0]
9. lvl = random_Level()
10. if lvl > L → level then
11. for i = L → level + 1 to lvl do
12.       update[i] = L → header
13.       L → level = lvl
14.
15. a = makeNode(lvl, Key, value)
16. for i = 0 to level do
17.       a → forward[i] = update[i] → forward[i]
18. update[i] → forward[i] = a
```

Algorithm of deletion operation

```
1. Deletion (L, Key)
2. local update [0... Max_Level + 1]
3. a = L → header
4. for i = L → level down to 0 do.
5.    while a → forward[i] → key forward[i]
6.       update[i] = a
7. a = a → forward[0]
8. if a → key = Key then
9.    for i = 0 to L → level do
10.       if update[i] → forward[i] ? a then break
11.       update[i] → forward[i] = a → forward[i]
12.    free(a)
13.    while L → level > 0 and L → header → forward[L → level] = NIL do
14.       L → level = L → level - 1
```

Algorithm of searching operation

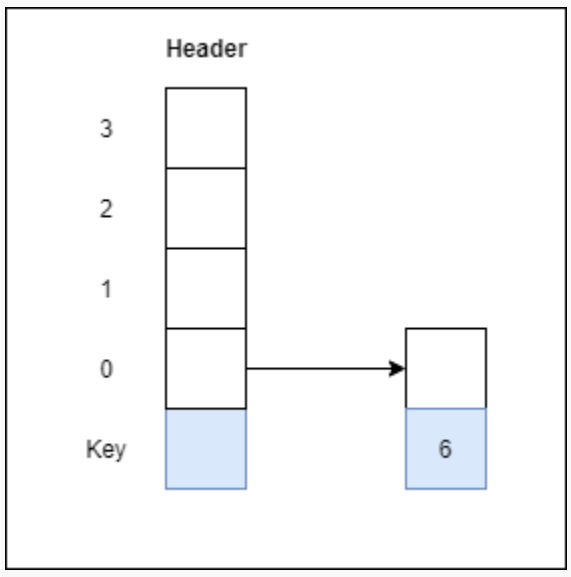
```
1. Searching (L, SKey)
2.         a = L → header
3.         loop invariant: a → key level down to 0 do.
4.             while a → forward[i] → key forward[i]
5.         a = a → forward[0]
6.        if a → key = SKey then return a → value
7.         else return failure
```

Example 1: Create a skip list, we want to insert these following keys in the empty skip list.

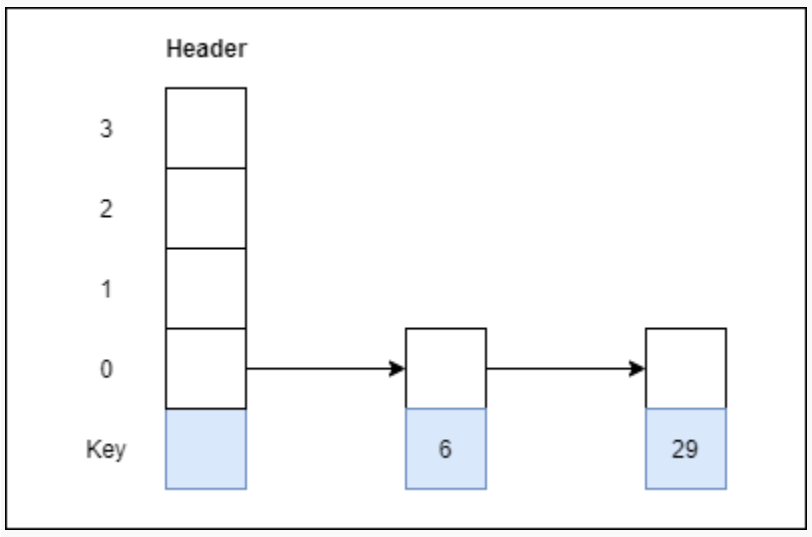
1. 6 with level 1.
2. 29 with level 1.
3. 22 with level 4.
4. 9 with level 3.
5. 17 with level 1.
6. 4 with level 2.

Ans:

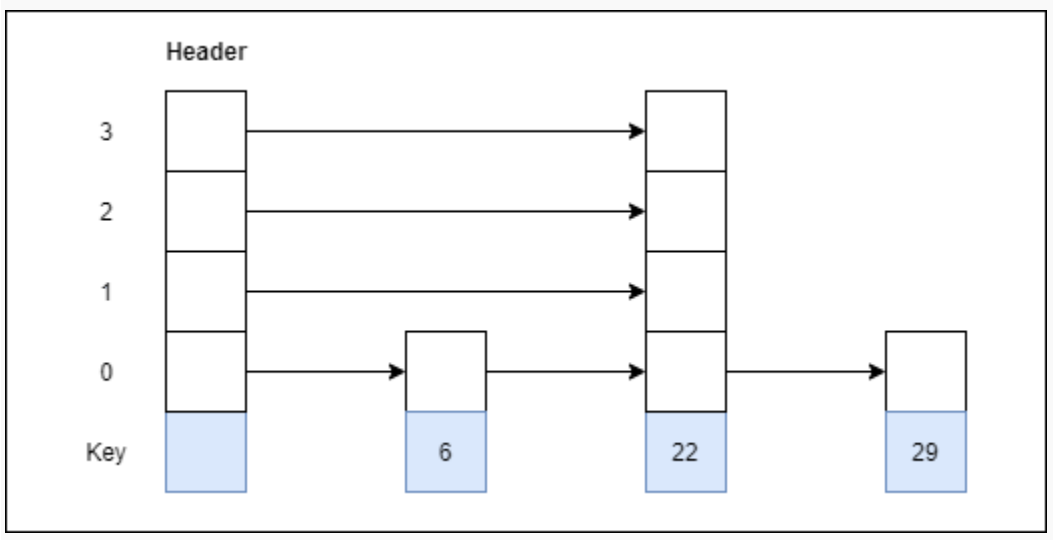
Step 1: Insert 6 with level 1



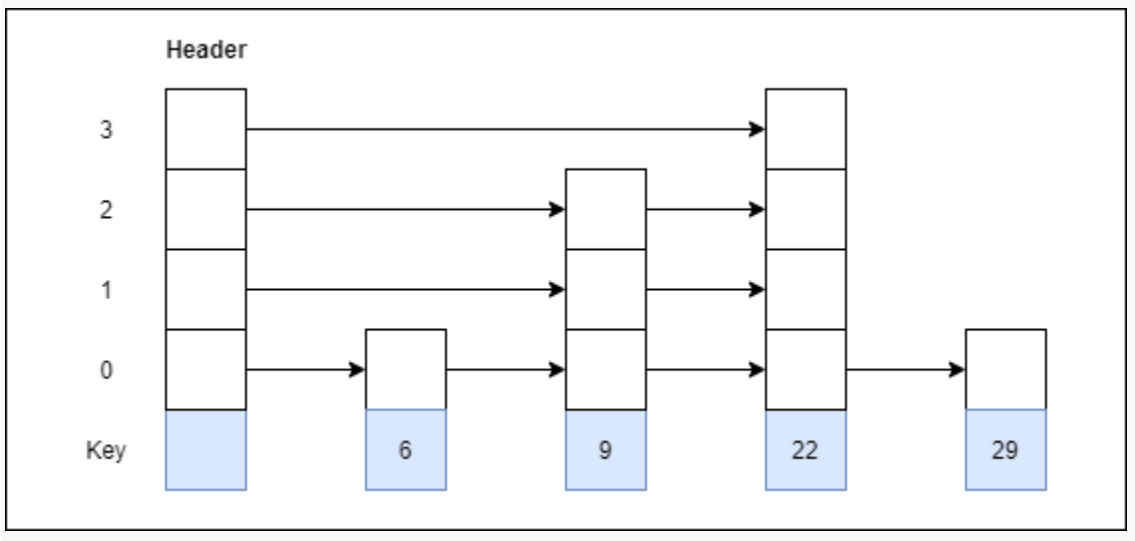
Step 2: Insert 29 with level 1



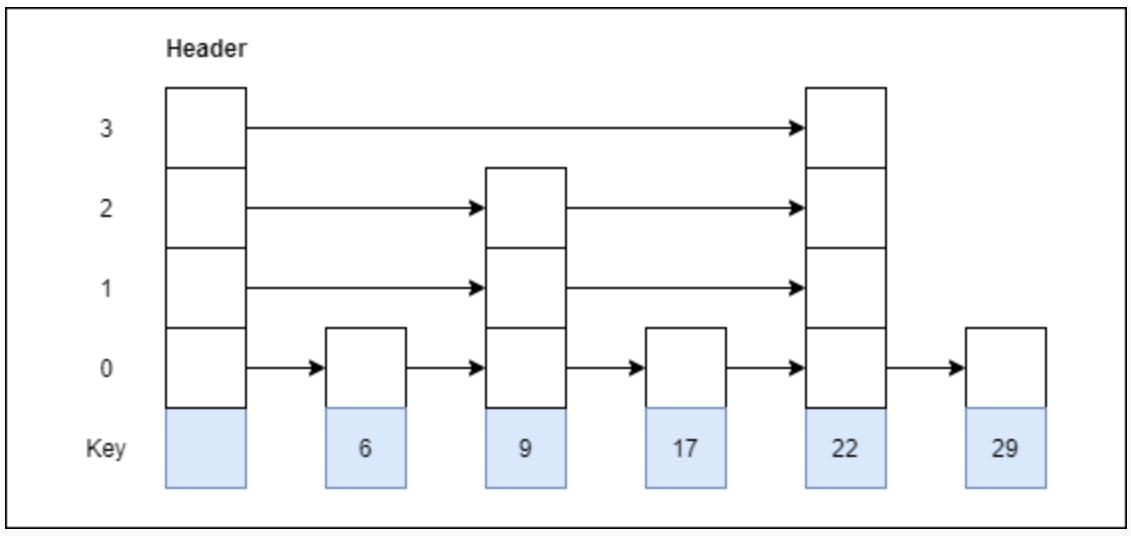
Step 3: Insert 22 with level 4



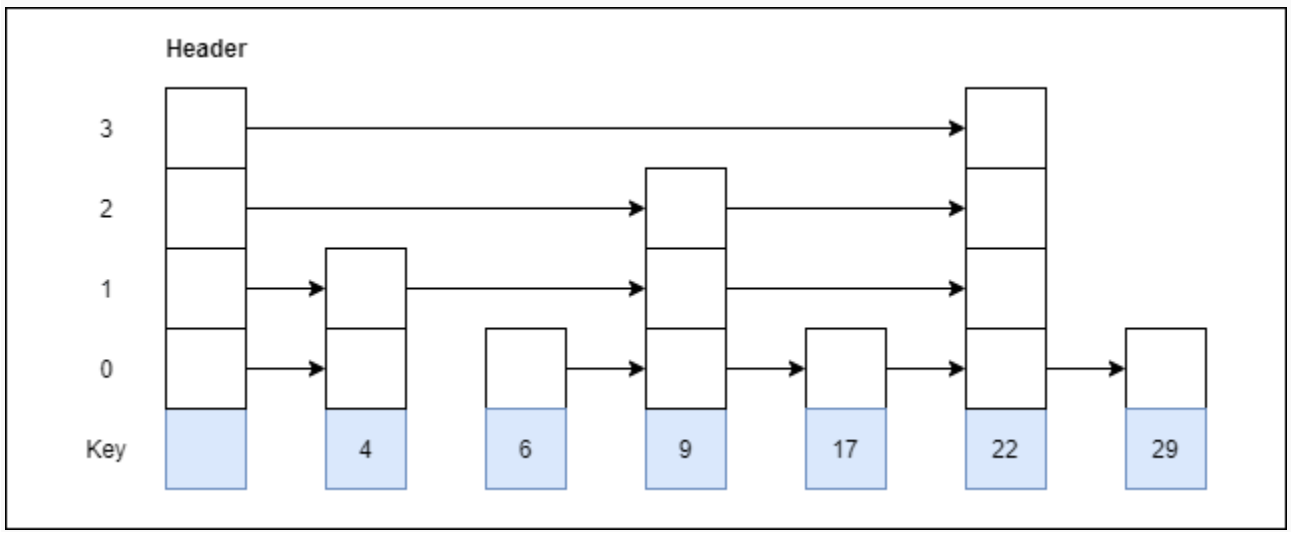
Step 4: Insert 9 with level 3



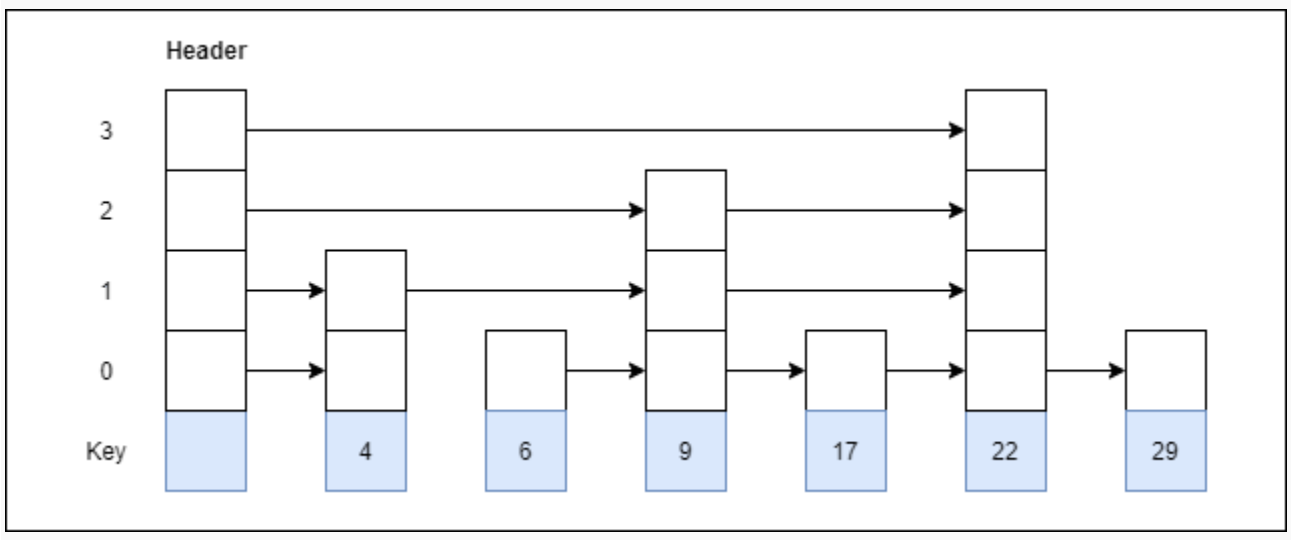
Step 5: Insert 17 with level 1



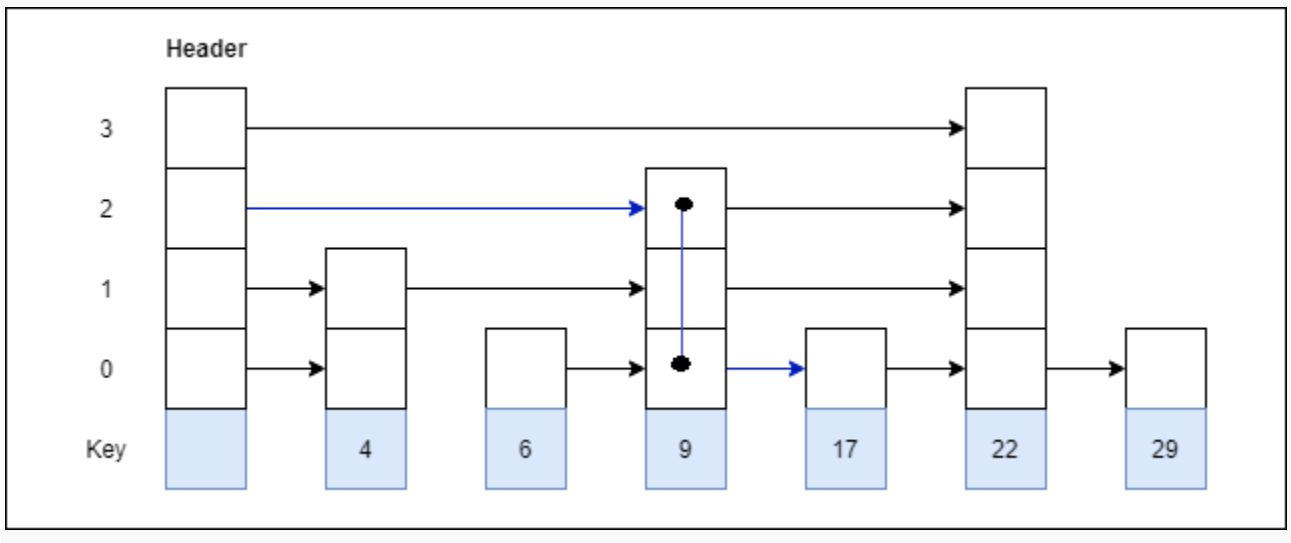
Step 6: Insert 4 with level 2



Example 2: Consider this example where we want to search for key 17.



Ans:



Advantages of the Skip list

1. If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
2. The skip list is simple to implement as compared to the hash table and the binary search tree.
3. It is very simple to find a node in the list because it stores the nodes in sorted form.

4. The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
5. The skip list is a robust and reliable list.

Disadvantages of the Skip list

1. It requires more memory than the balanced tree.
2. Reverse searching is not allowed.
3. The skip list searches the node much slower than the linked list.

Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. It is also used with the QMap template class.
4. The indexing of the skip list is used in running median problems.
5. The skip list is used for the delta-encoding posting in the Lucene search.

What is a Stack?

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

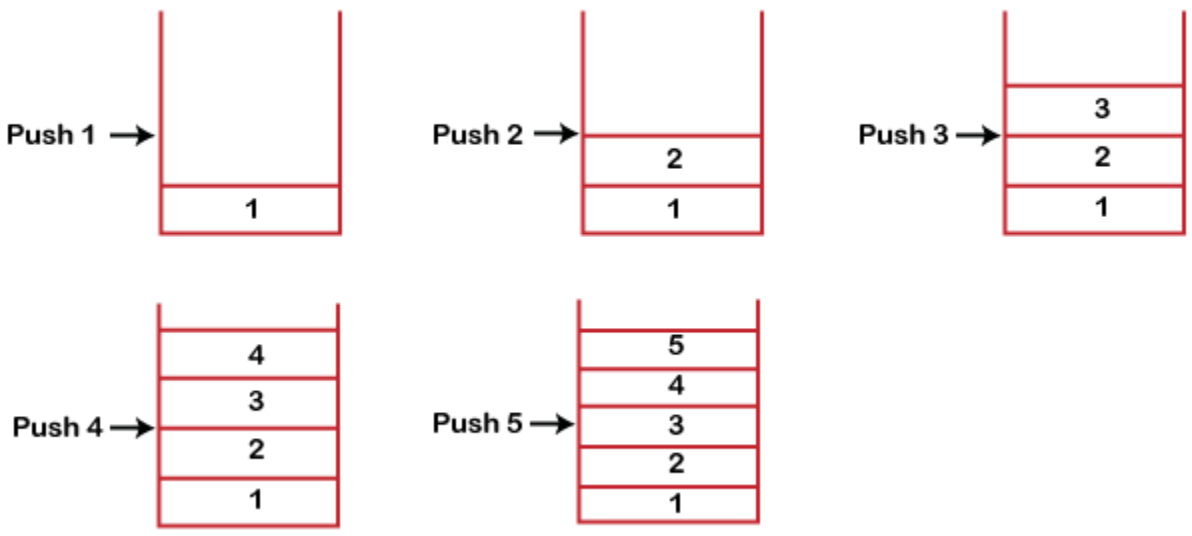
Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

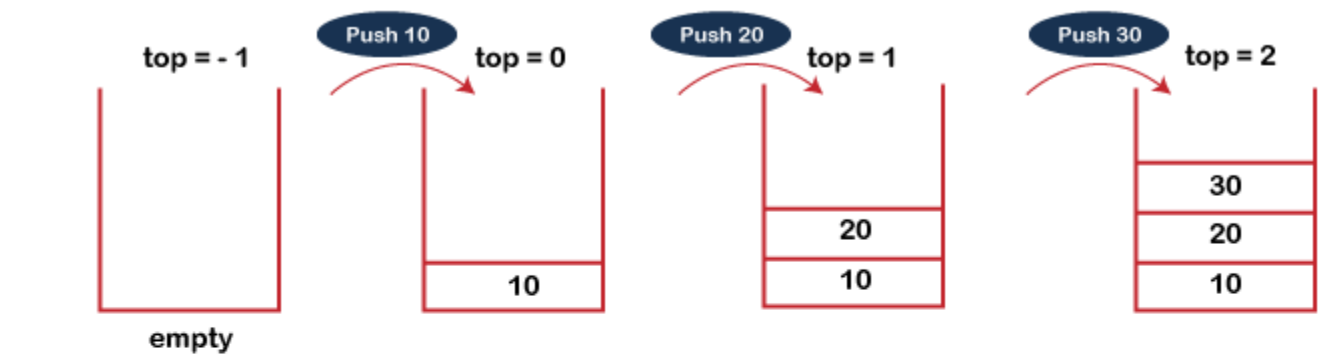
The following are some common operations implemented on the stack:

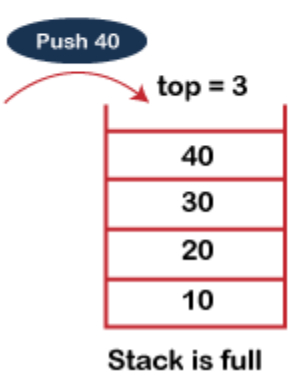
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.

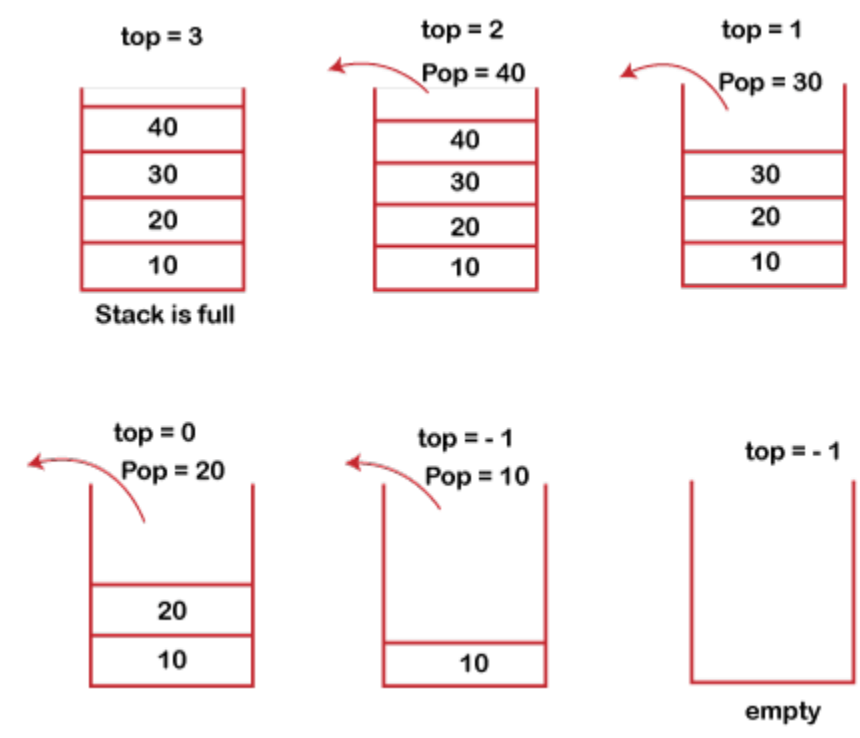




POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

The following are the applications of the stack:

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

```
1. int main()
2. {
3.    cout<<"Hello";
```

```
4.     cout<<"javaTpoint";
5.  }
```

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack.
  First, we push all the characters of the string in a stack until we reach the null character.
  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
- `Infix to prefix`
- `Infix to postfix`
- `Prefix to infix`
- `Prefix to postfix`
  `Postfix to infix`
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
1. begin
2.     if top = n then stack full
3.     top = top + 1
4.     stack (top) : = item;
5. end
```

Time Complexity : o(1)

implementation of push algorithm in C language

```c
void push (int val,int n) //n is size of the stack
{
    if (top == n )
    printf("\n Overflow");
    else
    {
    top = top +1;
    stack[top] = val;
    }
}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
    if top = 0 then stack empty;
    item := stack(top);
    top = top - 1;
end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```c
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

```
Begin
    if top = -1 then stack empty
    item = stack[top]
    return item
End
```

Time complexity: o(n)

Implementation of Peek algorithm in C language

```
int peek()
{
    if (top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack [top];
    }
}
```

C program

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");

printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
```

```
44.                 printf("Please Enter valid choice ");
45.             }
46.         };
47.     }
48. }
49.
50. void push ()
51. {
52.     int val;
53.     if (top == n )
54.     printf("\n Overflow");
55.     else
56.     {
57.         printf("Enter the value?");
58.         scanf("%d",&val);
59.         top = top +1;
60.         stack[top] = val;
61.     }
62. }
63.
64. void pop ()
65. {
66.     if(top == -1)
67.     printf("Underflow");
68.     else
69.     top = top -1;
70. }
71. void show()
72. {
73.     for (i=top;i>=0;i--)
74.     {
75.         printf("%d\n",stack[i]);
76.     }
77.     if(top == -1)
78.     {
79.         printf("Stack is empty");
80.     }
81. }
```

Java Program

```
1.  import java.util.Scanner;
2.  class Stack
3.  {
4.      int top;
5.      int maxsize = 10;
6.      int[] arr = new int[maxsize];
7.
8.
9.      boolean isEmpty()
10.     {
11.         return (top < 0);
12.     }
13.     Stack()
14.     {
15.         top = -1;
16.     }
17.     boolean push (Scanner sc)
```

```
18.    {
19.        if(top == maxsize-1)
20.        {
21.            System.out.println("Overflow !!");
22.            return false;
23.        }
24.        else
25.        {
26.            System.out.println("Enter Value");
27.            int val = sc.nextInt();
28.            top++;
29.            arr[top]=val;
30.            System.out.println("Item pushed");
31.            return true;
32.        }
33.    }
34.    boolean pop ()
35.    {
36.        if (top == -1)
37.        {
38.            System.out.println("Underflow !!");
39.            return false;
40.        }
41.        else
42.        {
43.            top --;
44.            System.out.println("Item popped");
45.            return true;
46.        }
47.    }
48.    void display ()
49.    {
50.        System.out.println("Printing stack elements .....");
51.        for(int i = top; i>=0;i--)
52.        {
53.            System.out.println(arr[i]);
54.        }
55.    }
56. }
57. public class Stack_Operations {
58. public static void main(String[] args) {
59.    int choice=0;
60.    Scanner sc = new Scanner(System.in);
61.    Stack s = new Stack();
62.    System.out.println("*********Stack operations using array*********\n");
63.    System.out.println("\n-----------------------------------------------\n");
64.    while(choice != 4)
65.    {
66.        System.out.println("\nChose one from the below options...\n");
67.        System.out.println("\n1.Push\n2.Pop\n3.Show\n4.Exit");
68.        System.out.println("\n Enter your choice \n");
69.        choice = sc.nextInt();
70.        switch(choice)
71.        {
72.            case 1:
73.            {
74.                s.push(sc);
```

```
            break;
        }
        case 2:
        {
            s.pop();
            break;
        }
        case 3:
        {
            s.display();
            break;
        }
        case 4:
        {
            System.out.println("Exiting....");
            System.exit(0);
            break;
        }
            default:
            {
                System.out.println("Please Enter valid choice ");
            }
        };
    }
}
        }
```

C# Program

```
using System;

public class Stack
{
    int top;
    int maxsize=10;
    int[] arr = new int[10];
    public static void Main()
    {
    Stack st = new Stack();
    st.top=-1;
    int choice=0;
    Console.WriteLine("*********Stack operations using array*********");
    Console.WriteLine("\n-----------------------------------------------\n");
    while(choice != 4)
    {
        Console.WriteLine("Chose one from the below options...\n");
        Console.WriteLine("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        Console.WriteLine("\n Enter your choice \n");
        choice = Convert.ToInt32(Console.ReadLine());
        switch(choice)
        {
            case 1:
            {
                st.push();
                break;
            }
            case 2:
            {
```

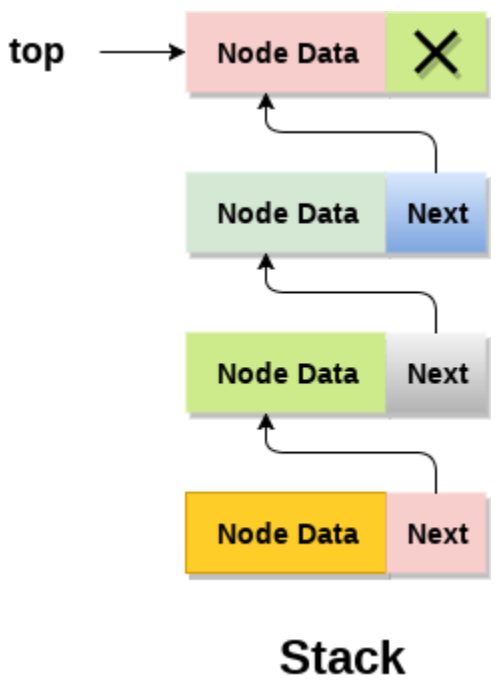
```
30.             st.pop();
31.             break;
32.         }
33.         case 3:
34.         {
35.             st.show();
36.             break;
37.         }
38.         case 4:
39.         {
40.         Console.WriteLine("Exiting....");
41.             break;
42.         }
43.         default:
44.         {
45.             Console.WriteLine("Please Enter valid choice ");
46.             break;
47.         }
48.         };
49.     }
50. }
51.
52. Boolean push ()
53. {
54.     int val;
55.     if(top == maxsize-1)
56.     {
57.
58.         Console.WriteLine("\n Overflow");
59.         return false;
60.     }
61.     else
62.     {
63.         Console.WriteLine("Enter the value?");
64.         val = Convert.ToInt32(Console.ReadLine());
65.         top = top +1;
66.         arr[top] = val;
67.         Console.WriteLine("Item pushed");
68.         return true;
69.     }
70. }
71.
72. Boolean pop ()
73. {
74.     if (top == -1)
75.     {
76.         Console.WriteLine("Underflow");
77.         return false;
78.     }
79.
80.     else
81.
82.     {
83.         top = top -1;
84.         Console.WriteLine("Item popped");
85.         return true;
86.     }
```

```
87. }
88. void show()
89. {
90.
91.     for (int i=top;i>=0;i--)
92.     {
93.         Console.WriteLine(arr[i]);
94.     }
95.     if(top == -1)
96.     {
97.         Console.WriteLine("Stack is empty");
98.     }
99. }
100.         }
```

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



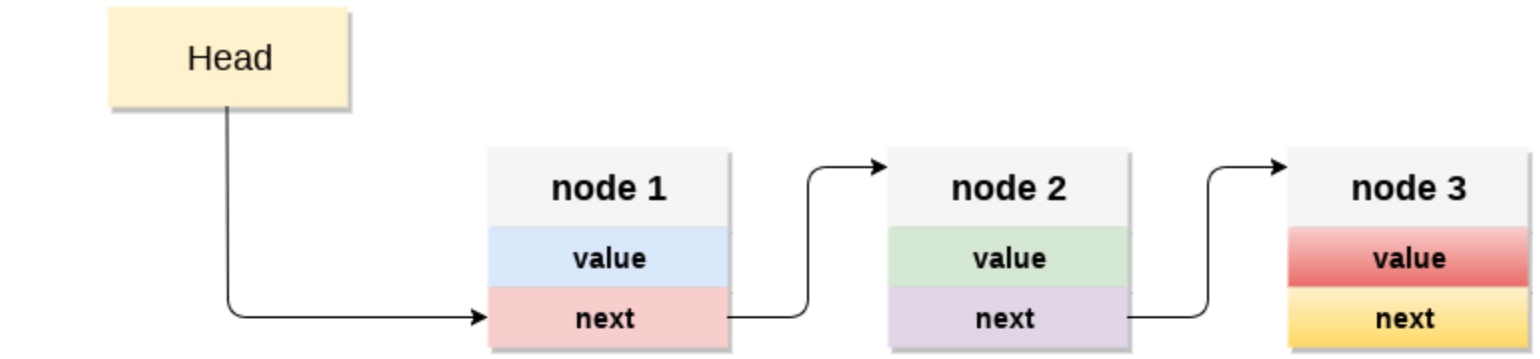
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

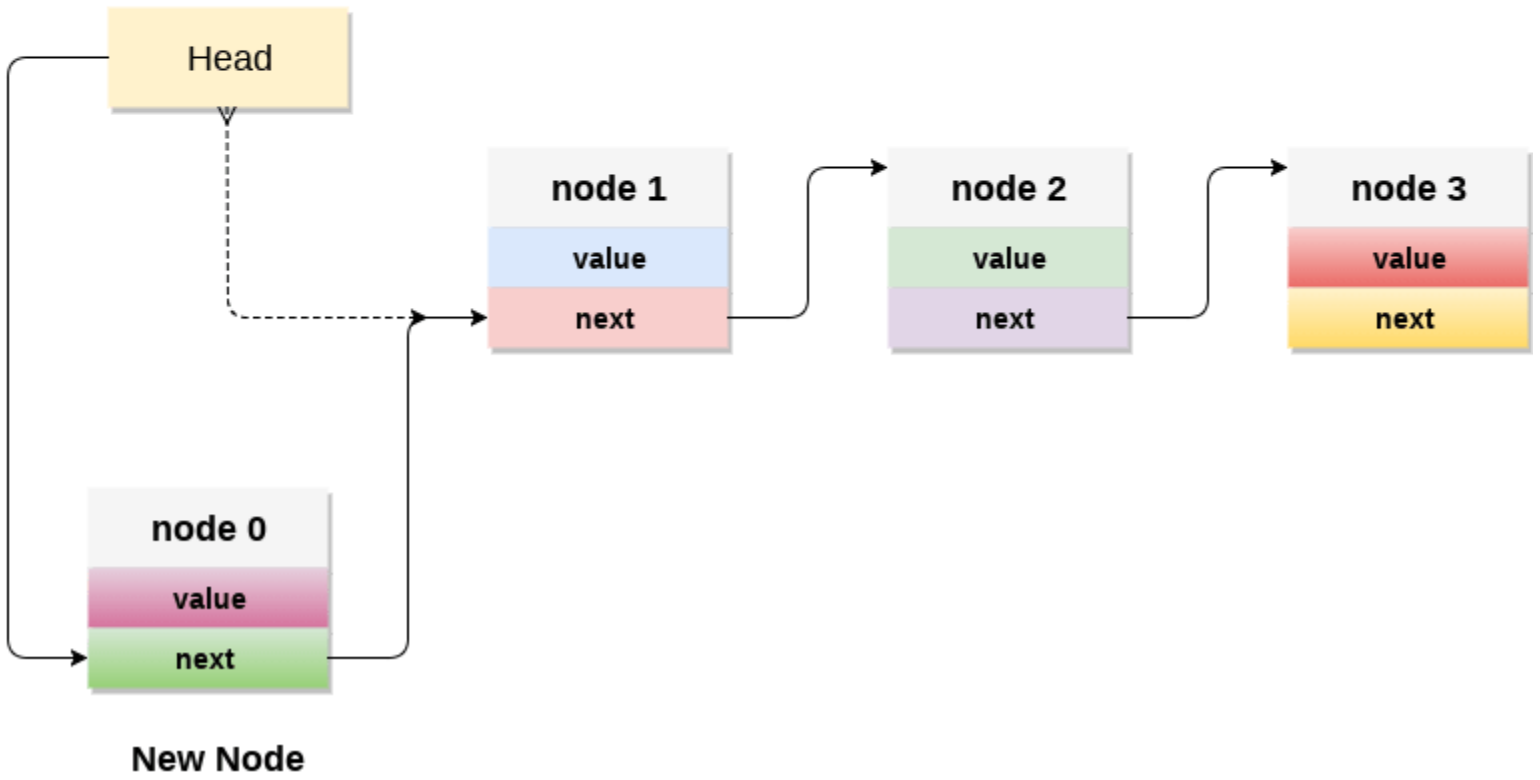
Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

   Time Complexity : o(1)





C implementation :

```
1.  void push ()
2.  {
3.      int val;
4.      struct node *ptr =(struct node*)malloc(sizeof(struct node));
5.      if(ptr == NULL)
6.      {
7.          printf("not able to push the element");
8.      }
9.      else
10.     {
11.      printf("Enter the value");
12.     scanf("%d",&val);
13.      if(head==NULL)
14.     {
15.          ptr->val = val;
16.         ptr -> next = NULL;
17.          head=ptr;
18.     }
19.      else
20.     {
21.          ptr->val = val;
22.         ptr->next = head;
23.          head=ptr;
24.
25.      }
26.     printf("Item pushed");
27.
```

```
28.     }
29. }
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

30. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
31. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
1.  void pop()
2.  {
3.      int item;
4.      struct node *ptr;
5.      if (head == NULL)
6.      {
7.          printf("Underflow");
8.      }
9.      else
10.     {
11.         item = head->val;
12.         ptr = head;
13.         head = head->next;
14.         free(ptr);
15.         printf("Item popped");
16.
17.     }
18. }
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

19. Copy the head pointer into a temporary pointer.
20. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

C Implementation

```
1.  void display()
2.  {
3.      int i;
4.      struct node *ptr;
5.      ptr=head;
6.      if(ptr == NULL)
7.      {
8.          printf("Stack is empty\n");
9.      }
10.     else
11.     {
12.         printf("Printing Stack elements \n");
```

```
        while(ptr!=NULL)
        {
             printf("%d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}
```

Menu Driven program in C implementing all the stack operations using linked list :

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;

void main ()
{
    int choice=0;
    printf("\n*********Stack operations using linked list*********\n");
    printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("\n\nChose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
```

```
            }
    };
}
}
void push ()
{
    int val;
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
       scanf("%d",&val);
        if(head==NULL)
       {
            ptr->val = val;
           ptr -> next = NULL;
            head=ptr;
       }
        else
       {
            ptr->val = val;
           ptr->next = head;
            head=ptr;

        }
       printf("Item pushed");

    }
}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
       ptr = head;
        head = head->next;
       free(ptr);
        printf("Item popped");

    }
        }
        void display()
        {
            int i;
            struct node *ptr;
            ptr=head;
```

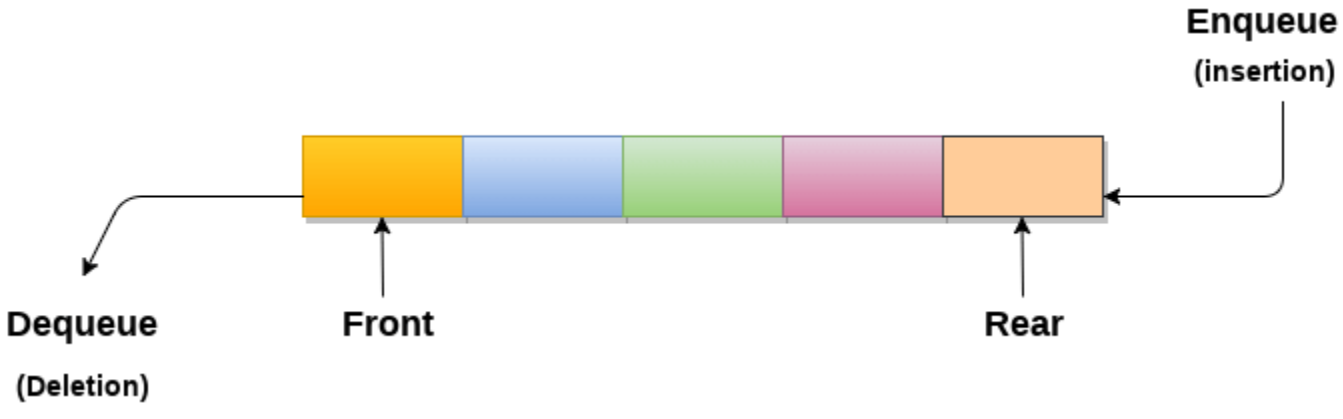
```
106.            if(ptr == NULL)
107.            {
108.                printf("Stack is empty\n");
109.            }
110.            else
111.            {
112.                printf("Printing Stack elements \n");
113.                 while(ptr!=NULL)
114.                {
115.                     printf("%d\n",ptr->val);
116.                    ptr = ptr->next;
117.                 }
118.            }
119.        }
```

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	**Time Complexity**								**Space Compleity**
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Types of Queues

Before understanding the types of queues, we first look at '**what is Queue**'.

What is the Queue?

A queue in the data structure can be considered similar to the queue in the real-world. A queue is a data structure in which whatever comes first will go out first. It follows the FIFO (First-In-First-Out) policy. In Queue, the insertion is done from one end known as the rear end or the tail of the queue, whereas the deletion is done from another end known as the front end or the head of the queue. In other words, it can be defined as a list or a collection with a constraint that the insertion can be performed at one end called as the rear end or tail of the queue and deletion is performed on another end called as the front end or the head of the queue.

Operations on Queue

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

A Queue can be represented as a container opened from both the sides in which the element can be enqueued from one side and dequeued from another side as shown in the below figure:

Implementation of Queue

There are two ways of implementing the Queue:

- **Sequential allocation:** The sequential allocation in a Queue can be implemented using an array. **For more details, click on the below link:** https://www.javatpoint.com/array-representation-of-queue
- **Linked list allocation:** The linked list allocation in a Queue can be implemented using a linked list. **For more details, click on the below link:** https://www.javatpoint.com/linked-list-implementation-of-queue

What are the use cases of Queue?

Here, we will see the real-world scenarios where we can use the Queue data structure. The Queue data structure is mainly used where there is a shared resource that has to serve the multiple requests but can serve a single request at a time. In such cases, we need to use the Queue data structure for queuing up the requests. The request that arrives first in the queue will be served first. The following are the real-world scenarios in which the Queue concept is used:

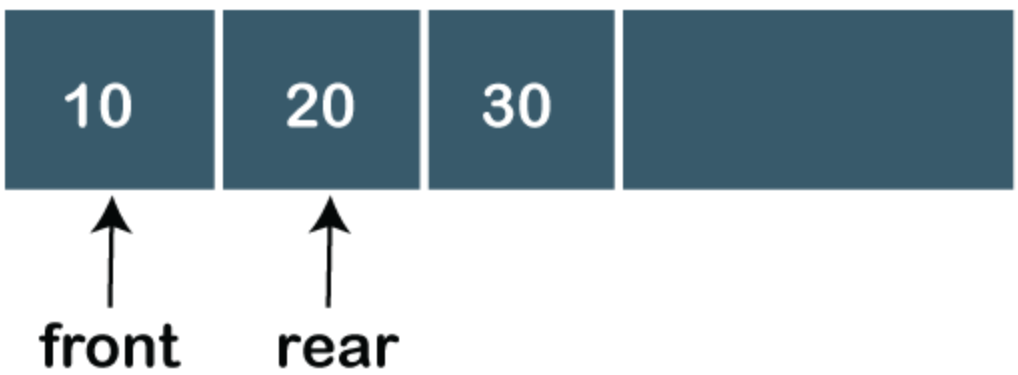
- Suppose we have a printer shared between various machines in a network, and any machine or computer in a network can send a print request to the printer. But, the printer can serve a single request at a time, i.e., a printer can print a single document at a time. When any print request comes from the network, and if the printer is busy, the printer's program will put the print request in a queue.
- . If the requests are available in the Queue, the printer takes a request from the front of the queue, and serves it.
- The processor in a computer is also used as a shared resource. There are multiple requests that the processor must execute, but the processor can serve a single request or execute a single process at a time. Therefore, the processes are kept in a Queue for execution.

Types of Queue

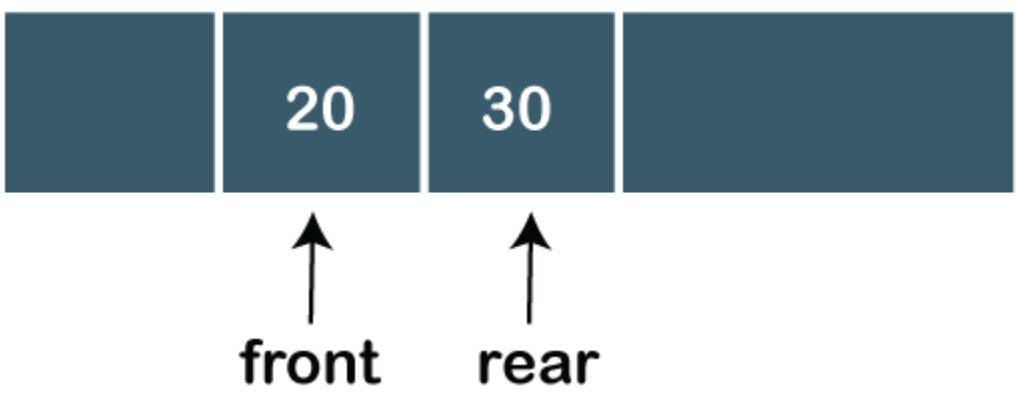
There are four types of Queues:

- **Linear Queue**

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule. The linear Queue can be represented, as shown in the below figure:



The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:

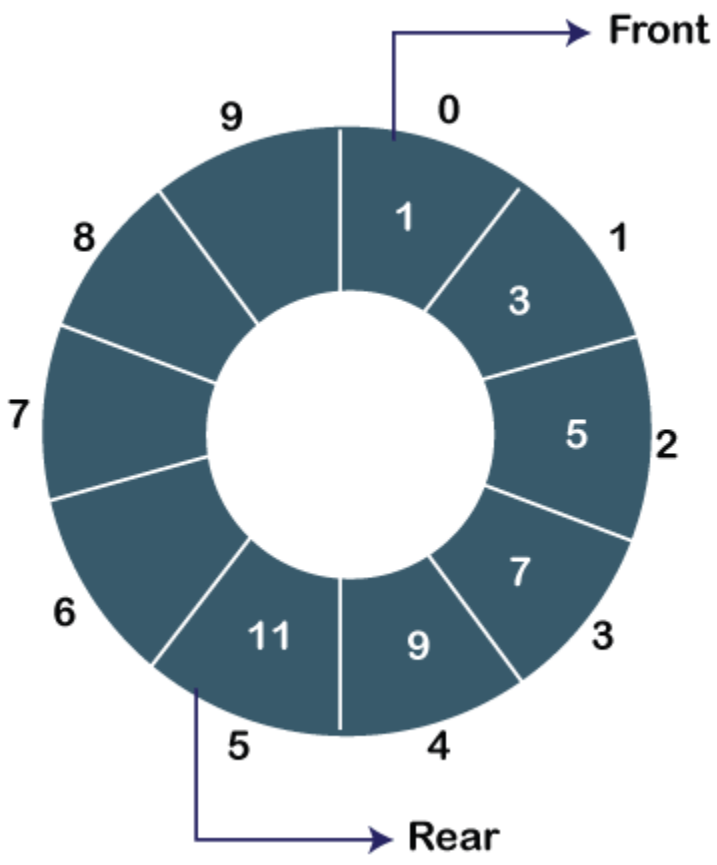


In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

The major drawback of using a **linear Queue** is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the **overflow** condition as the rear is pointing to the last element of the Queue.

- **Circular Queue**

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as **Ring Buffer** as all the ends are connected to another end. The circular queue can be represented as:



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

To know more about circular queue, click on the below link: https://www.javatpoint.com/circular-queue

- **Priority Queue**

A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:

The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

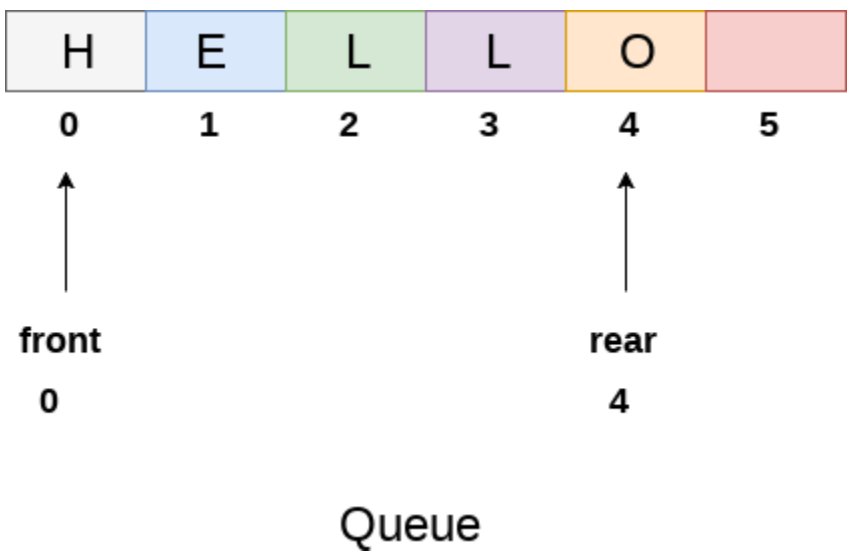
- **Deque**

Both the Linear Queue and Deque are different as the linear queue follows the FIFO principle whereas, deque does not follow the FIFO principle. In Deque, the insertion and deletion can occur from both ends.

To know more about Deque, click on the below link: https://www.javatpoint.com/ds-deque

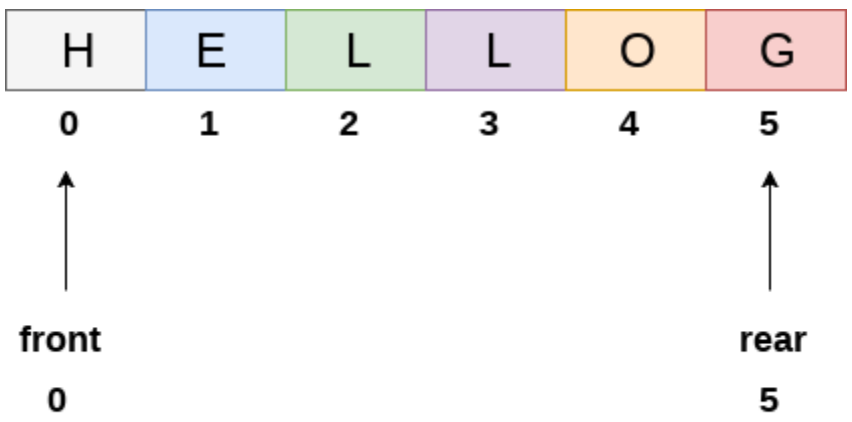
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



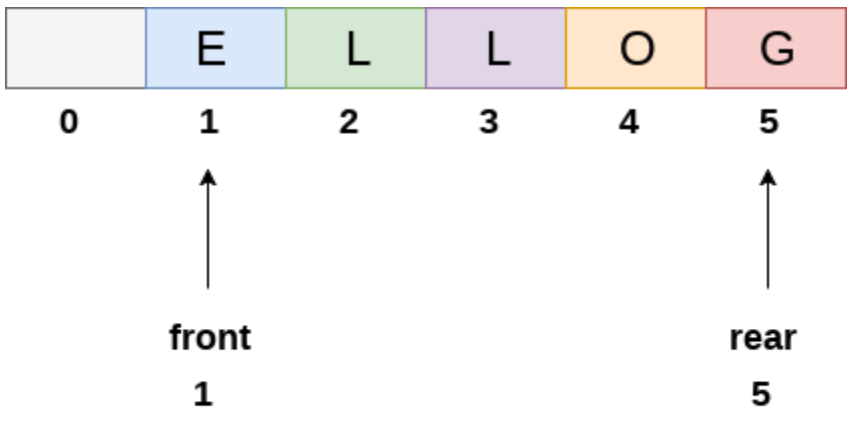
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

C Function

```
1. void insert (int queue[], int max, int front, int rear, int item)
2. {
3.     if (rear + 1 == max)
4.     {
5.         printf("overflow");
6.     }
7.     else
8.     {
9.         if(front == -1 && rear == -1)
10.        {
11.            front = 0;
12.           rear = 0;
13.        }
14.        else
15.        {
16.           rear = rear + 1;
17.        }
18.        queue[rear]=item;
19.    }
```

```
20. }
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
- **Step 2:** EXIT

C Function

```
1.  int delete (int queue[], int max, int front, int rear)
2.  {
3.      int y;
4.      if (front == -1 || front > rear)
5.
6.      {
7.          printf("underflow");
8.      }
9.      else
10.     {
11.         y = queue[front];
12.        if(front == rear)
13.        {
14.            front = rear = -1;
15.             else
16.            front = front + 1;
17.
18.        }
19.         return y;
20.     }
21. }
```

Menu driven program to implement queue using array

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  #define maxsize 5
4.  void insert();
5.  void delete();
6.  void display();
7.  int front = -1, rear = -1;
8.  int queue[maxsize];
9.  void main ()
10. {
11.     int choice;
12.     while(choice != 4)
13.     {
14.         printf("\n*************************Main Menu*****************************\n");
15.         printf("\n===========================================================================
    \n");
16.         printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
17.         printf("\nEnter your choice ?");
18.         scanf("%d",&choice);
```

```
19.       switch(choice)
20.       {
21.           case 1:
22.          insert();
23.           break;
24.          case 2:
25.           delete();
26.          break;
27.           case 3:
28.          display();
29.           break;
30.          case 4:
31.           exit(0);
32.          break;
33.           default:
34.          printf("\nEnter valid choice??\n");
35.       }
36.   }
37. }
38. void insert()
39. {
40.     int item;
41.     printf("\nEnter the element\n");
42.     scanf("\n%d",&item);
43.     if(rear == maxsize-1)
44.     {
45.         printf("\nOVERFLOW\n");
46.        return;
47.     }
48.     if(front == -1 && rear == -1)
49.     {
50.        front = 0;
51.         rear = 0;
52.     }
53.     else
54.     {
55.         rear = rear+1;
56.     }
57.     queue[rear] = item;
58.     printf("\nValue inserted ");
59.
60. }
61. void delete()
62. {
63.     int item;
64.     if (front == -1 || front > rear)
65.     {
66.         printf("\nUNDERFLOW\n");
67.         return;
68.
69.     }
70.     else
71.     {
72.         item = queue[front];
73.         if(front == rear)
74.         {
75.             front = -1;
```

```
76.             rear = -1 ;
77.         }
78.         else
79.         {
80.             front = front + 1;
81.         }
82.         printf("\nvalue deleted ");
83.     }
84.
85.
86. }
87.
88. void display()
89. {
90.     int i;
91.     if(rear == -1)
92.     {
93.         printf("\nEmpty queue\n");
94.     }
95.     else
96.     {   printf("\nprinting values .....\n");
97.         for(i=front;i<=rear;i++)
98.         {
99.             printf("\n%d\n",queue[i]);
100.                }
101.            }
102.        }
```

Output:

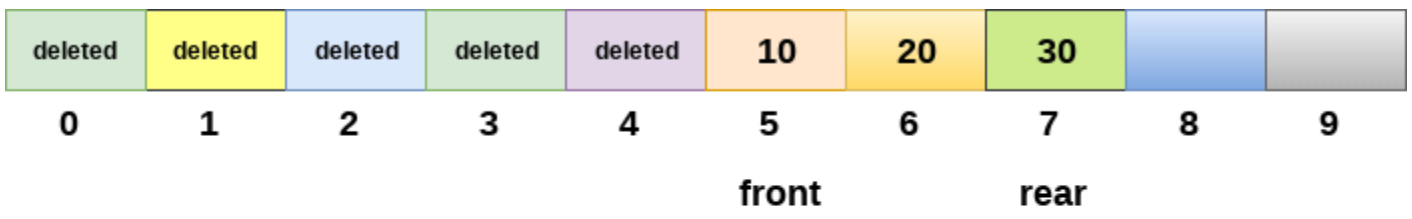
```
*************Main Menu**************

==========================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
123

Value inserted

*************Main Menu**************

==========================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

*************Main Menu**************

==================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

value deleted

*************Main Menu**************
```

```
==========================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

*************Main Menu**************

==========================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4
```

Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

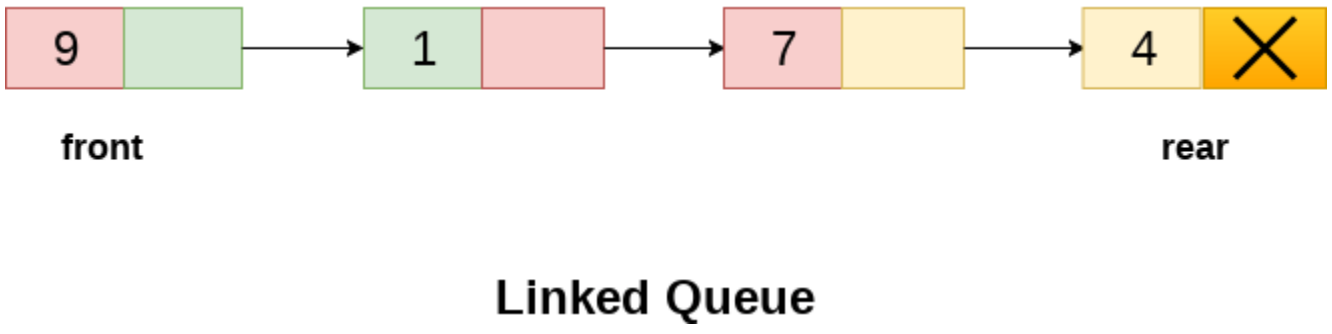
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

```
1. Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
1. ptr -> data = item;
2.         if(front == NULL)
3.         {
4.             front = ptr;
5.              rear = ptr;
6.             front -> next = NULL;
7.              rear -> next = NULL;
8.         }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
1. rear -> next = ptr;
2.              rear = ptr;
3.               rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
  SET FRONT = REAR = PTR
  SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
  SET REAR -> NEXT = PTR
  SET REAR = PTR

SET REAR -> NEXT = NULL
[END OF IF]

- **Step 4:** END

C Function

```
1. void insert(struct node *ptr, int item; )
2. {
3.
4.
5.      ptr = (struct node *) malloc (sizeof(struct node));
6.     if(ptr == NULL)
7.     {
8.         printf("\nOVERFLOW\n");
9.          return;
10.    }
11.    else
12.    {
13.         ptr -> data = item;
14.        if(front == NULL)
15.        {
16.            front = ptr;
17.             rear = ptr;
18.            front -> next = NULL;
19.             rear -> next = NULL;
20.        }
21.        else
22.        {
23.             rear -> next = ptr;
24.            rear = ptr;
25.             rear->next = NULL;
26.        }
27.    }
28. }
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
1. ptr = front;
2.          front = front -> next;
3.          free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

- **Step 1:** IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

C Function

```c
void delete (struct node *ptr)
{
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
```

Menu-Driven Program implementing all the operations on Linked Queue

```c
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n************************Main Menu*****************************\n");
        printf("\n==========================================================================
    \n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",& choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
```

```
41. }
42. void insert()
43. {
44.     struct node *ptr;
45.     int item;
46.
47.     ptr = (struct node *) malloc (sizeof(struct node));
48.     if(ptr == NULL)
49.     {
50.         printf("\nOVERFLOW\n");
51.         return;
52.     }
53.     else
54.     {
55.         printf("\nEnter value?\n");
56.         scanf("%d",&item);
57.         ptr -> data = item;
58.         if(front == NULL)
59.         {
60.             front = ptr;
61.             rear = ptr;
62.             front -> next = NULL;
63.             rear -> next = NULL;
64.         }
65.         else
66.         {
67.             rear -> next = ptr;
68.             rear = ptr;
69.             rear->next = NULL;
70.         }
71.     }
72. }
73. void delete ()
74. {
75.     struct node *ptr;
76.     if(front == NULL)
77.     {
78.         printf("\nUNDERFLOW\n");
79.         return;
80.     }
81.     else
82.     {
83.         ptr = front;
84.         front = front -> next;
85.         free(ptr);
86.     }
87. }
88. void display()
89. {
90.     struct node *ptr;
91.     ptr = front;
92.     if(front == NULL)
93.     {
94.         printf("\nEmpty queue\n");
95.     }
96.     else
97.     {   printf("\nprinting values .....\n");
```

```
98.         while(ptr != NULL)
99.         {
100.                    printf("\n%d\n",ptr -> data);
101.                     ptr = ptr -> next;
102.                }
103.            }
104.         }
```

Output:

```
***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
123

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
90

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

123

90

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

***********Main Menu**********

==============================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit
```

```
Enter your choice ?4
```

Circular Queue

Why was the concept of the circular queue introduced?

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.

As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0^{th} position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.

- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- front== rear + 1;

Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT != 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

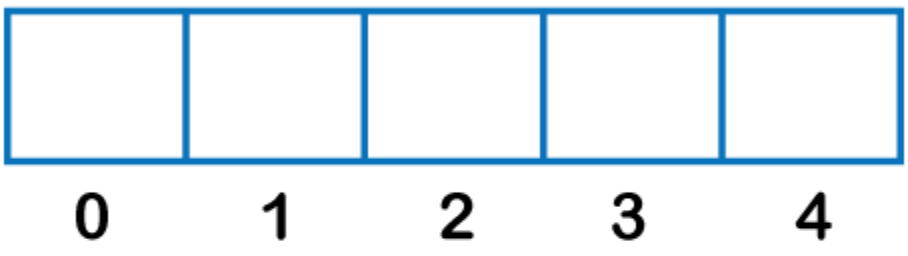
Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1

```
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]
```

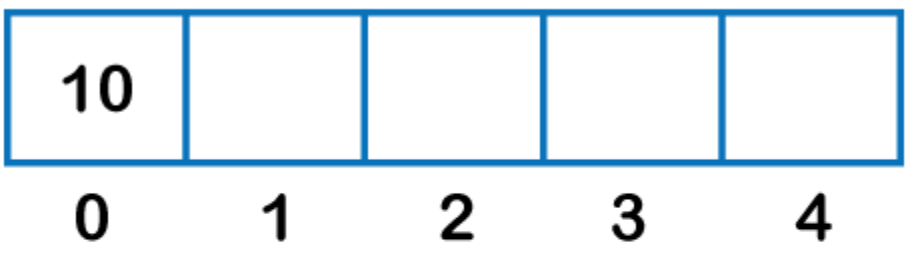
Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.



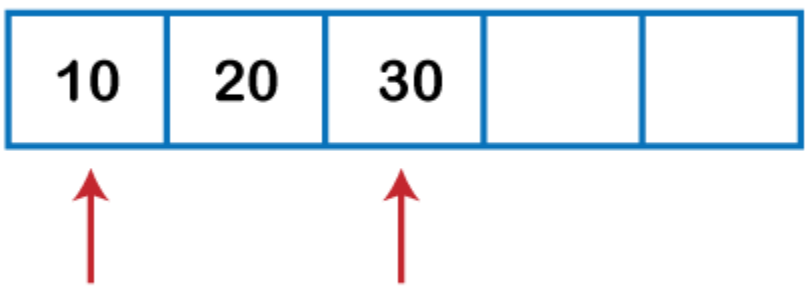
Front = -1

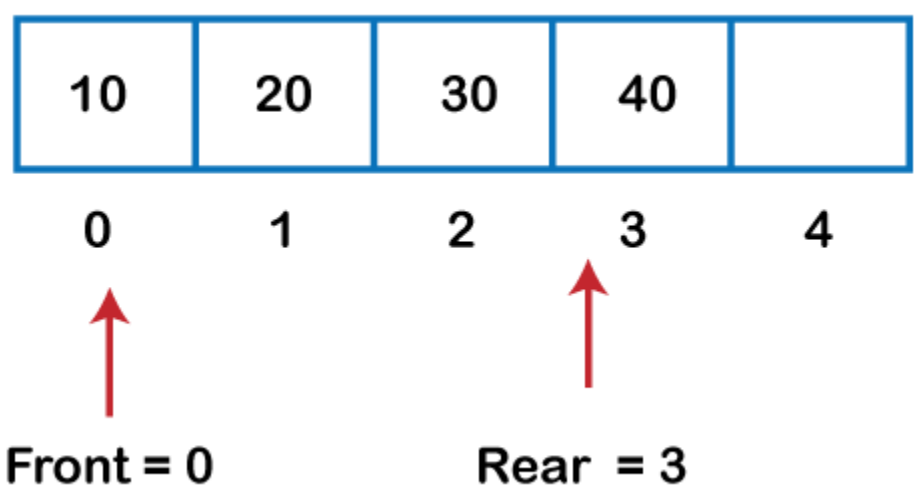
Rear = -1

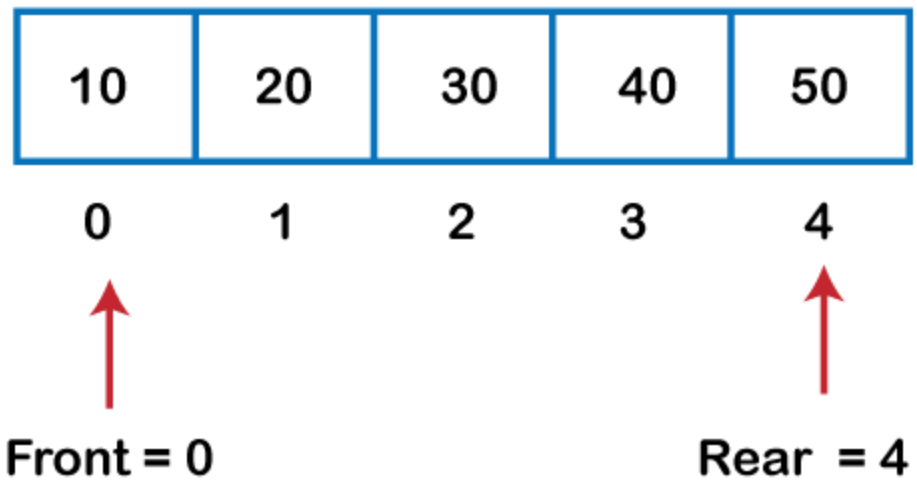


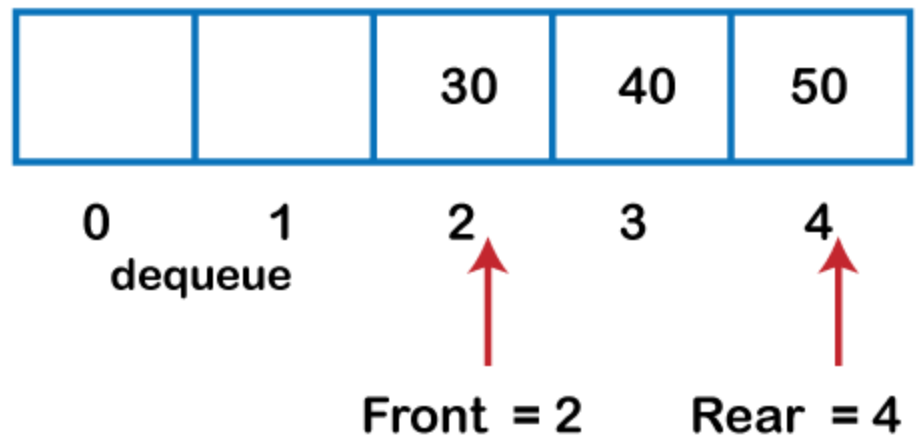
Front = 0

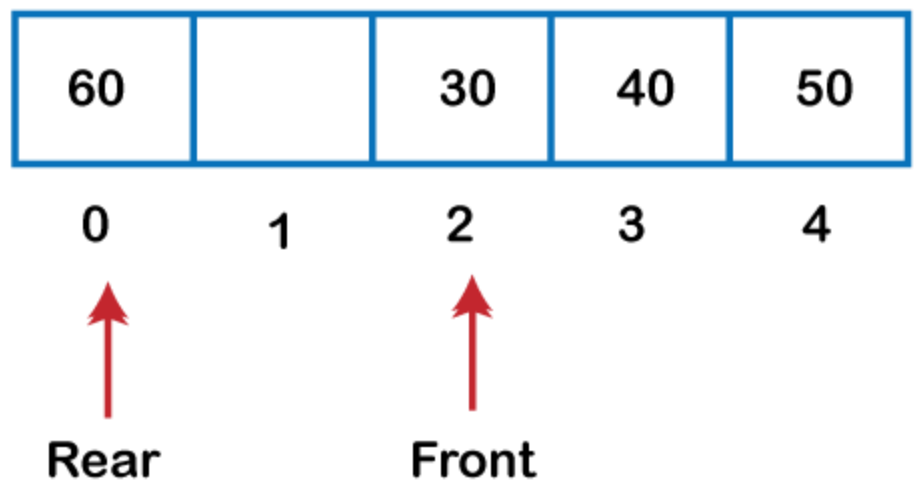
Rear = 0

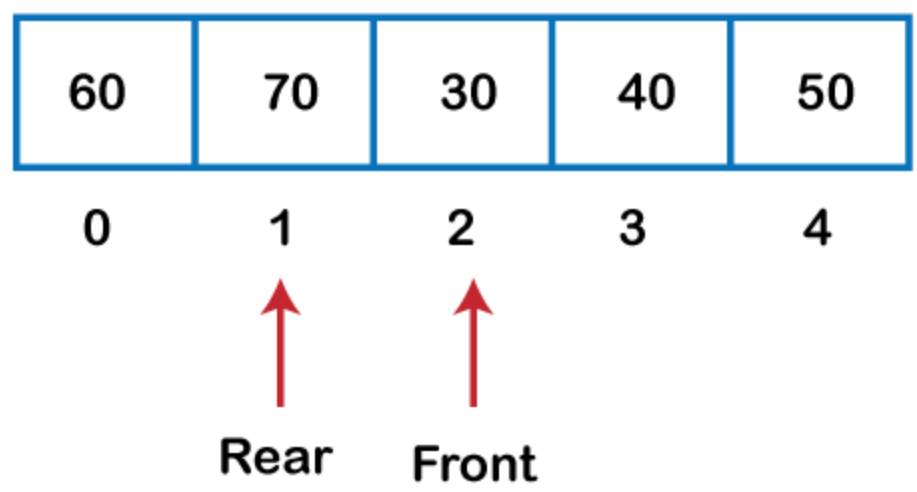












Implementation of circular queue using Array

```
1. #include <stdio.h>
2.
3. # define max 6
4. int queue[max];  // array declaration
5. int front=-1;
6. int rear=-1;
7. // function to insert an element in a circular queue
```

```
8.  void enqueue(int element)
9.  {
10.    if(front==-1 && rear==-1)   // condition to check queue is empty
11.    {
12.        front=0;
13.        rear=0;
14.        queue[rear]=element;
15.    }
16.    else if((rear+1)%max==front)  // condition to check queue is full
17.    {
18.        printf("Queue is overflow..");
19.    }
20.    else
21.    {
22.        rear=(rear+1)%max;       // rear is incremented
23.        queue[rear]=element;     // assigning a value to the queue at the rear position.
24.    }
25. }
26.
27. // function to delete the element from the queue
28. int dequeue()
29. {
30.     if((front==-1) && (rear==-1))  // condition to check queue is empty
31.     {
32.         printf("\nQueue is underflow..");
33.     }
34.  else if(front==rear)
35. {
36.    printf("\nThe dequeued element is %d", queue[front]);
37.    front=-1;
38.    rear=-1;
39. }
40. else
41. {
42.     printf("\nThe dequeued element is %d", queue[front]);
43.    front=(front+1)%max;
44. }
45. }
46. // function to display the elements of a queue
47. void display()
48. {
49.    int i=front;
50.    if(front==-1 && rear==-1)
51.    {
52.        printf("\n Queue is empty..");
53.    }
54.    else
55.    {
56.        printf("\nElements in a Queue are :");
57.        while(i<=rear)
58.        {
59.            printf("%d,", queue[i]);
60.            i=(i+1)%max;
61.        }
62.    }
63. }
64. int main()
```

```
{
    int choice=1,x;   // variables declaration

    while(choice<4 && choice!=0)   // while loop
    {
    printf("\n Press 1: Insert an element");
    printf("\nPress 2: Delete an element");
    printf("\nPress 3: Display the element");
    printf("\nEnter your choice");
    scanf("%d", &choice);

    switch(choice)
    {

        case 1:

        printf("Enter the element which is to be inserted");
        scanf("%d", &x);
        enqueue(x);
        break;
        case 2:
        dequeue();
        break;
        case 3:
        display();

    }}
    return 0;
}
```

Output:

```
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
10

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
20

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
30

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
3

Elements in a Queue are :10,20,30,
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
2

The dequeued element is 10
```

Implementation of circular queue using linked list

As we know that linked list is a linear data structure that stores two parts, i.e., data part and the address part where address part contains the address of the next node. Here, linked list is used to implement the circular queue; therefore, the linked list follows the properties of the Queue. When we are implementing the circular queue using linked list then both the ***enqueue and dequeue*** operations take ***O(1)*** time.

```
1. #include <stdio.h>
2. // Declaration of struct type node
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
8. struct node *front=-1;
9. struct node *rear=-1;
10. // function to insert the element in the Queue
11. void enqueue(int x)
12. {
13.     struct node *newnode;  // declaration of pointer of struct node type.
14.     newnode=(struct node *)malloc(sizeof(struct node));  // allocating the memory to the newnode
15.     newnode->data=x;
16.     newnode->next=0;
17.     if(rear==-1)  // checking whether the Queue is empty or not.
18.     {
19.         front=rear=newnode;
20.         rear->next=front;
21.     }
22.     else
23.     {
24.         rear->next=newnode;
25.         rear=newnode;
26.         rear->next=front;
27.     }
28. }
29.
30. // function to delete the element from the queue
31. void dequeue()
32. {
33.     struct node *temp;   // declaration of pointer of node type
34.     temp=front;
35.     if((front==-1)&&(rear==-1))  // checking whether the queue is empty or not
36.     {
37.         printf("\nQueue is empty");
38.     }
39.     else if(front==rear)  // checking whether the single element is left in the queue
40.     {
41.         front=rear=-1;
42.         free(temp);
43.     }
44.     else
45.     {
46.         front=front->next;
47.         rear->next=front;
48.         free(temp);
49.     }
50. }
51.
```

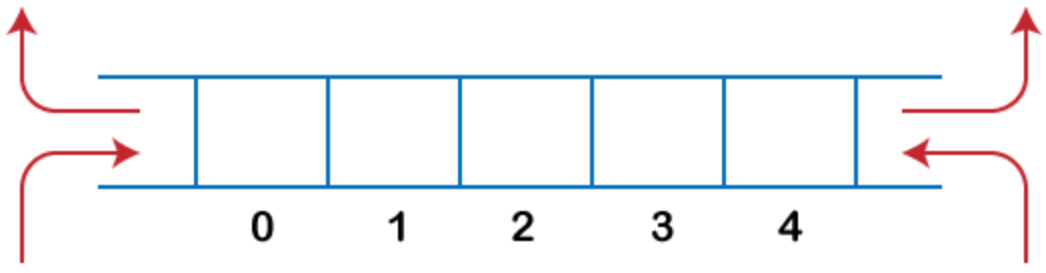
```
52. // function to get the front of the queue
53. int peek()
54. {
55.     if((front==-1) &&(rear==-1))
56.     {
57.         printf("\nQueue is empty");
58.     }
59.     else
60.     {
61.         printf("\nThe front element is %d", front->data);
62.     }
63. }
64.
65. // function to display all the elements of the queue
66. void display()
67. {
68.     struct node *temp;
69.     temp=front;
70.     printf("\n The elements in a Queue are : ");
71.     if((front==-1) && (rear==-1))
72.     {
73.         printf("Queue is empty");
74.     }
75.
76.     else
77.     {
78.         while(temp->next!=front)
79.         {
80.             printf("%d,", temp->data);
81.             temp=temp->next;
82.         }
83.         printf("%d", temp->data);
84.     }
85. }
86.
87. void main()
88. {
89.     enqueue(34);
90.     enqueue(10);
91.     enqueue(23);
92.     display();
93.     dequeue();
94.     peek();
95. }
```

Output:

```
 The elements in a Queue are : 34,10,23
The front element is 10

...Program finished with exit code 24
Press ENTER to exit console.
```

Deque

The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

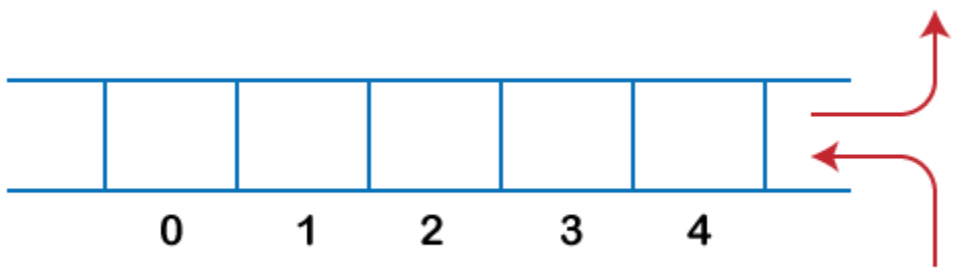


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

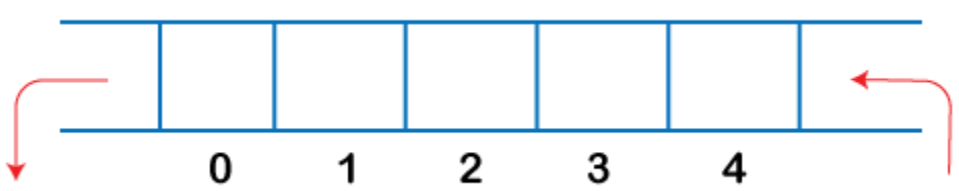
Let's look at some properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.

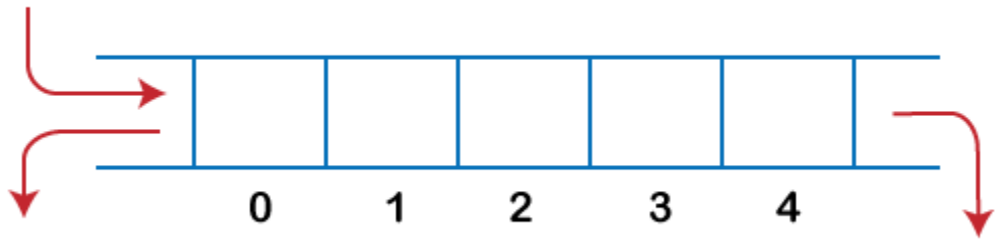


In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

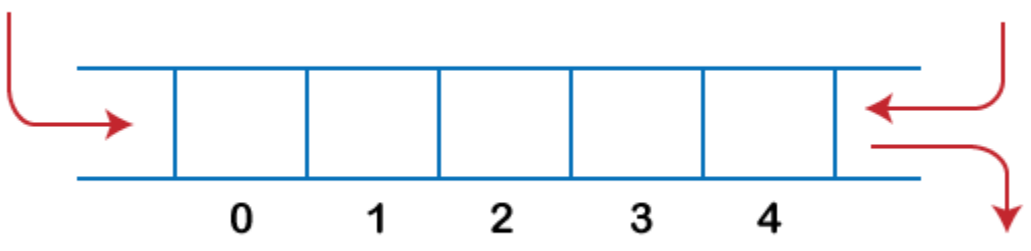


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Operations on Deque

The following are the operations applied on deque:

- **Insert at front**
- **Delete from end**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

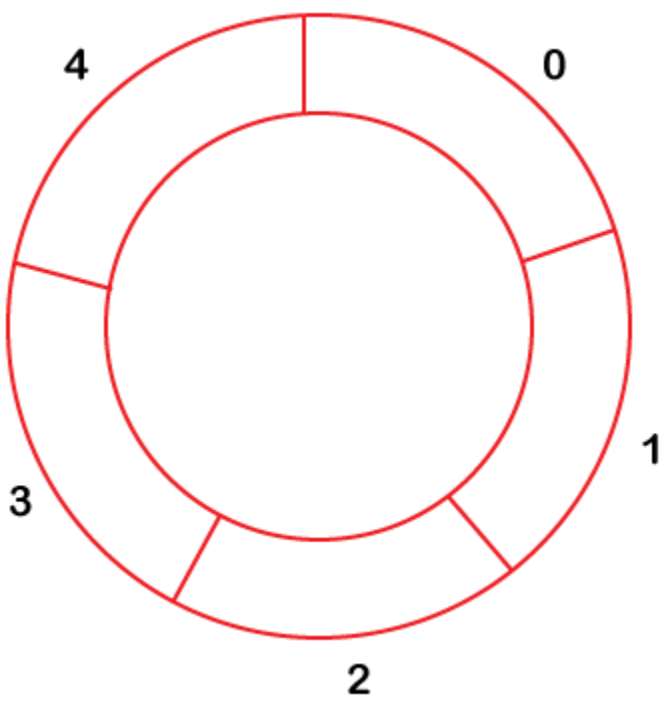
- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Memory Representation

The deque can be implemented using two data structures, i.e., **circular array**, and **doubly linked list**. To implement the deque using circular array, we first should know **what is circular array**.

What is a circular array?

An array is said to be **circular** if the last element of the array is connected to the first element of the array. Suppose the size of the array is 4, and the array is full but the first location of the array is empty. If we want to insert the array element, it will not show any overflow condition as the last element is connected to the first element. The value which we want to insert will be added in the first location of the array.



Applications of Deque

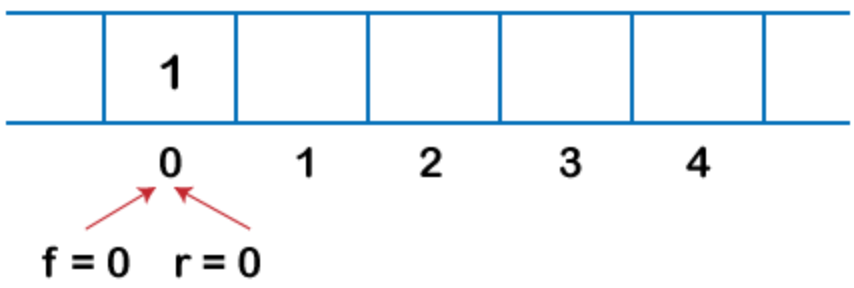
- The deque can be used as a **stack** and **queue**; therefore, it can perform both redo and undo operations.
- It can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- It can be used for multiprocessor scheduling. Suppose we have two processors, and each processor has one process to execute. Each processor is assigned with a process or a job, and each process contains multiple threads. Each processor maintains a deque that contains threads that are ready to execute. The processor executes a process, and if a process creates a child process then that process will be inserted at the front of the deque of the parent process. Suppose the processor P_2 has completed the execution of all its threads then it steals the thread from the rear end of the processor P_1 and adds to the front end of the processor P_2. The processor P_2 will take the thread from the front end; therefore, the deletion takes from both the ends, i.e., front and rear end. This is known as the **A-steal algorithm** for scheduling.

Implementation of Deque using a circular array

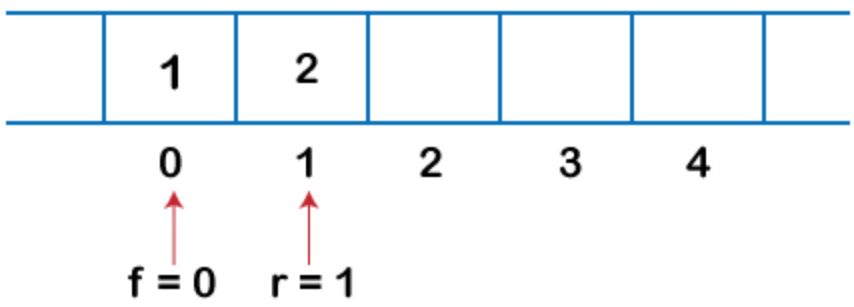
The following are the steps to perform the operations on the Deque:

Enqueue operation

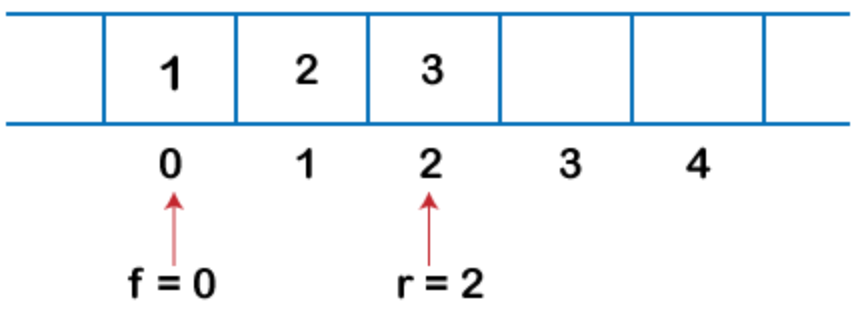
1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0,** and the **rear is also equal to 0.**



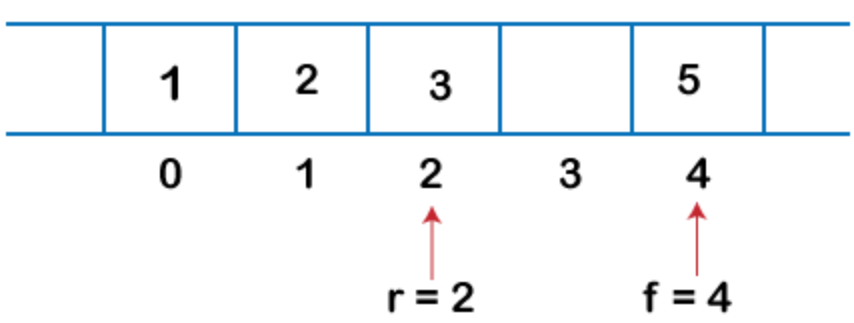
3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.



4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.



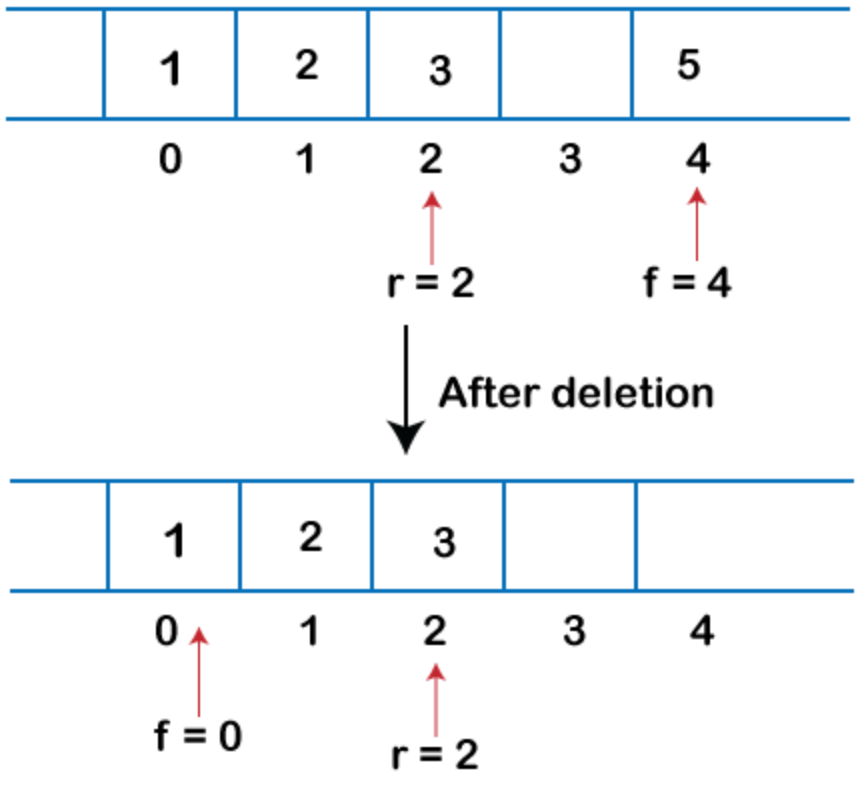
5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n -1),** which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:



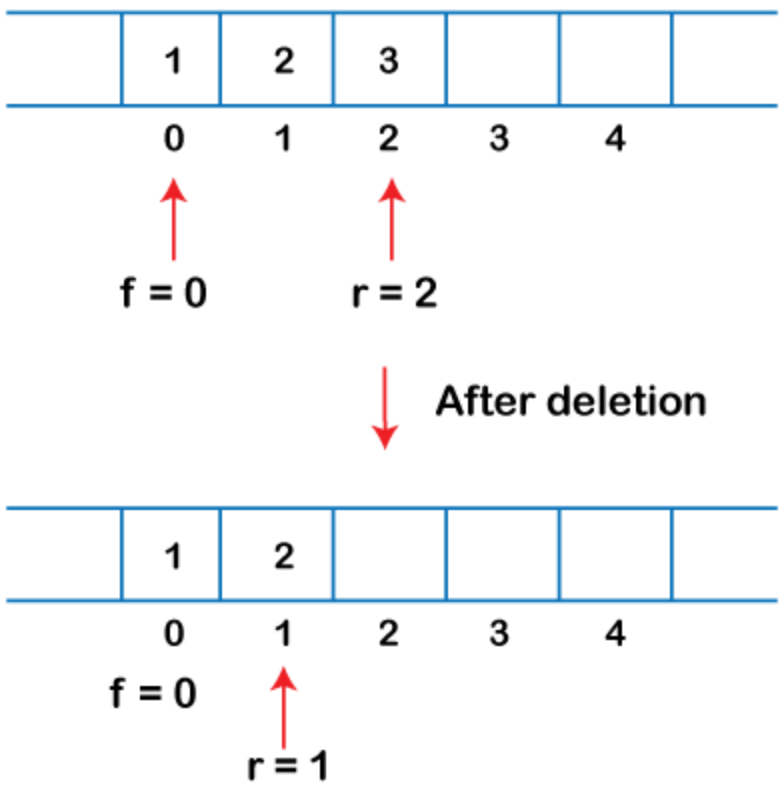
Dequeue Operation

1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front

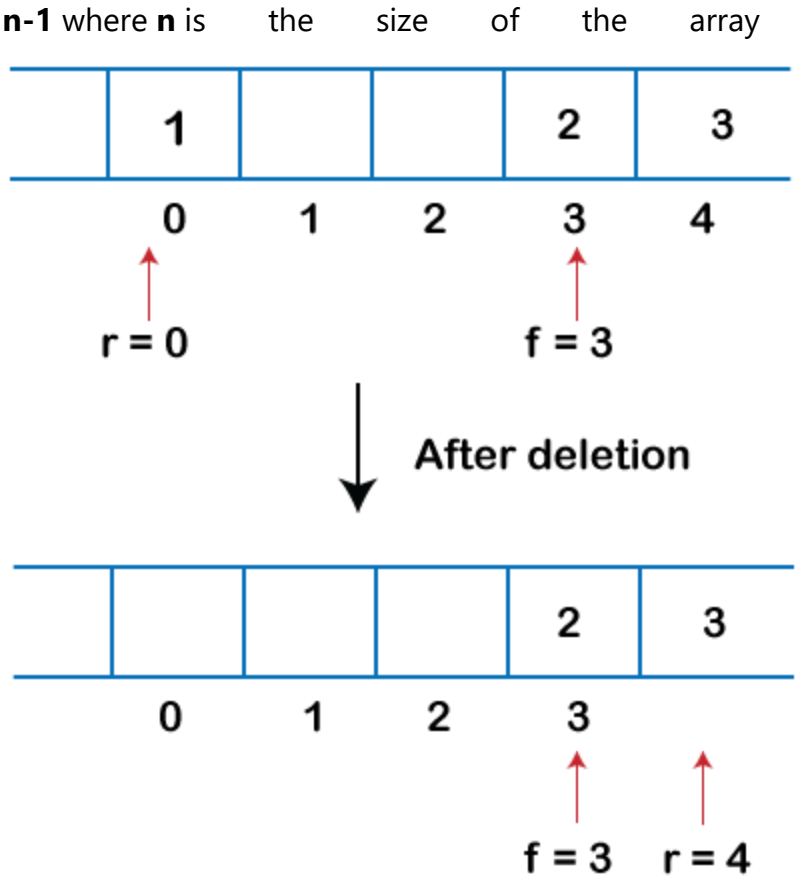
points to the last element, then front is set to 0 in case of delete operation.



2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the below figure:



3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below figure:



Let's create a program of deque.

The following are the six functions that we have used in the below program:

- **enqueue_front():** It is used to insert the element from the front end.
- **enqueue_rear():** It is used to insert the element from the rear end.
- **dequeue_front():** It is used to delete the element from the front end.
- **dequeue_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.

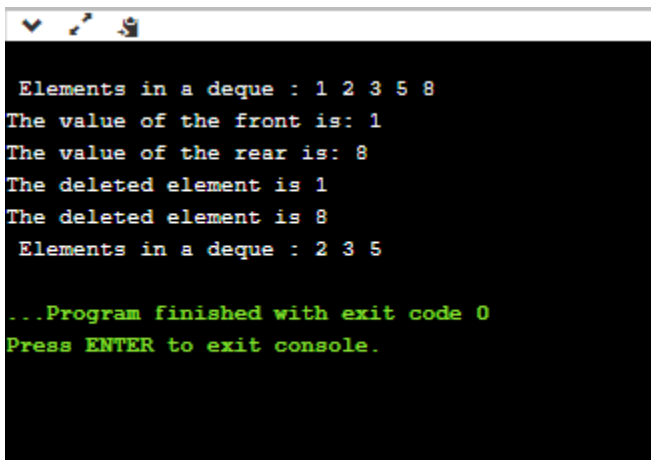
```
#define size 5
#include <stdio.h>
int deque[size];
int f=-1, r=-1;
//  enqueue_front function will insert the value from the front
void enqueue_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}

// enqueue_rear function will insert the value from the rear
void enqueue_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
```

```
49.         deque[r]=x;
50.     }
51.
52. }
53.
54. // display function prints all the value of deque.
55. void display()
56. {
57.     int i=f;
58.     printf("\n Elements in a deque : ");
59.
60.     while(i!=r)
61.     {
62.         printf("%d ",deque[i]);
63.         i=(i+1)%size;
64.     }
65.      printf("%d",deque[r]);
66. }
67.
68. // getfront function retrieves the first value of the deque.
69. void getfront()
70. {
71.     if((f==-1) && (r==-1))
72.     {
73.         printf("Deque is empty");
74.     }
75.     else
76.     {
77.         printf("\nThe value of the front is: %d", deque[f]);
78.     }
79.
80. }
81.
82. // getrear function retrieves the last value of the deque.
83. void getrear()
84. {
85.     if((f==-1) && (r==-1))
86.     {
87.         printf("Deque is empty");
88.     }
89.     else
90.     {
91.         printf("\nThe value of the rear is: %d", deque[r]);
92.     }
93.
94. }
95.
96. // dequeue_front() function deletes the element from the front
97. void dequeue_front()
98. {
99.     if((f==-1) && (r==-1))
100.             {
101.                 printf("Deque is empty");
102.             }
103.             else if(f==r)
104.             {
105.                 printf("\nThe deleted element is %d", deque[f]);
```

```
106.              f=-1;
107.              r=-1;
108.
109.           }
110.            else if(f==(size-1))
111.            {
112.                printf("\nThe deleted element is %d", deque[f]);
113.                f=0;
114.            }
115.            else
116.            {
117.                 printf("\nThe deleted element is %d", deque[f]);
118.                 f=f+1;
119.            }
120.        }
121.
122.        // dequeue_rear() function deletes the element from the rear
123.        void dequeue_rear()
124.        {
125.           if((f==-1) && (r==-1))
126.           {
127.               printf("Deque is empty");
128.           }
129.           else if(f==r)
130.           {
131.               printf("\nThe deleted element is %d", deque[r]);
132.               f=-1;
133.               r=-1;
134.
135.           }
136.            else if(r==0)
137.            {
138.                printf("\nThe deleted element is %d", deque[r]);
139.                r=size-1;
140.            }
141.            else
142.            {
143.                 printf("\nThe deleted element is %d", deque[r]);
144.                 r=r-1;
145.            }
146.        }
147.
148.        int main()
149.        {
150.           // inserting a value from the front.
151.             enqueue_front(2);
152.          // inserting a value from the front.
153.             enqueue_front(1);
154.          // inserting a value from the rear.
155.             enqueue_rear(3);
156.          // inserting a value from the rear.
157.             enqueue_rear(5);
158.          // inserting a value from the rear.
159.             enqueue_rear(8);
160.          // Calling the display function to retrieve the values of deque
161.             display();
162.         // Retrieve the front value
```

```
163.            getfront();
164.        // Retrieve the rear value.
165.            getrear();
166.        // deleting a value from the front
167.        dequeue_front();
168.        //deleting a value from the rear
169.        dequeue_rear();
170.         // Calling the display function to retrieve the values of deque
171.         display();
172.            return 0;
173.        }
```

Output:



What is a priority queue?

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

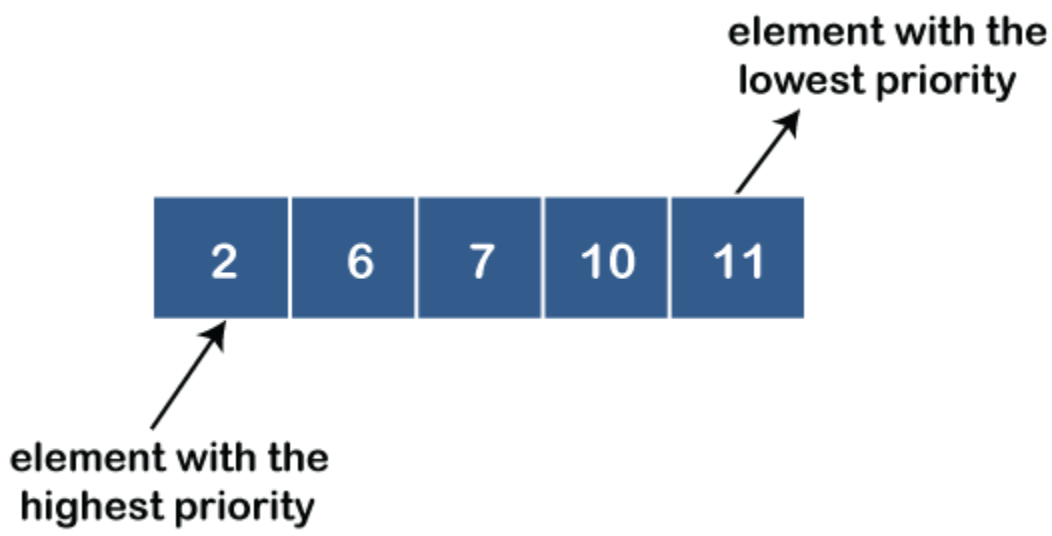
- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.

- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

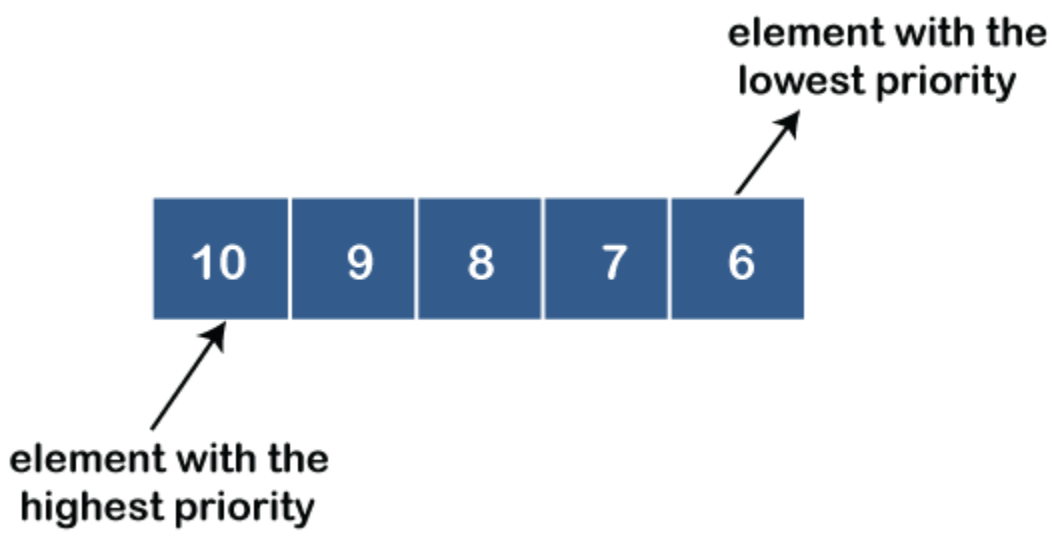
Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.

Let's create the priority queue step by step.

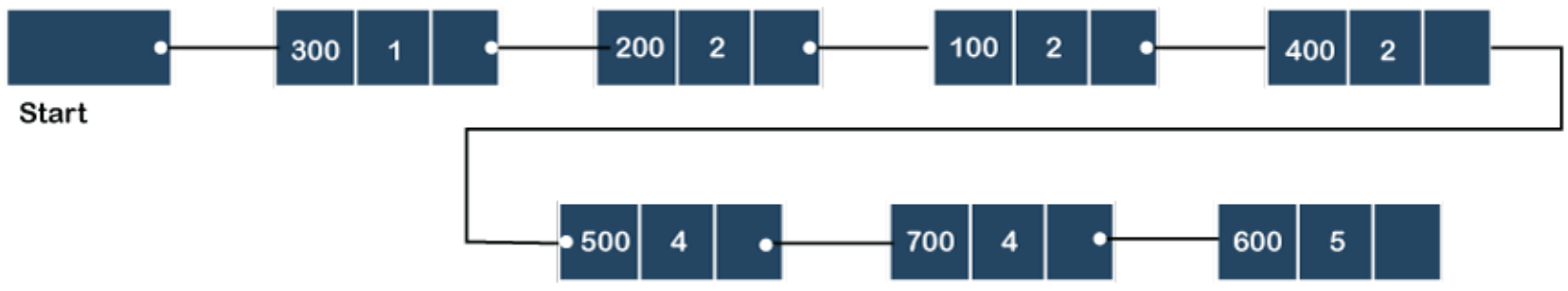
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

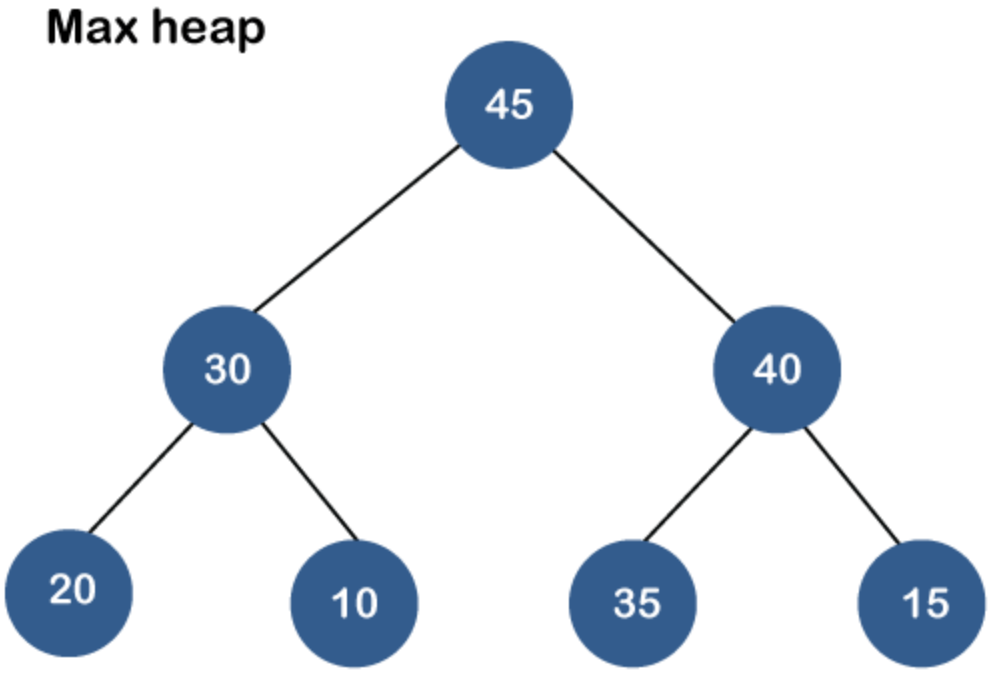
Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

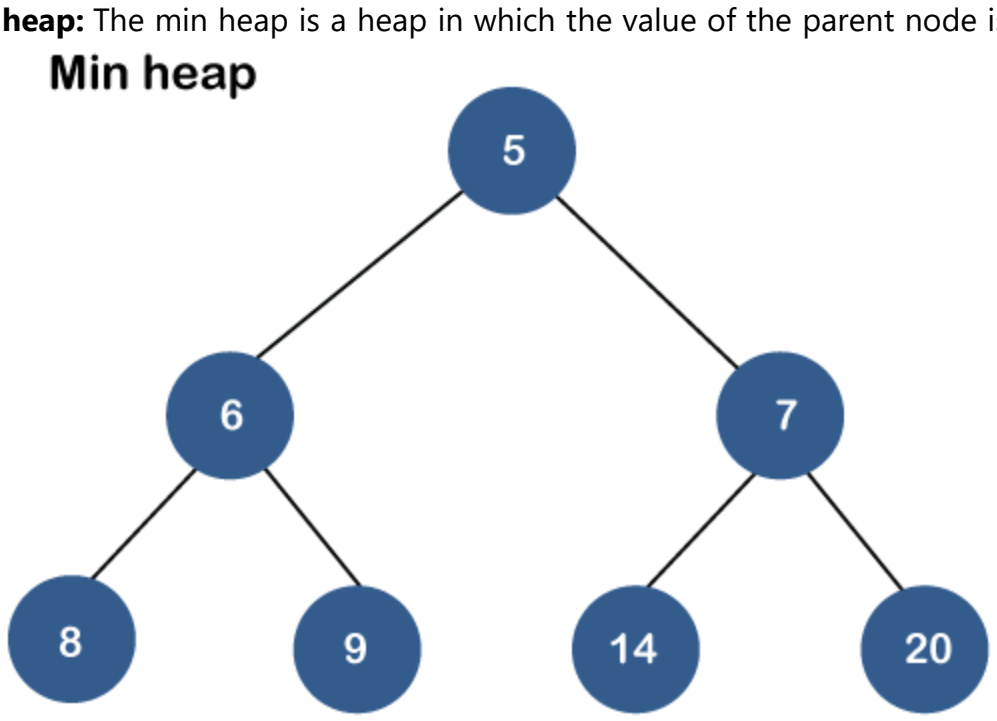
What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

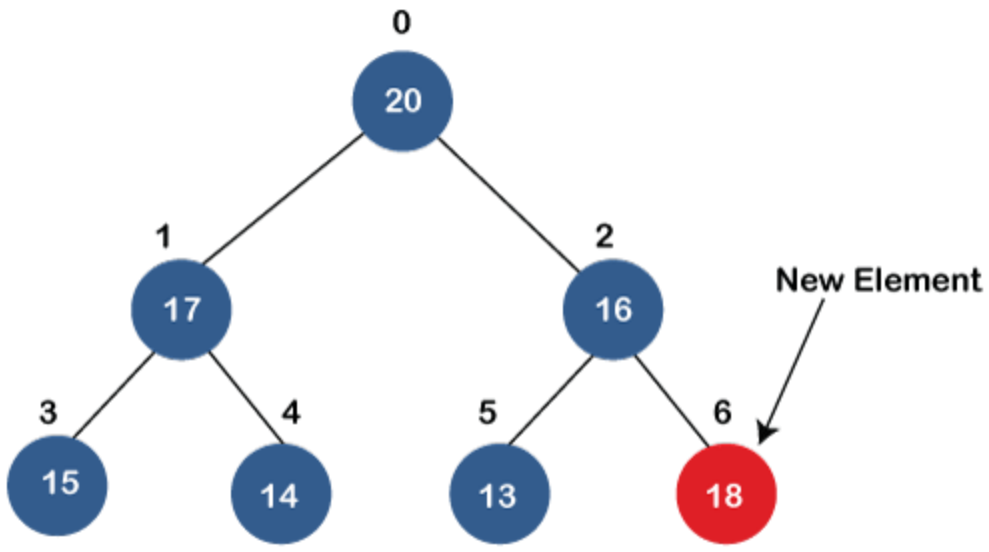
Priority Queue Operations

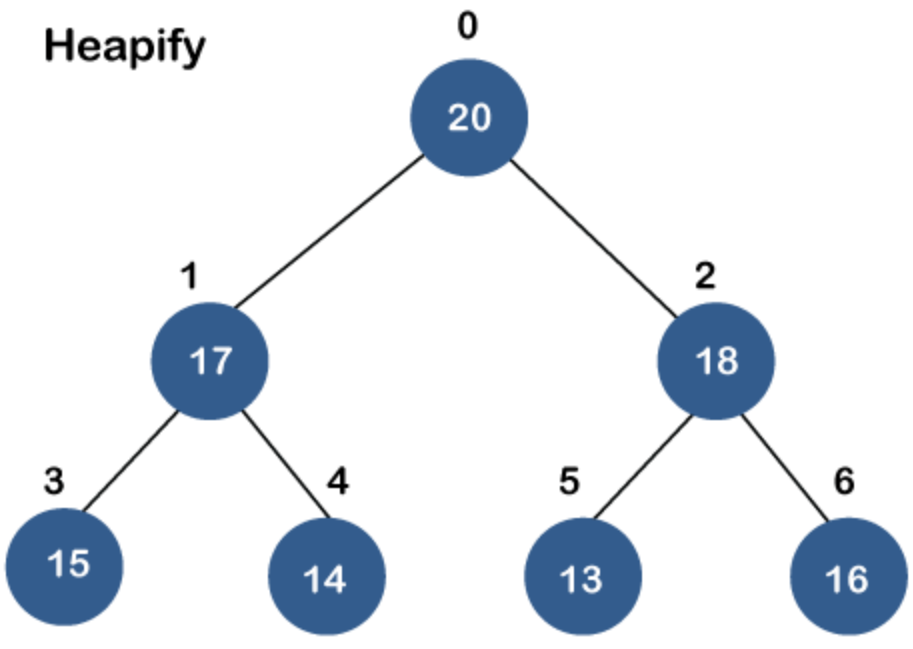
The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

- **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.





- **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Program to create the priority queue using the binary max heap.

```
#include <stdio.h>
#include <stdio.h>
int heap[40];
int size=-1;

// retrieving the parent node of the child node
int parent(int i)
{

    return (i - 1) / 2;
}

// retrieving the left child of the parent node.
```

```
14. int left_child(int i)
15. {
16.  return i+1;
17. }
18. // retrieving the right child of the parent
19. int right_child(int i)
20. {
21.  return i+2;
22. }
23. // Returning the element having the highest priority
24. int get_Max()
25. {
26.     return heap[0];
27. }
28. //Returning the element having the minimum priority
29. int get_Min()
30. {
31.     return heap[size];
32. }
33. // function to move the node up the tree in order to restore the heap property.
34. void moveUp(int i)
35. {
36.    while (i > 0)
37.    {
38.        // swapping parent node with a child node
39.        if(heap[parent(i)] < heap[i]) {
40.
41.        int temp;
42.        temp=heap[parent(i)];
43.        heap[parent(i)]=heap[i];
44.        heap[i]=temp;
45.
46.
47.    }
48.       // updating the value of i to i/2
49.       i=i/2;
50.    }
51. }
52.
53. //function to move the node down the tree in order to restore the heap property.
54. void moveDown(int k)
55. {
56.    int index = k;
57.
58.    // getting the location of the Left Child
59.    int left = left_child(k);
60.
61.    if (left <= size && heap[left] > heap[index]) {
62.        index = left;
63.    }
64.
65.    // getting the location of the Right Child
66.    int right = right_child(k);
67.
68.    if (right <= size && heap[right] > heap[index]) {
69.        index = right;
70.    }
```

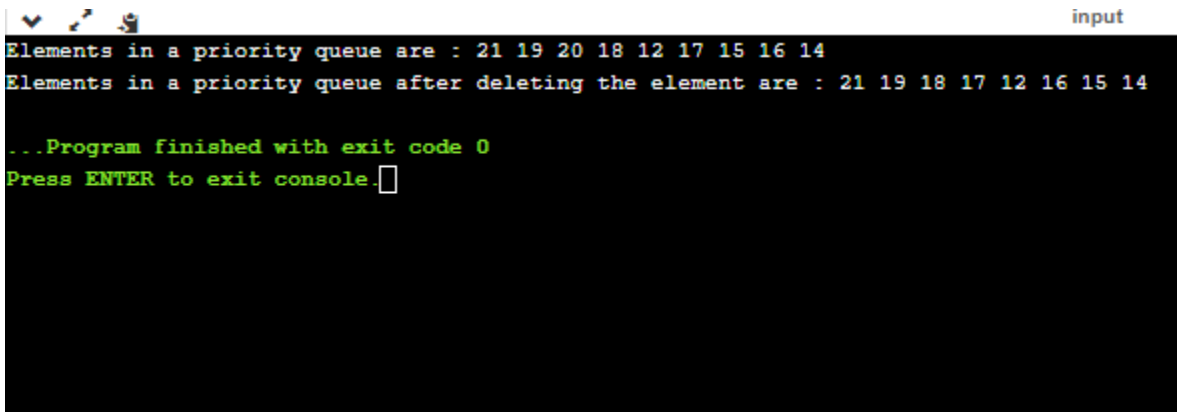
```
71.
72.     // If k is not equal to index
73.     if (k != index) {
74.       int temp;
75.       temp=heap[index];
76.       heap[index]=heap[k];
77.       heap[k]=temp;
78.         moveDown(index);
79.     }
80. }
81.
82. // Removing the element of maximum priority
83. void removeMax()
84. {
85.     int r= heap[0];
86.     heap[0]=heap[size];
87.     size=size-1;
88.     moveDown(0);
89. }
90. //inserting the element in a priority queue
91. void insert(int p)
92. {
93.     size = size + 1;
94.     heap[size] = p;
95.
96.     // move Up to maintain heap property
97.     moveUp(size);
98. }
99.
100.         //Removing the element from the priority queue at a given index i.
101.         void delete(int i)
102.         {
103.             heap[i] = heap[0] + 1;
104.
105.            // move the node stored at ith location is shifted to the root node
106.             moveUp(i);
107.
108.             // Removing the node having maximum priority
109.             removeMax();
110.         }
111.         int main()
112.         {
113.             // Inserting the elements in a priority queue
114.
115.             insert(20);
116.             insert(19);
117.             insert(21);
118.             insert(18);
119.             insert(12);
120.             insert(17);
121.             insert(15);
122.             insert(16);
123.             insert(14);
124.             int i=0;
125.
126.          printf("Elements in a priority queue are : ");
127.          for(int i=0;i<=size;i++)
```

```
128.            {
129.               printf("%d ",heap[i]);
130.            }
131.            delete(2); // deleting the element whose index is 2.
132.            printf("\nElements in a priority queue after deleting the element are : ");
133.            for(int i=0;i<=size;i++)
134.            {
135.               printf("%d ",heap[i]);
136.            }
137.         int max=get_Max();
138.            printf("\nThe element which is having the highest priority is %d: ",max);
139.
140.
141.            int min=get_Min();
142.               printf("\nThe element which is having the minimum priority is : %d",min);
143.            return 0;
144.         }
```

In the above program, we have created the following functions:

- **int parent(int i):** This function returns the index of the parent node of a child node, i.e., i.
- **int left_child(int i):** This function returns the index of the left child of a given index, i.e., i.
- **int right_child(int i):** This function returns the index of the right child of a given index, i.e., i.
- **void moveUp(int i):** This function will keep moving the node up the tree until the heap property is restored.
- **void moveDown(int i):** This function will keep moving the node down the tree until the heap property is restored.
- **void removeMax():** This function removes the element which is having the highest priority.
- **void insert(int p):** It inserts the element in a priority queue which is passed as an argument in a function**.**
- **void delete(int i):** It deletes the element from a priority queue at a given index.
- **int get_Max():** It returns the element which is having the highest priority, and we know that in max heap, the root node contains the element which has the largest value, and highest priority.
- **int get_Min():** It returns the element which is having the minimum priority, and we know that in max heap, the last node contains the element which has the smallest value, and lowest priority.

Output



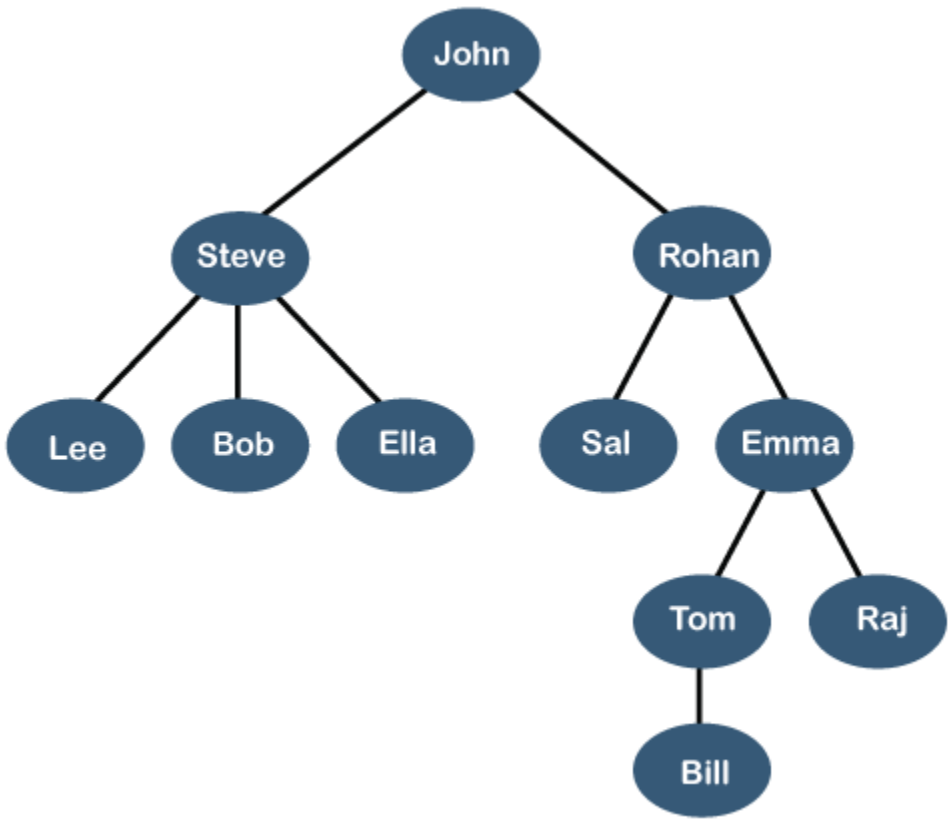
Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

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



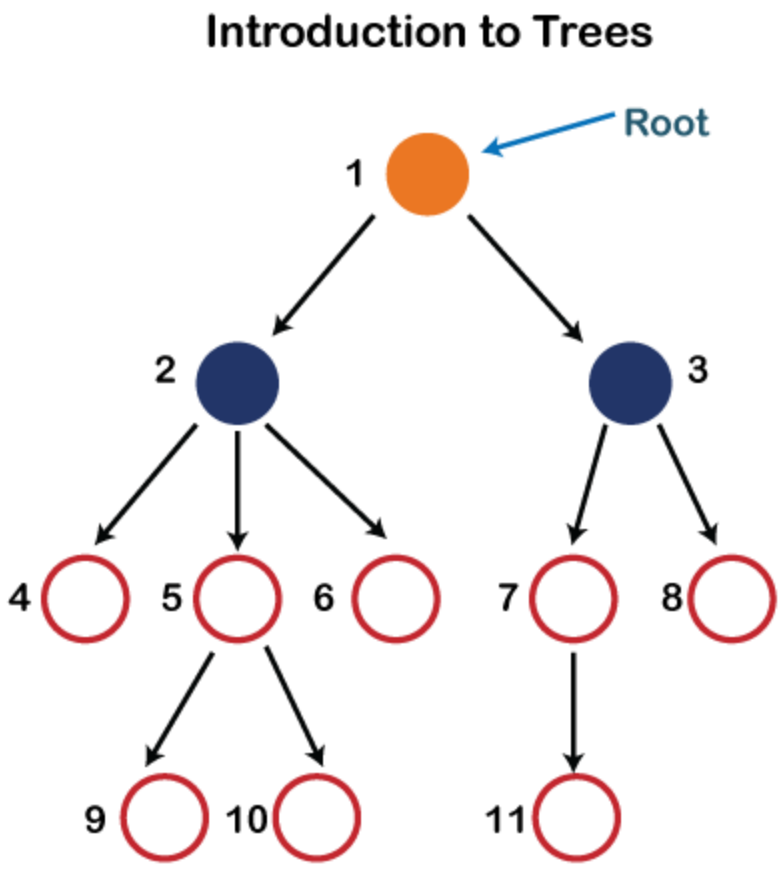
The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has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

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

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

Some basic terms used in Tree data structure.

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



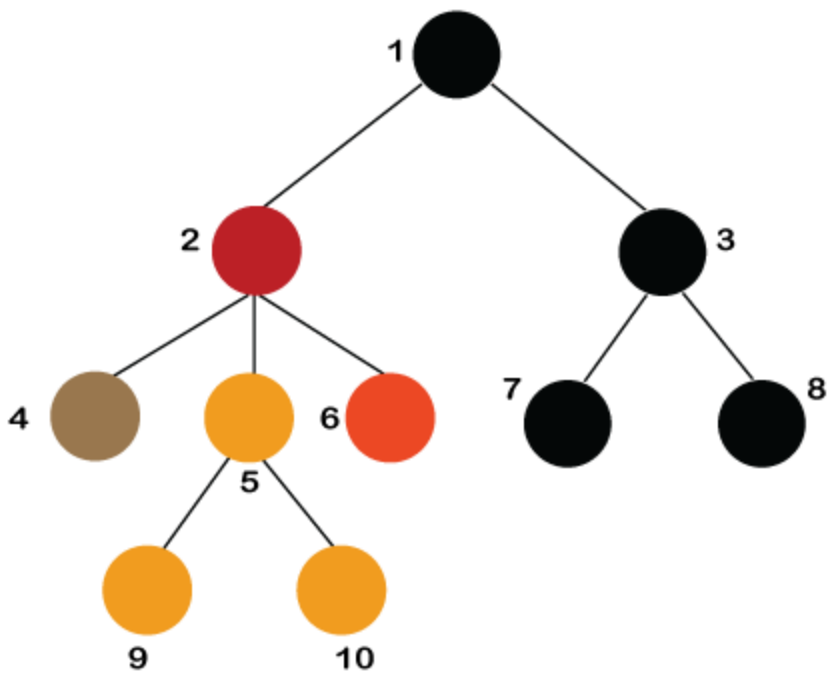
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

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

implemented in various applications.

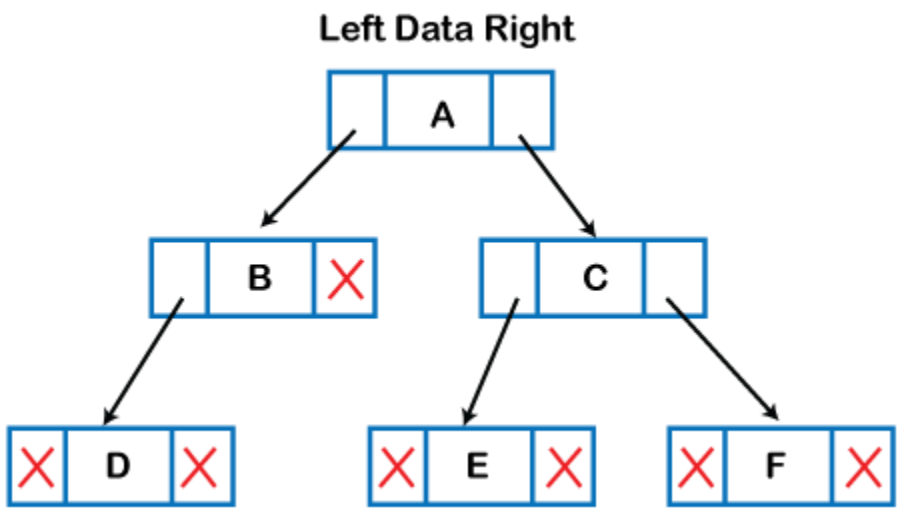


- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The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

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

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

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

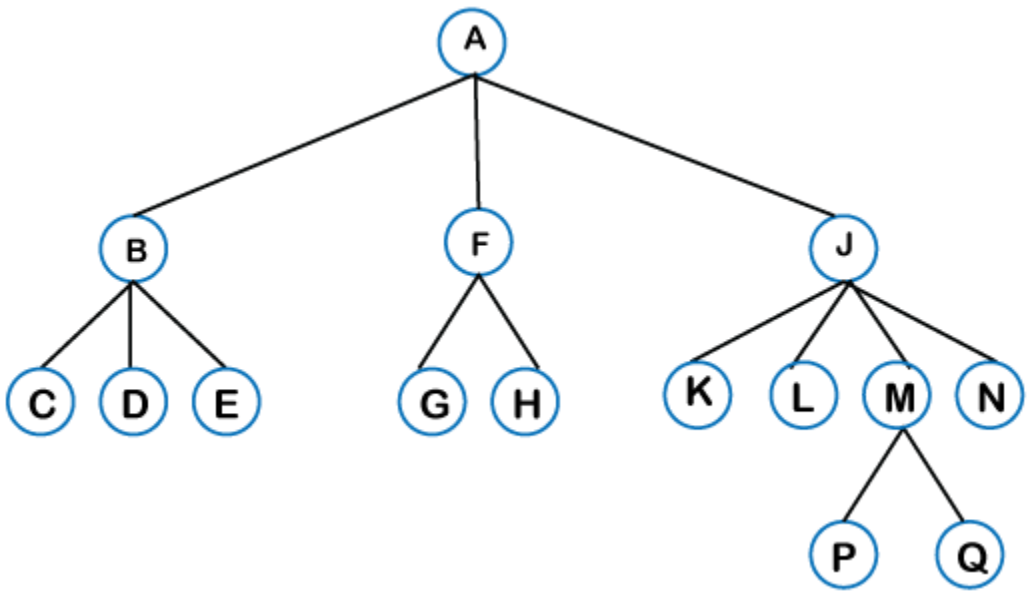
The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

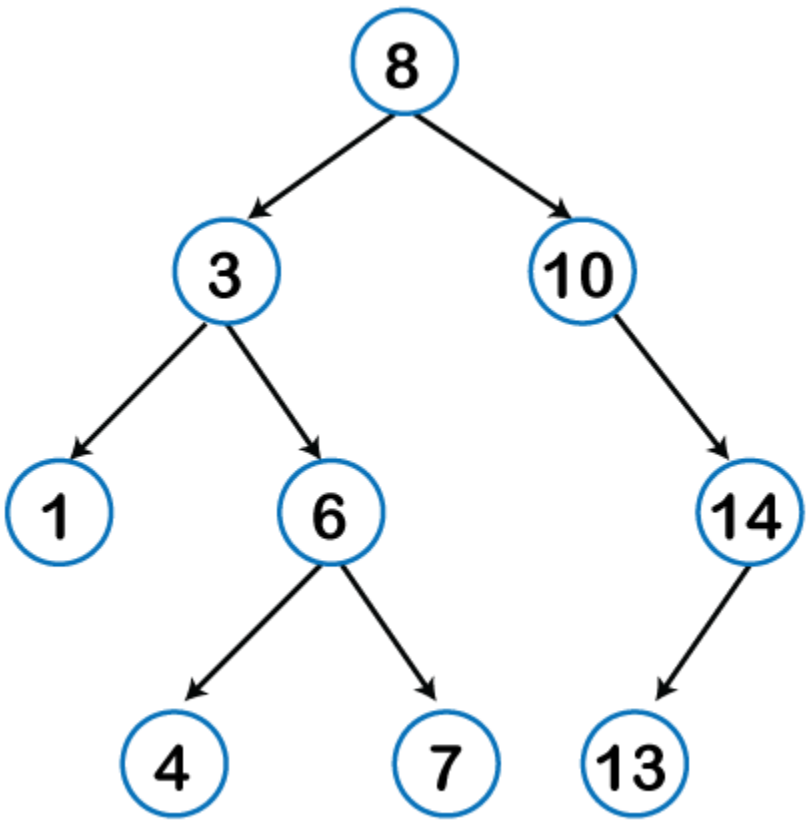
The following are the types of a tree data structure:

- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

- **Binary tree:** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



To know more about the binary tree, click on the link given below: https://www.javatpoint.com/binary-tree

- **Binary Search tree:** Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer). Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

A node can be created with the help of a user-defined data type known as ***struct,*** as shown below:

```
1. struct node
2. {
3.     int data;
4.     struct node *left;
5. struct node *right;
6. }
```

The above is the node structure with three fields: data field, the second field is the left pointer of the node type, and the third field is the right pointer of the node type.

To know more about the binary search tree, click on the link given below:

https://www.javatpoint.com/binary-search-tree

- **AVL tree**

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

To know more about the AVL tree, click on the link given below:

https://www.javatpoint.com/avl-tree

- **Red-Black Tree**

The red-Black tree is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node. As we know that the time complexity of binary search in the average case is $\log_2 n$, the best case is O(1), and the worst case is O(n).

When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of $\boldsymbol{\log_2 n}$.

The red-black tree is a self-balancing binary search tree. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

- **Splay tree**

The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, ***splaying*** means the recently accessed node. It is a ***self-balancing*** binary search tree having no explicit balance condition like **AVL** tree.

It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of **logN** time where **n** is the number of nodes.

Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.

- **Treap**

Treap data structure came from the Tree and Heap data structure. So, it comprises the properties of both Tree and Heap data structures. In Binary search tree, each node on the left subtree must be equal or less than the value of the root node and each node on the right subtree must be equal or greater than the value of the root node. In heap data structure, both right and left subtrees contain larger keys than the root; therefore, we can say that the root node contains the lowest value.

In treap data structure, each node has both ***key*** and ***priority*** where key is derived from the Binary search tree and priority is derived from the heap data structure.

The **Treap** data structure follows two properties which are given below:

- Right child of a node>=current node and left child of a node <=current node (binary tree)
- Children of any subtree must be greater than the node (heap)

- **B-tree**

B-tree is a balanced **m-way** tree where **m** defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children. It always maintains the sorted data. In binary tree, it is possible that leaf nodes can be at different levels, but in b-tree, all the leaf nodes must be at the same level.

If order is m then node has the following properties:

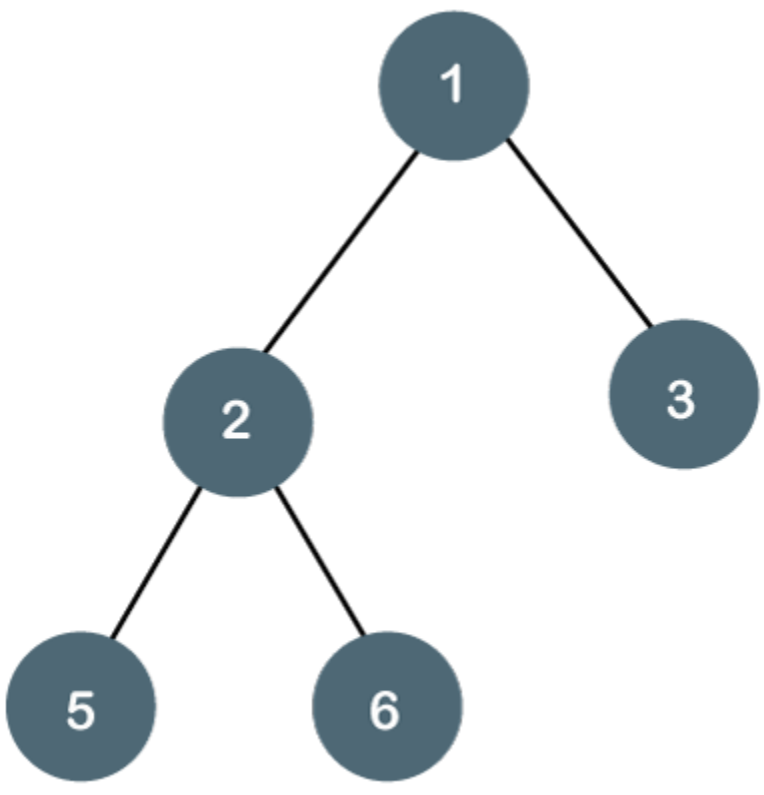
- Each node in a b-tree can have maximum **m** children
- For minimum children, a leaf node has 0 children, root node has minimum 2 children and internal node has minimum ceiling of m/2 children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain maximum 3 children.
- Each node has maximum (m-1) keys.

The root node must contain minimum 1 key and all other nodes must contain atleast **ceiling of m/2 minus 1** keys.

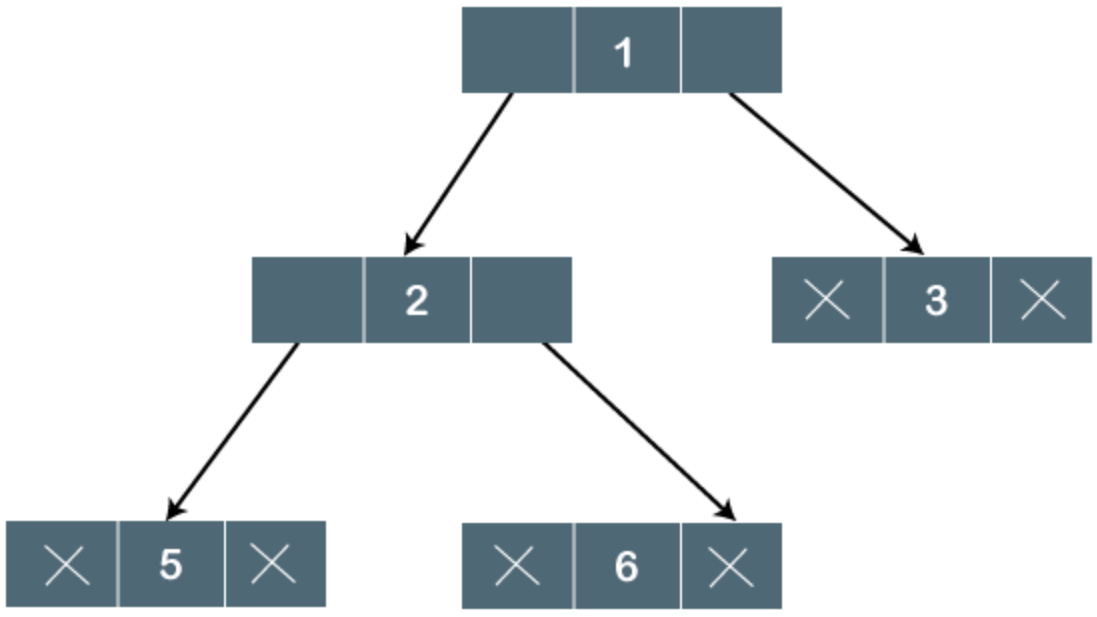
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + ...2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

$n = h+1$

$h = n-1$

Types of Binary Tree

There are four types of Binary tree:

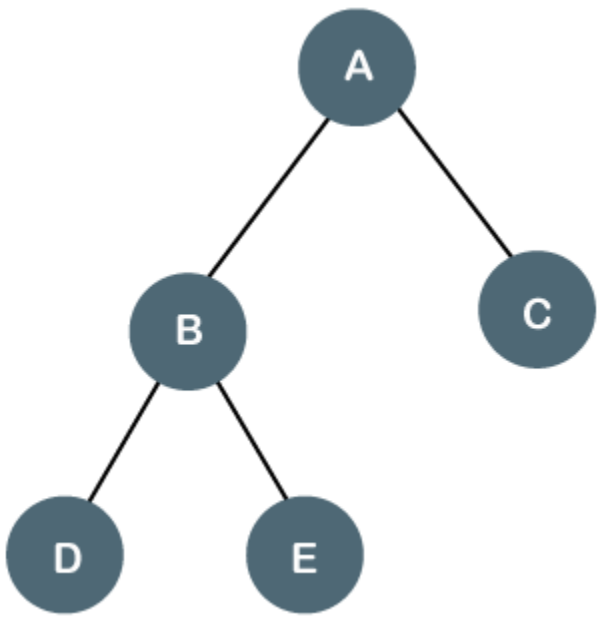
- **Full/ proper/ strict Binary tree**
- **Complete Binary tree**

- **Perfect Binary tree**
- **Degenerate Binary tree**
- **Balanced Binary tree**

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as:

n= 2*h - 1

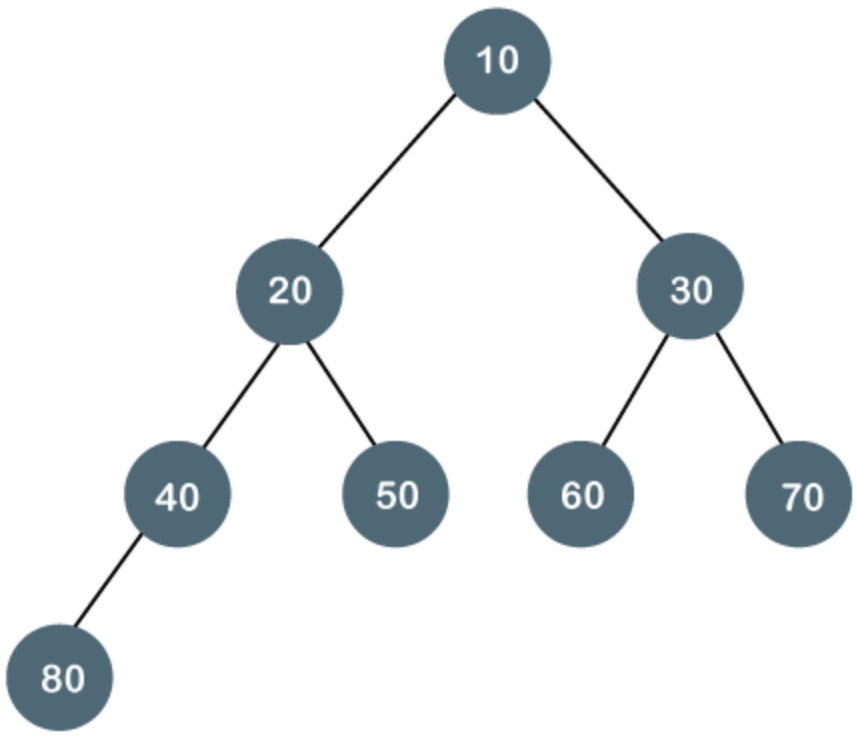
n+1 = 2*h

h = n+1/2

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



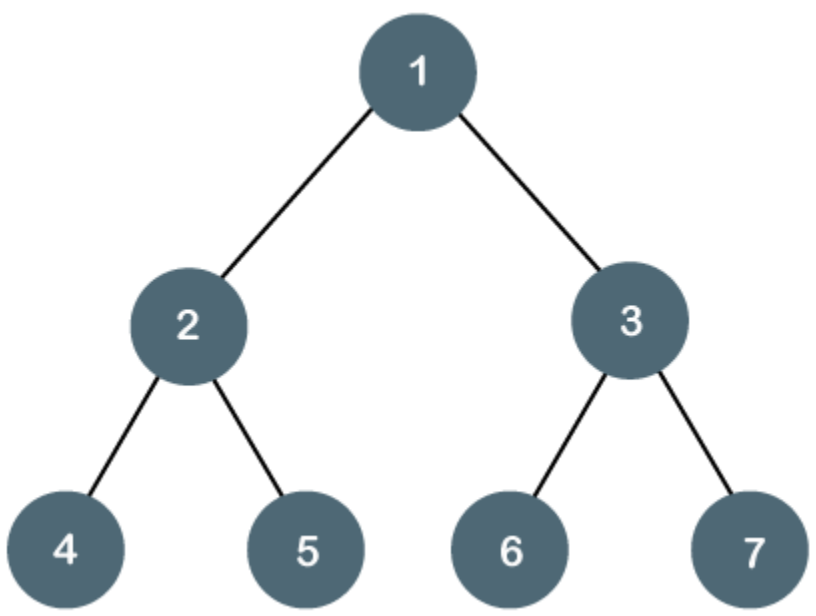
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

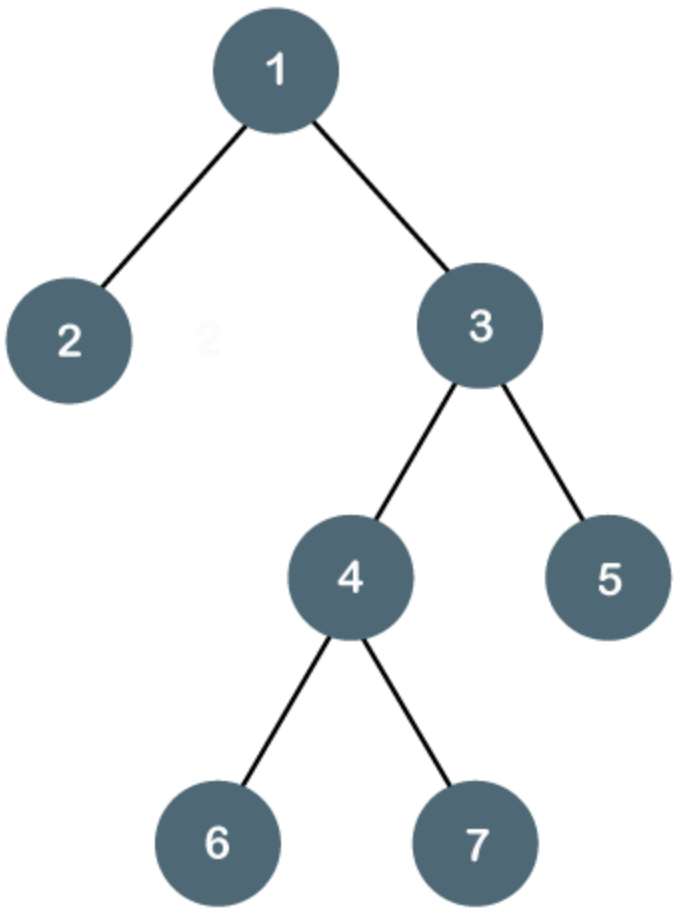
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

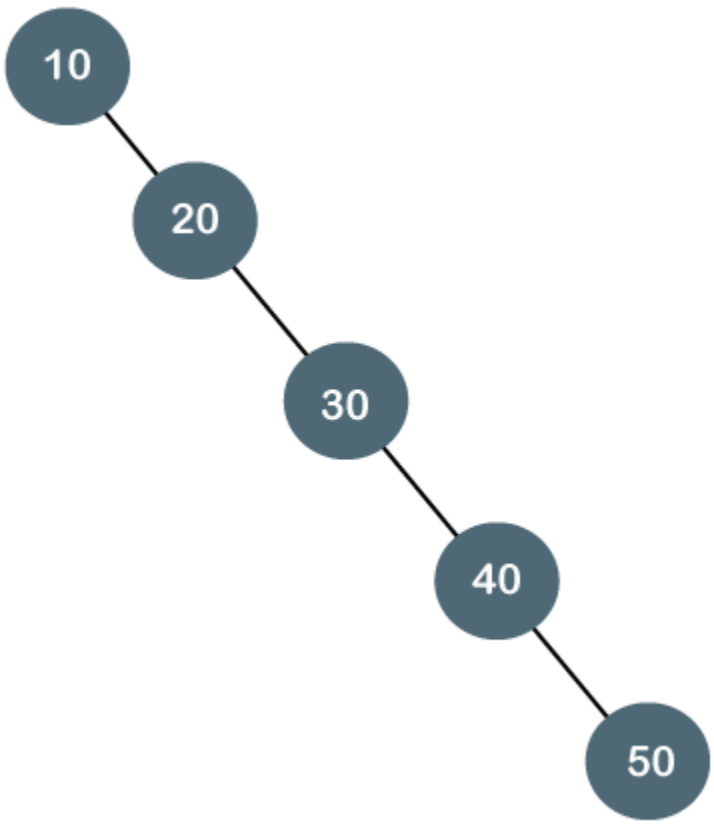


> ***Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.***

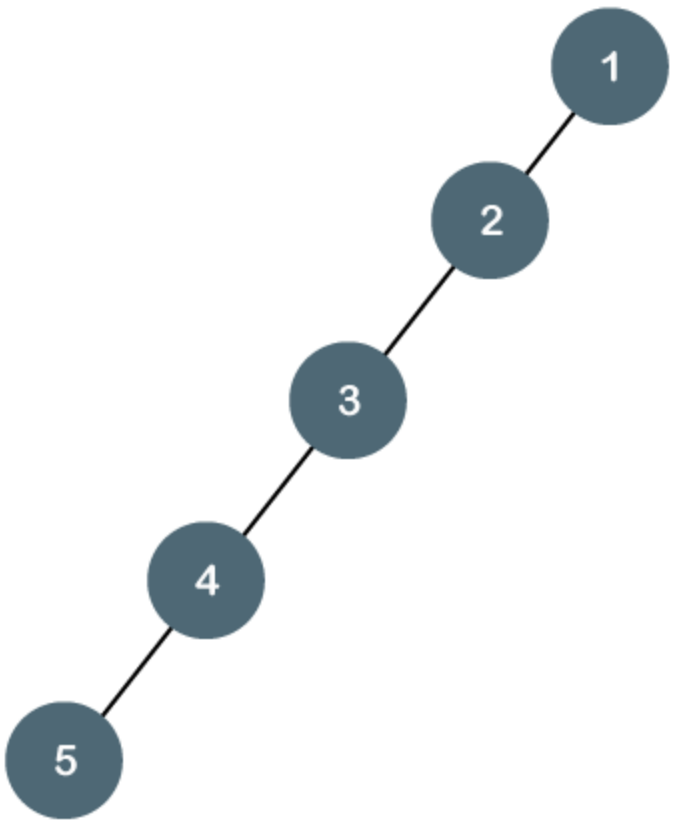
Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

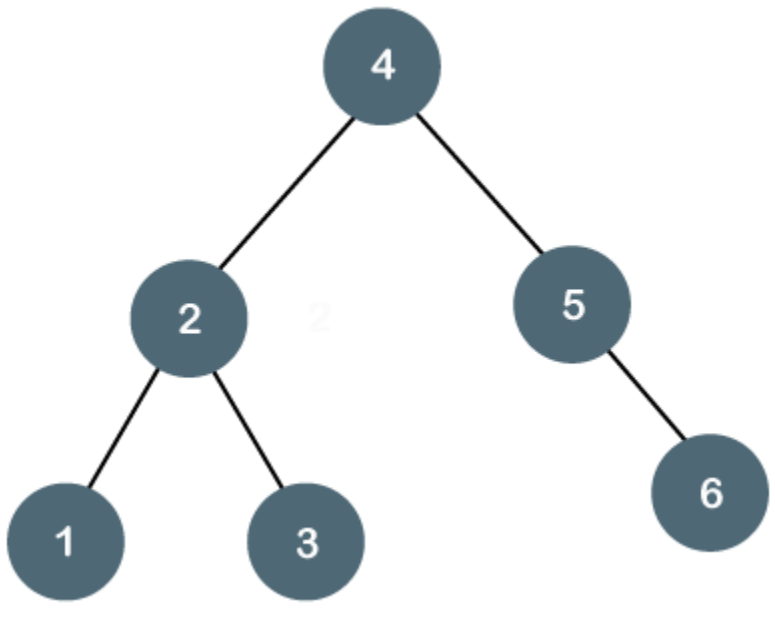


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

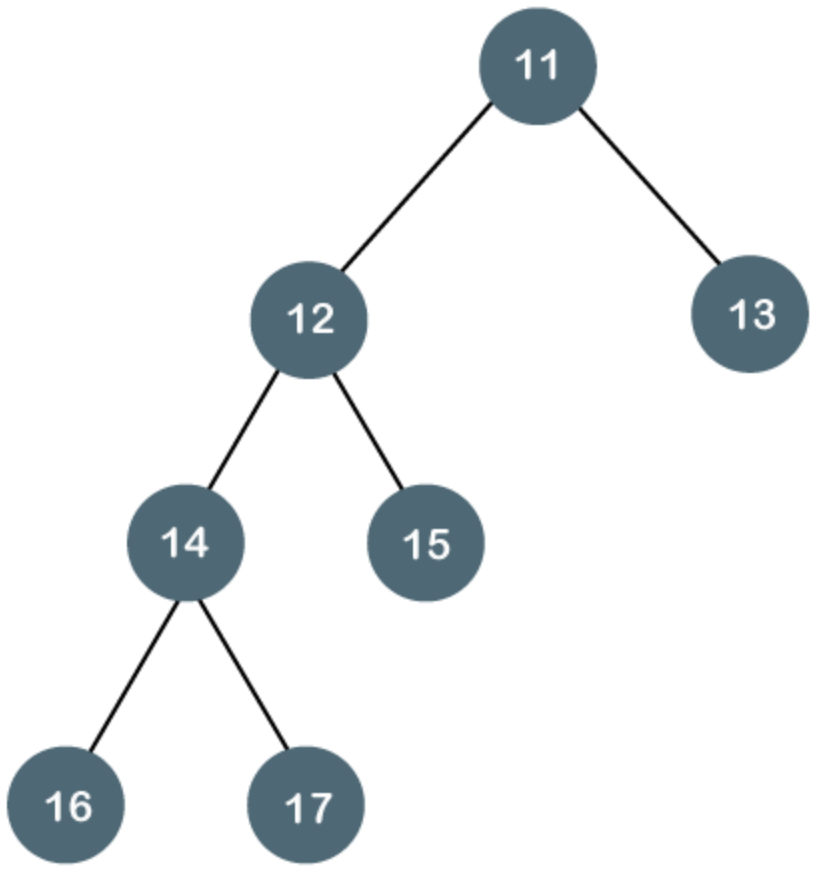
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
1. struct node
2. {
3.    int data,
4.    struct node *left, *right;
5. }
```

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Binary Tree program in C

```
1. #include<stdio.h>
2.       struct node
3.      {
4.          int data;
5.           struct node *left, *right;
6.       }
7.       void main()
8.     {
9.        struct node *root;
10.      root = create();
11.   }
12. struct node *create()
13. {
14.   struct node *temp;
15.   int data;
16.   temp = (struct node *)malloc(sizeof(struct node));
17.   printf("Press 0 to exit");
18.   printf("\nPress 1 for new node");
19.   printf("Enter your choice : ");
20.   scanf("%d", &choice);
21.   if(choice==0)
```

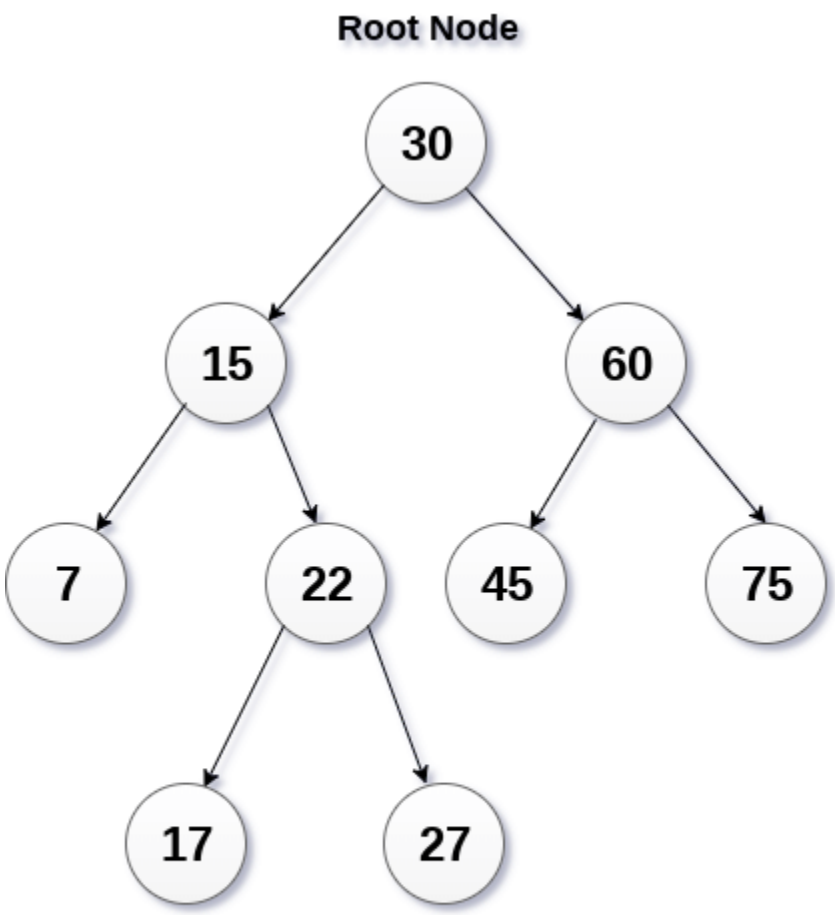
```
22. {
23. return 0;
24. }
25. else
26. {
27.    printf("Enter the data:");
28.    scanf("%d", &data);
29.    temp->data = data;
30.    printf("Enter the left child of %d", data);
31.    temp->left = create();
32. printf("Enter the right child of %d", data);
33. temp->right = create();
34. return temp;
35. }
36. }
```

The above code is calling the create() function recursively and creating new node on each recursive call. When all the nodes are created, then it forms a binary tree structure. The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

- Inorder traversal
- Preorder traversal
- Postorder traversal

Binary Search Tree

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

Advantages of using binary search tree

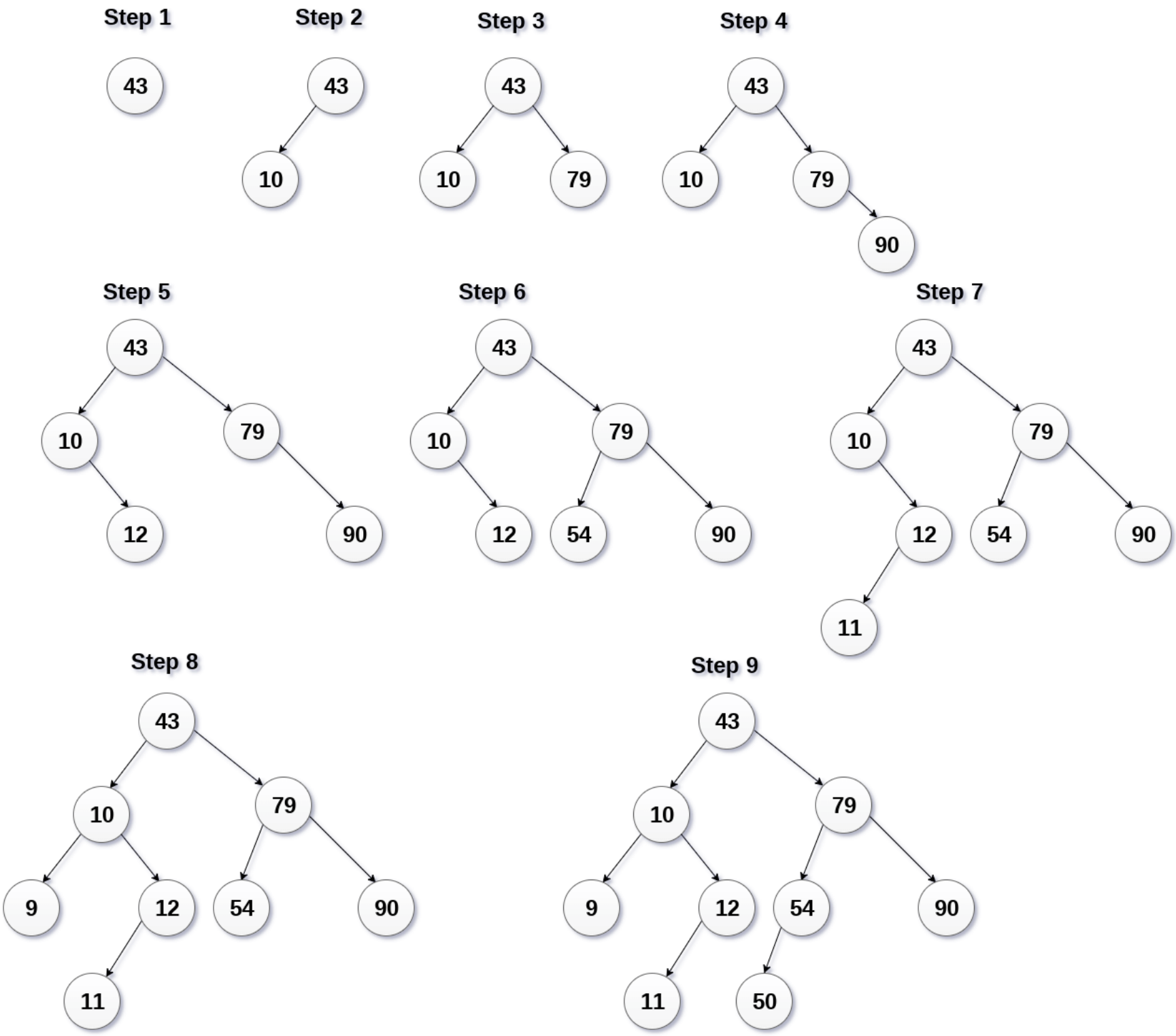
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

Q. Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Program to implement BST operations

```
1. #include <iostream>
2. #include <stdlib.h>
3. using namespace std;
```

```
4.  struct Node {
5.      int data;
6.      Node *left;
7.      Node *right;
8.  };
9.  Node* create(int item)
10. {
11.     Node* node = new Node;
12.     node->data = item;
13.     node->left = node->right = NULL;
14.     return node;
15. }
16.
17. void inorder(Node *root)
18. {
19.     if (root == NULL)
20.         return;
21.
22.     inorder(root->left);
23.     cout<< root->data << "   ";
24.     inorder(root->right);
25. }
26. Node* findMinimum(Node* cur)
27. {
28.     while(cur->left != NULL) {
29.         cur = cur->left;
30.     }
31.     return cur;
32. }
33. Node* insertion(Node* root, int item)
34. {
35.     if (root == NULL)
36.         return create(item);
37.     if (item < root->data)
38.         root->left = insertion(root->left, item);
39.     else
40.         root->right = insertion(root->right, item);
41.
42.     return root;
43. }
44.
45. void search(Node* &cur, int item, Node* &parent)
46. {
47.     while (cur != NULL && cur->data != item)
48.     {
49.         parent = cur;
50.
51.         if (item < cur->data)
52.             cur = cur->left;
53.         else
54.             cur = cur->right;
55.     }
56. }
57.
58. void deletion(Node*& root, int item)
59. {
60.     Node* parent = NULL;
```

```
    Node* cur = root;

    search(cur, item, parent);
    if (cur == NULL)
        return;

    if (cur->left == NULL && cur->right == NULL)
    {
        if (cur != root)
        {
            if (parent->left == cur)
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        else
            root = NULL;

        free(cur);
    }
    else if (cur->left && cur->right)
    {
        Node* succ  = findMinimum(cur- >right);

        int val = succ->data;

        deletion(root, succ->data);

        cur->data = val;
    }

    else
    {
        Node* child = (cur->left)? Cur- >left: cur->right;

        if (cur != root)
        {
            if (cur == parent->left)
                parent->left = child;
                    else
                        parent->right = child;
                }

                else
                    root = child;
                free(cur);
            }
        }

        int main()
        {
            Node* root = NULL;
            int keys[8];
            for(int i=0;i<8;i++)
             {
             cout << "Enter value to be inserted";
             cin>>keys[i];
```

```
118.                root = insertion(root, keys[i]);
119.            }
120.
121.            inorder(root);
122.            cout<<"\n";
123.            deletion(root, 10);
124.            inorder(root);
125.            return 0;
126.        }
```

Output:

```
Enter value to be inserted? 10
Enter value to be inserted? 20
Enter value to be inserted? 30
Enter value to be inserted? 40
Enter value to be inserted?  5
Enter value to be inserted? 25
Enter value to be inserted? 15
Enter value to be inserted?  5

5       5       10      15      20      25      30      40
5       5       15      20      25      30      40
```

AVL Tree

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

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

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

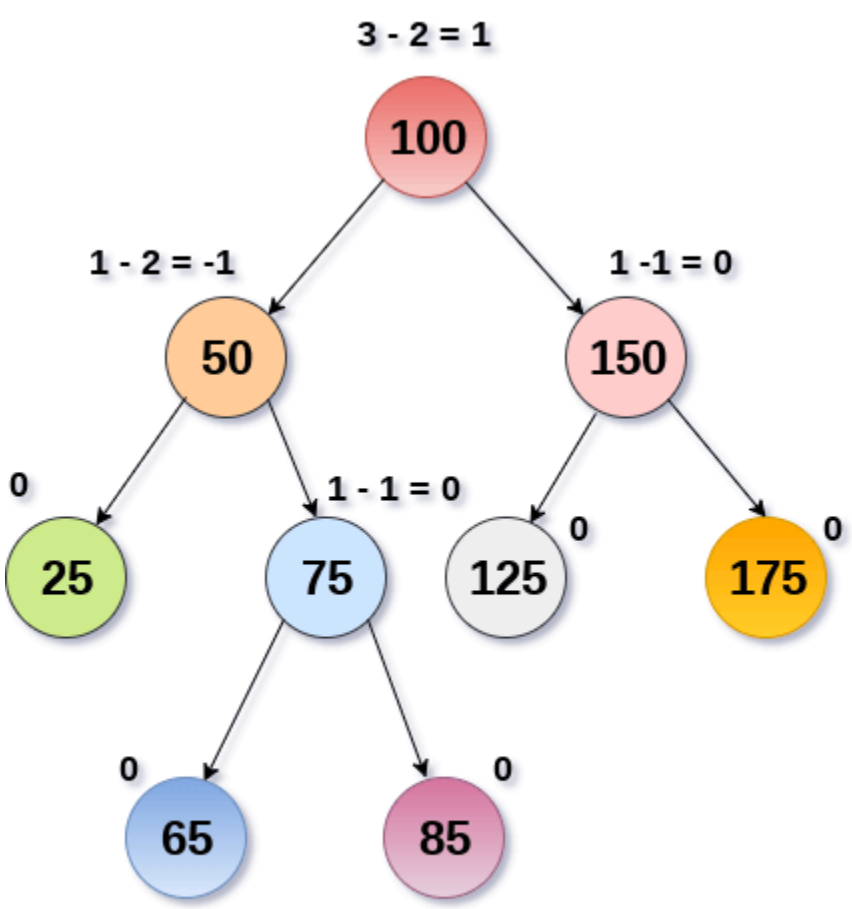
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

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



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

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

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

Why AVL Tree?

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

AVL Rotations

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

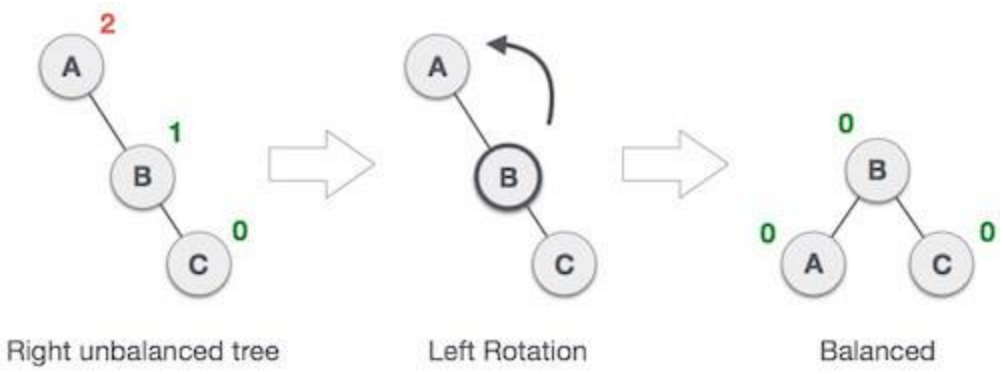
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree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

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

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

1. RR Rotation

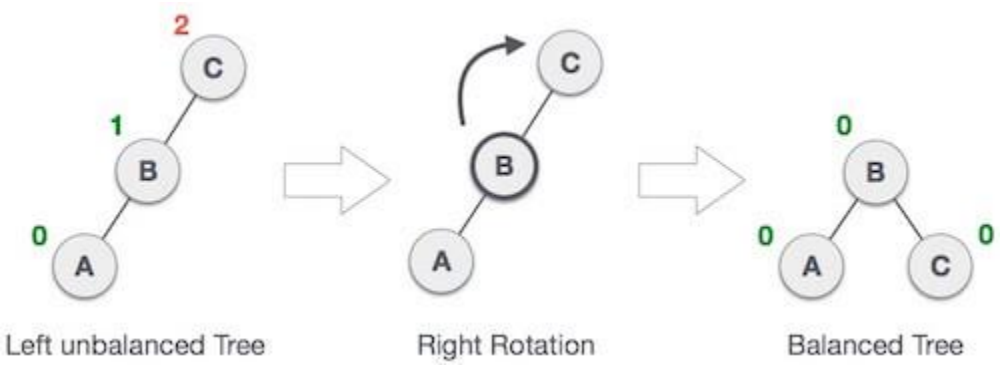
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



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

2. LL Rotation

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

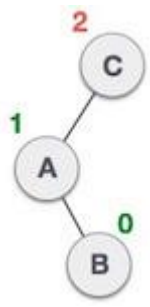
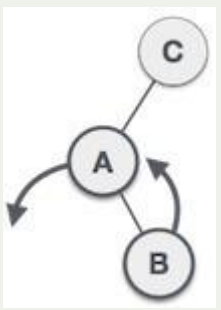
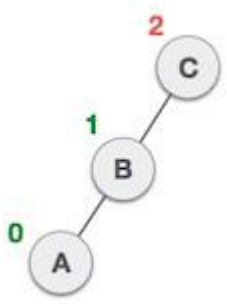


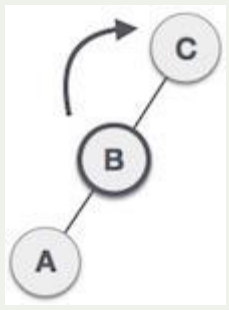
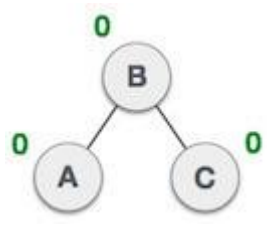
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

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

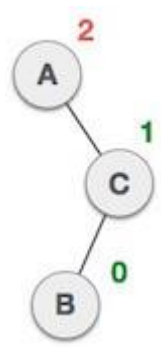
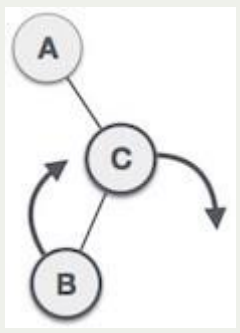
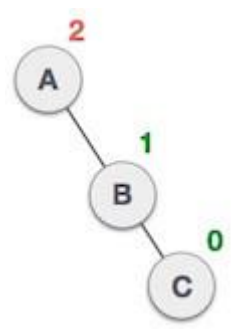
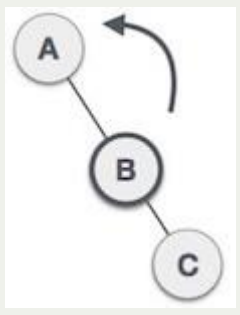
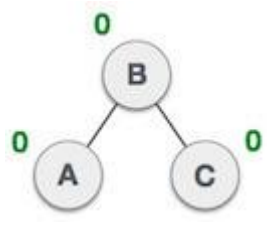
Let us understand each and every step very clearly:

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**

	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

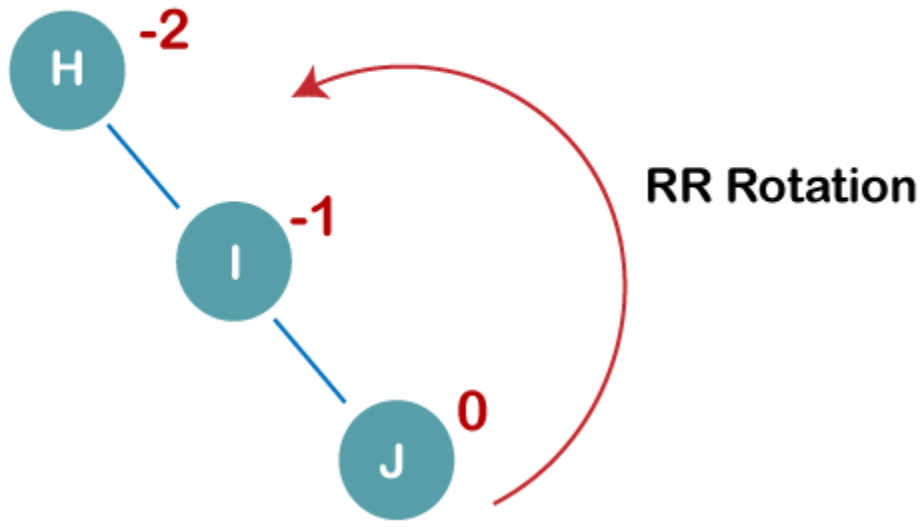
As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path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

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

Q: Construct an AVL tree having the following elements

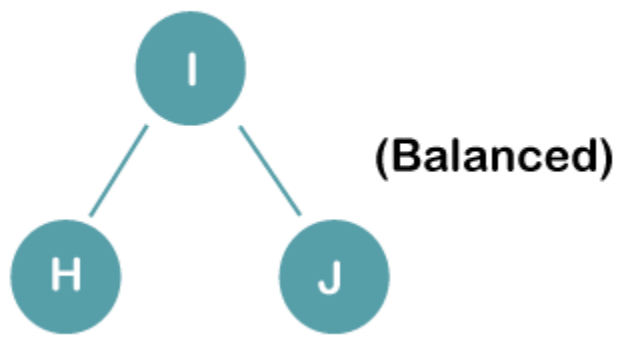
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

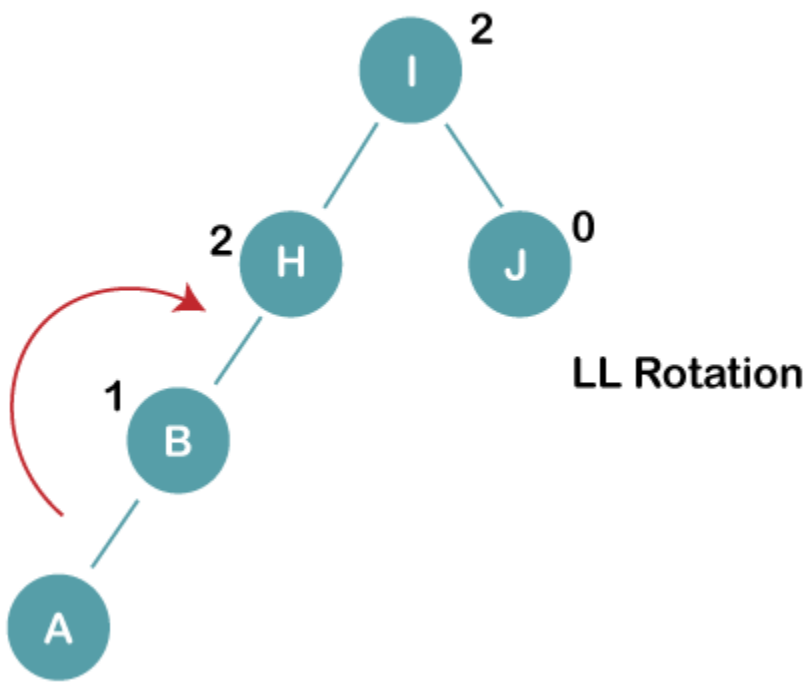


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

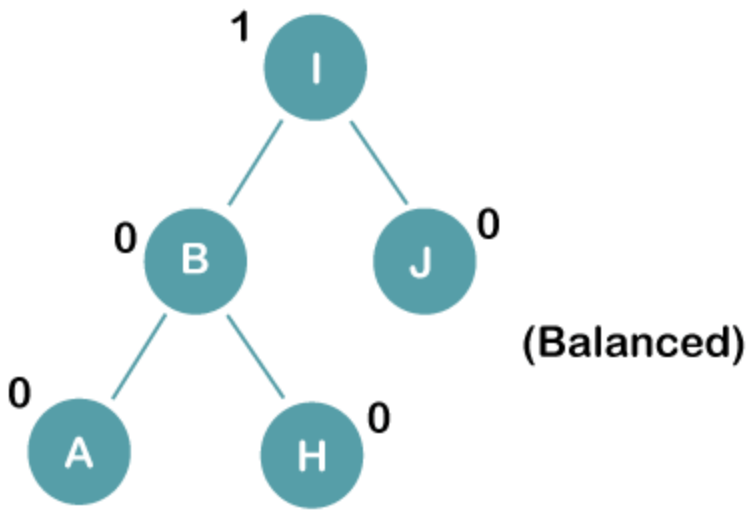


2. Insert B, A

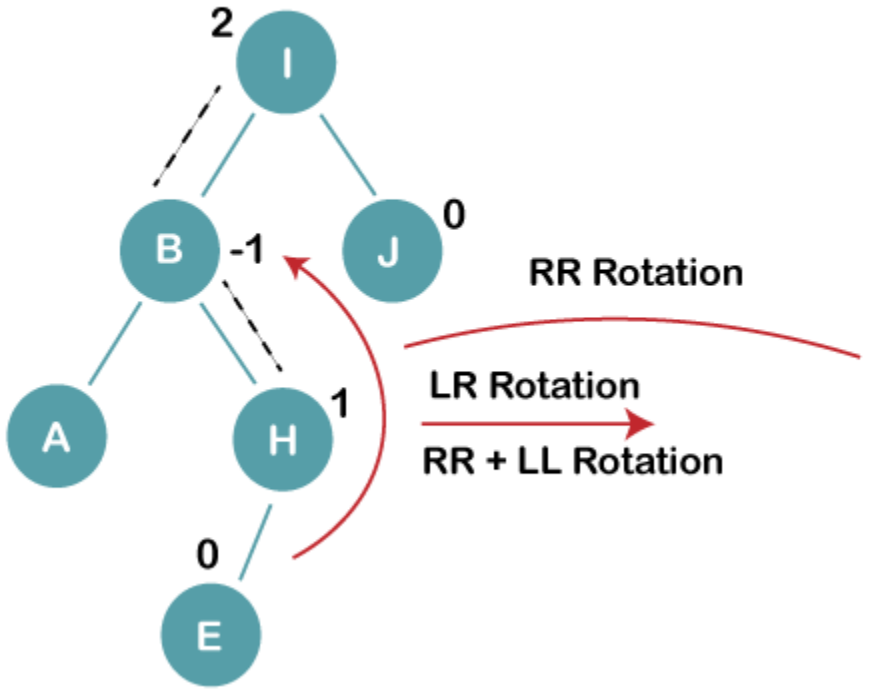


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



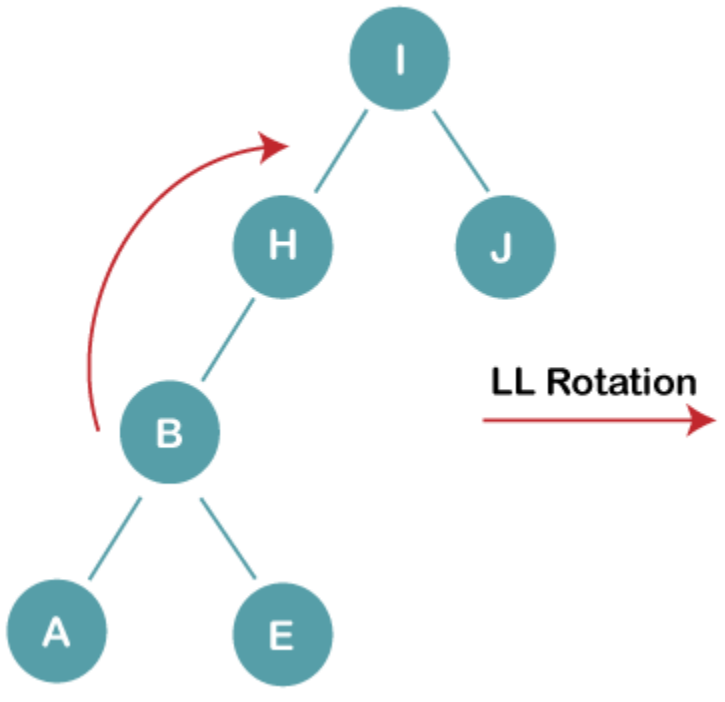
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

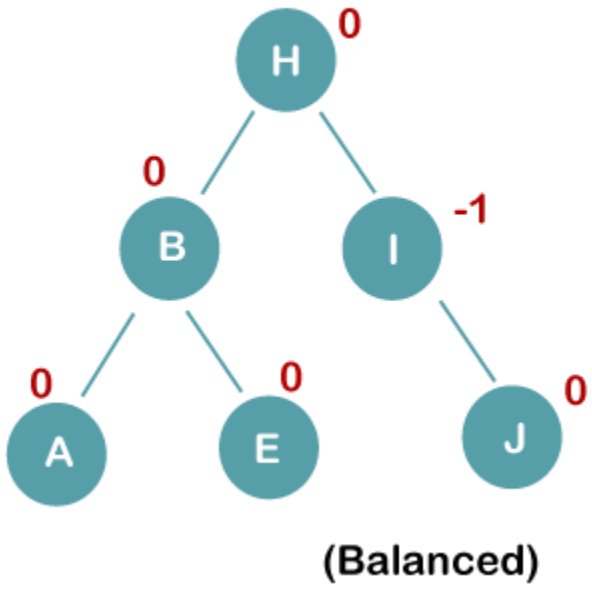
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:



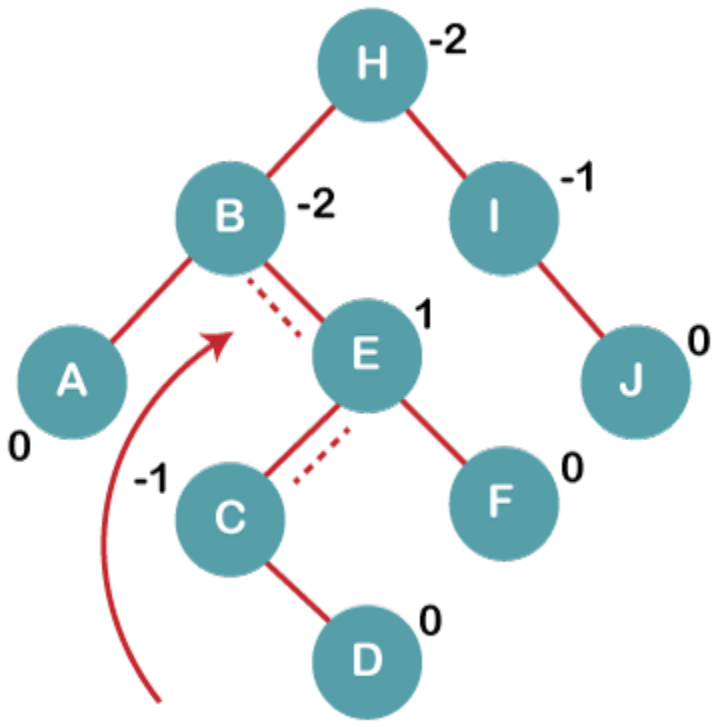
3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



(Balanced)

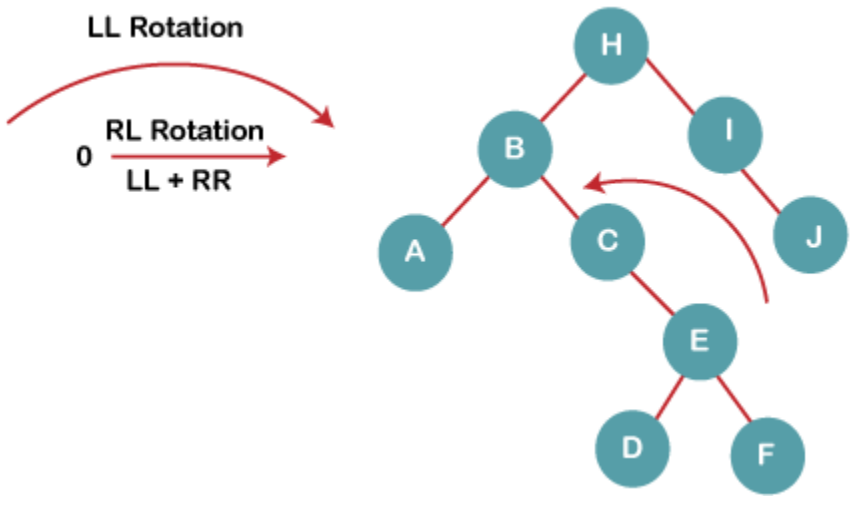
4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

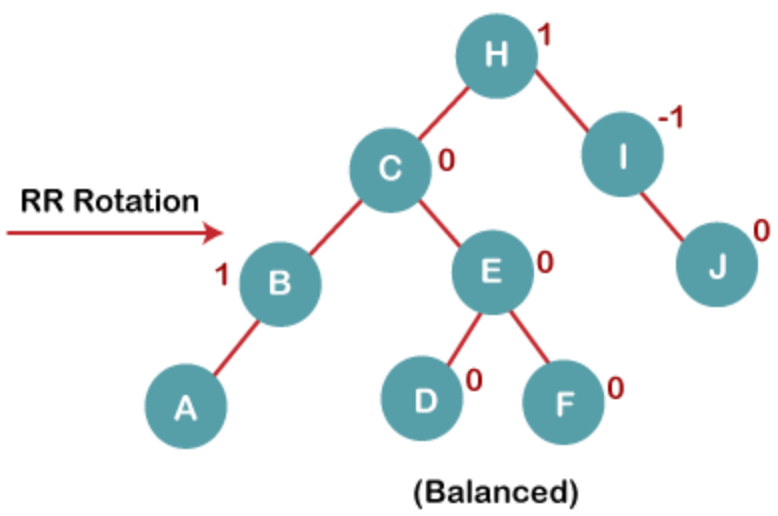
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

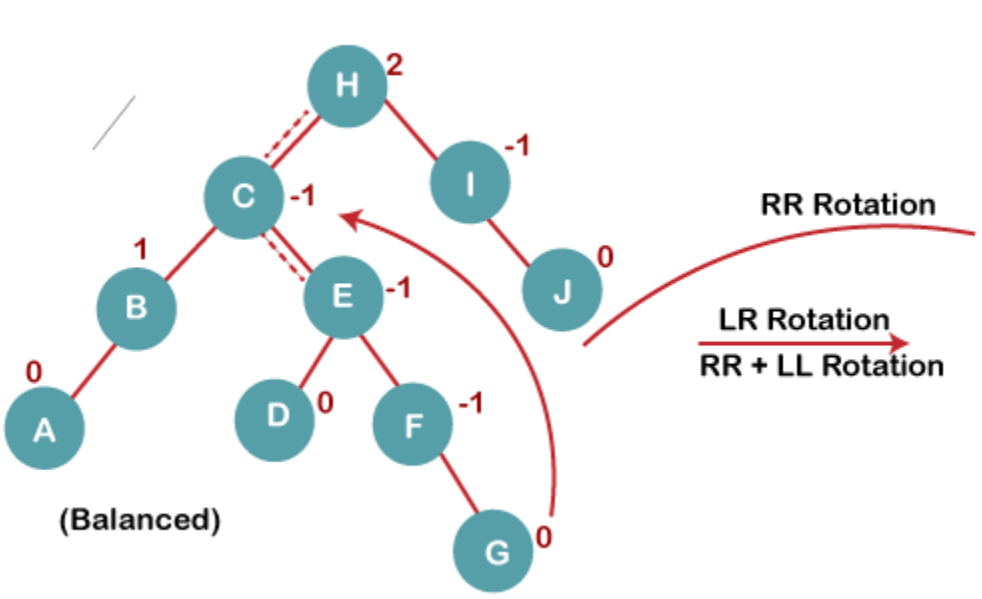


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



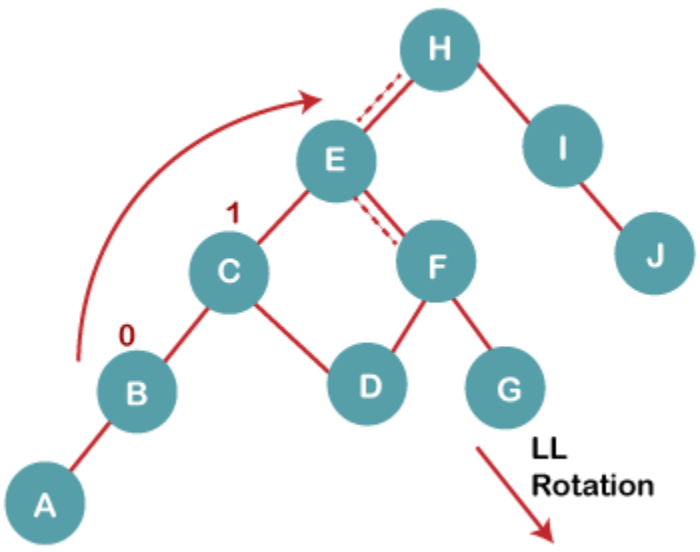
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

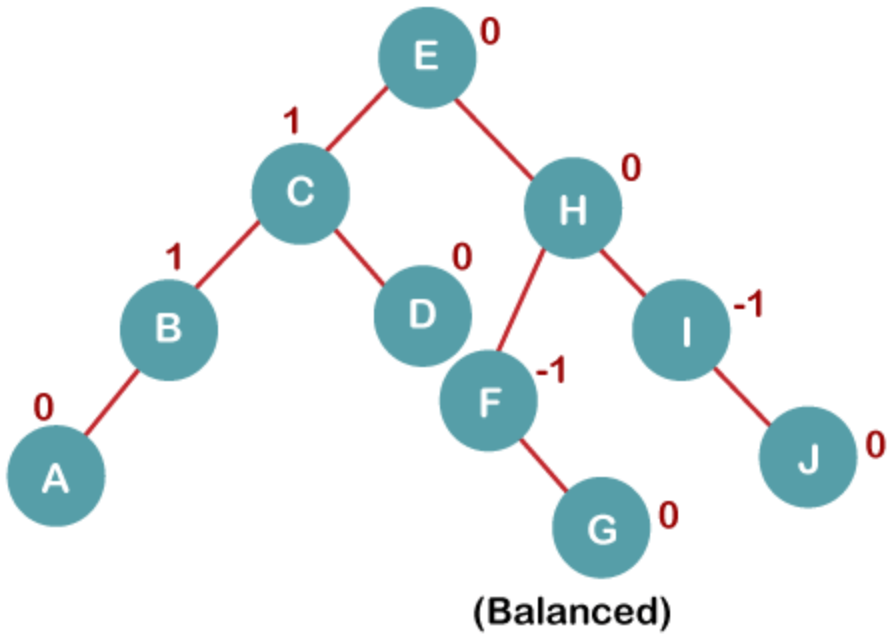
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

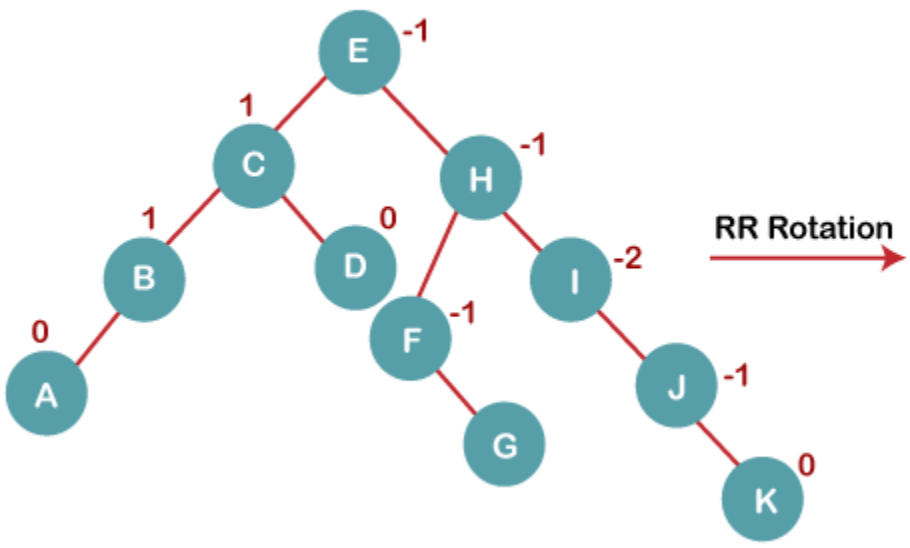


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

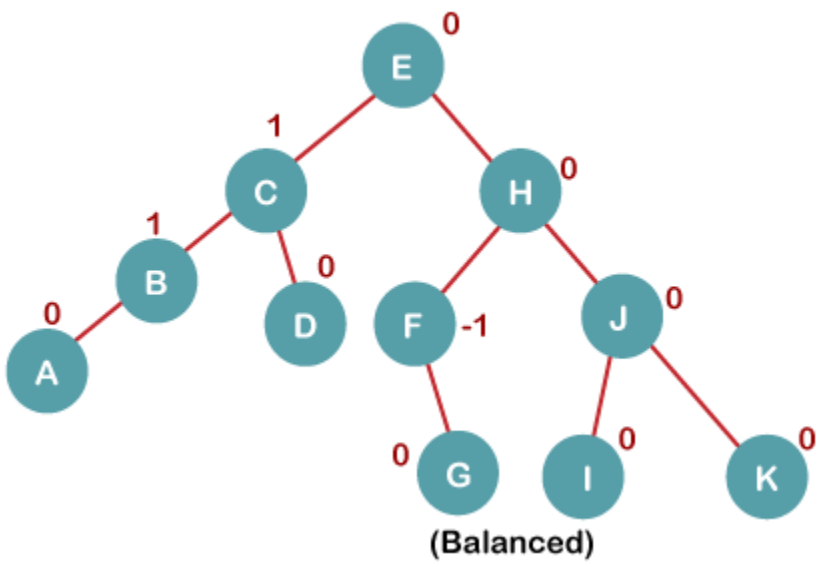


6. Insert K



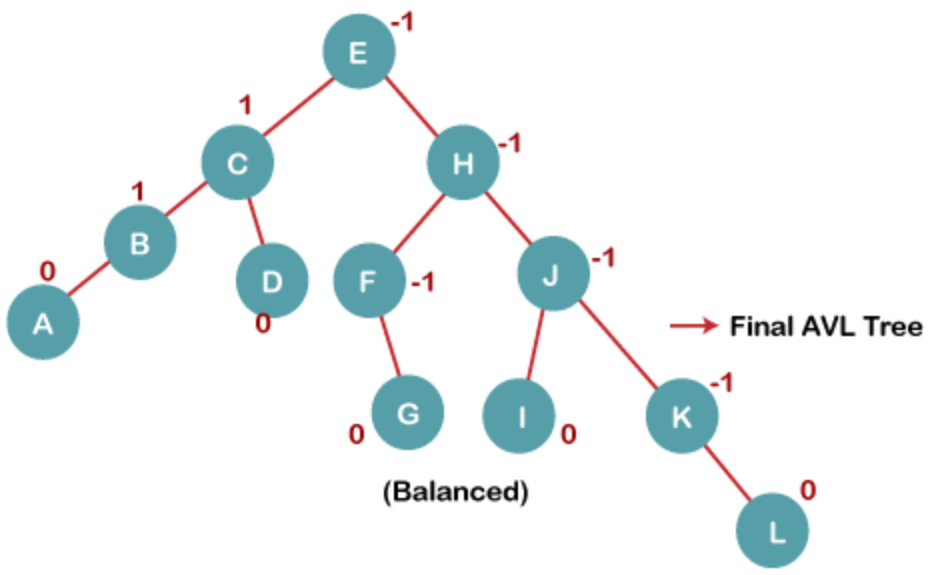
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



B Tree

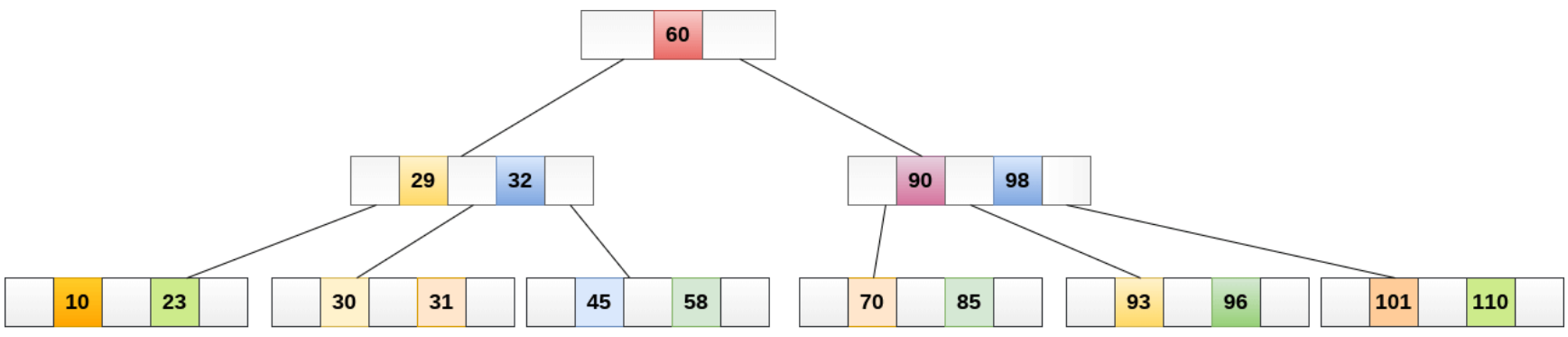
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively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

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

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

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

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



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

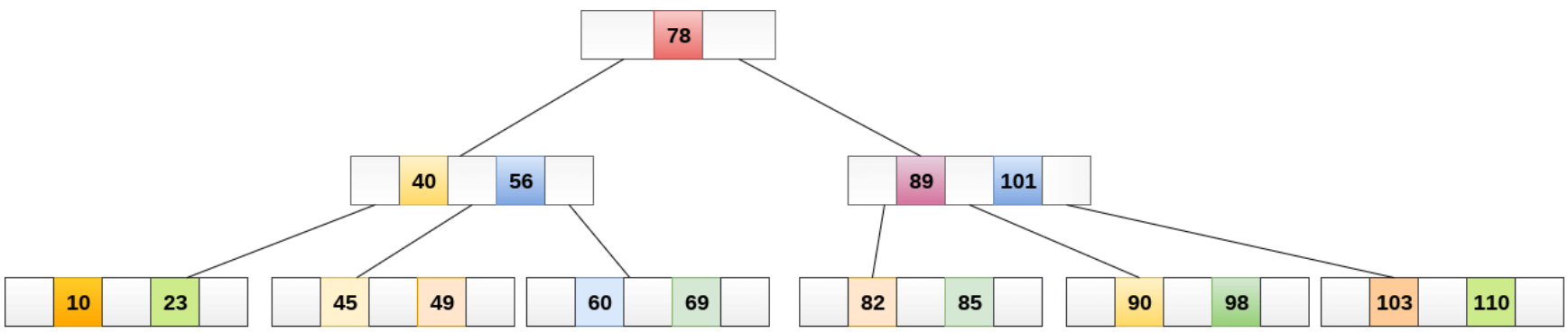
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



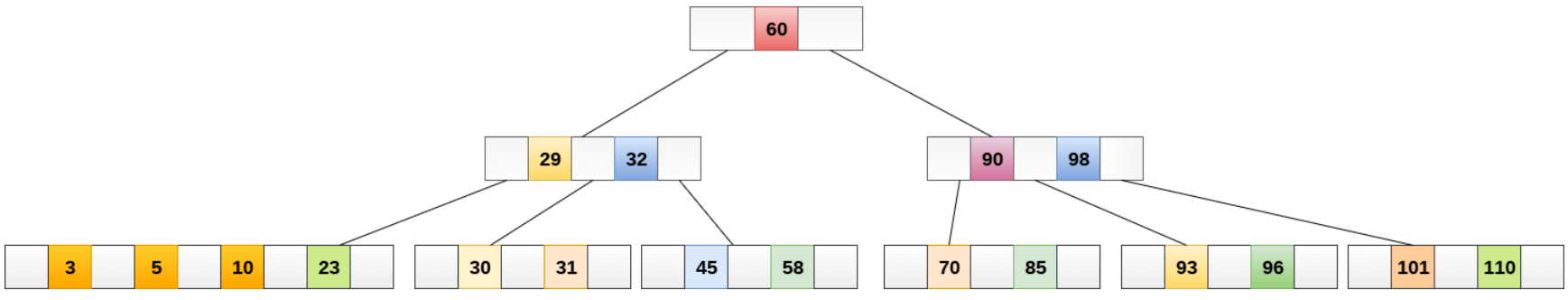
Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

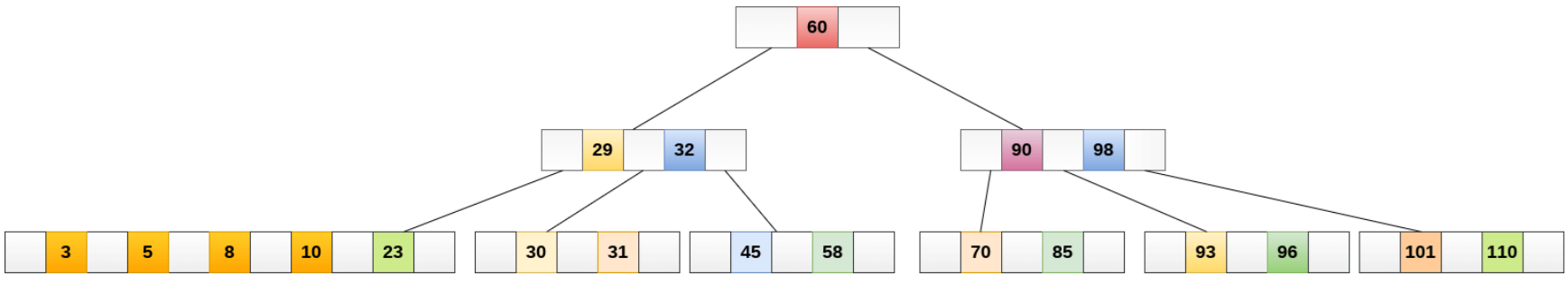
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

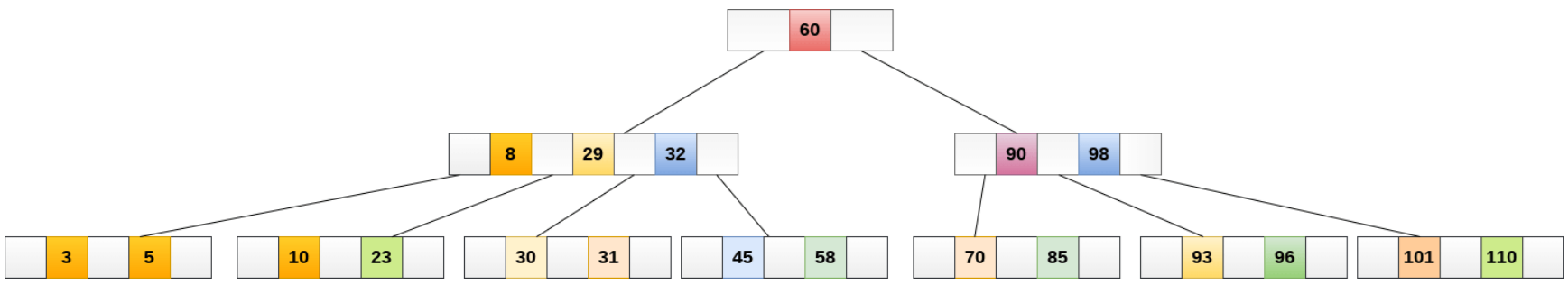
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

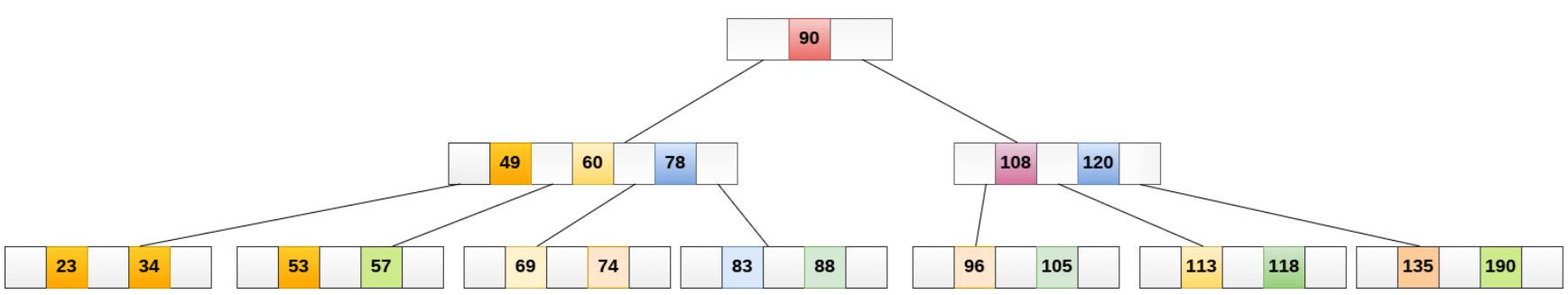
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

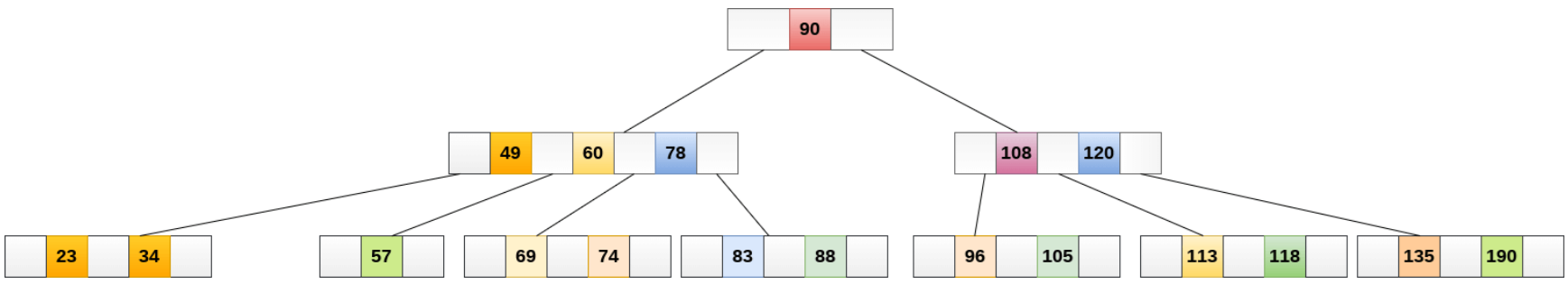
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

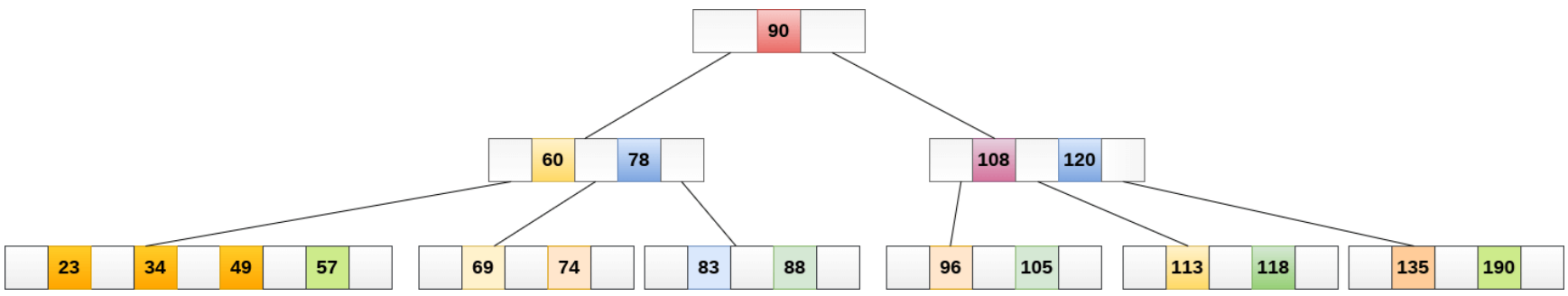


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

B+ Tree

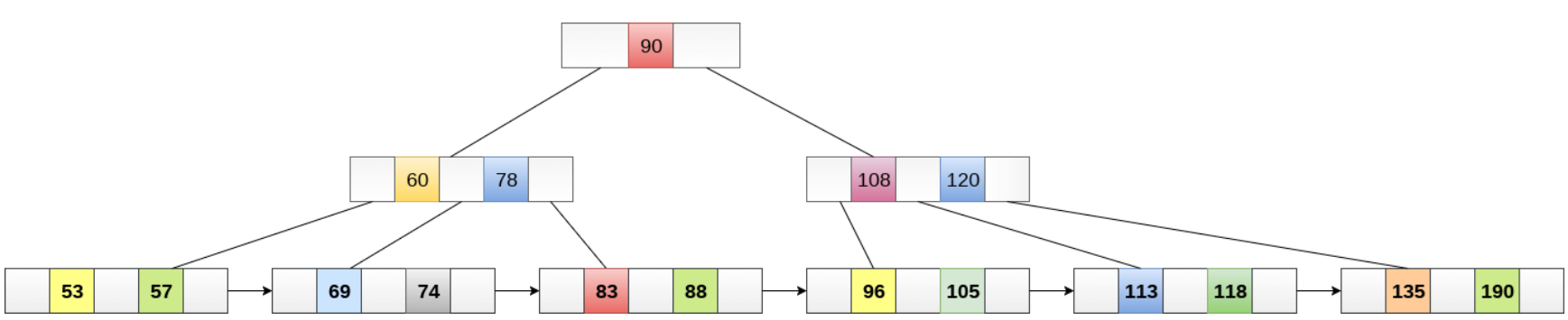
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

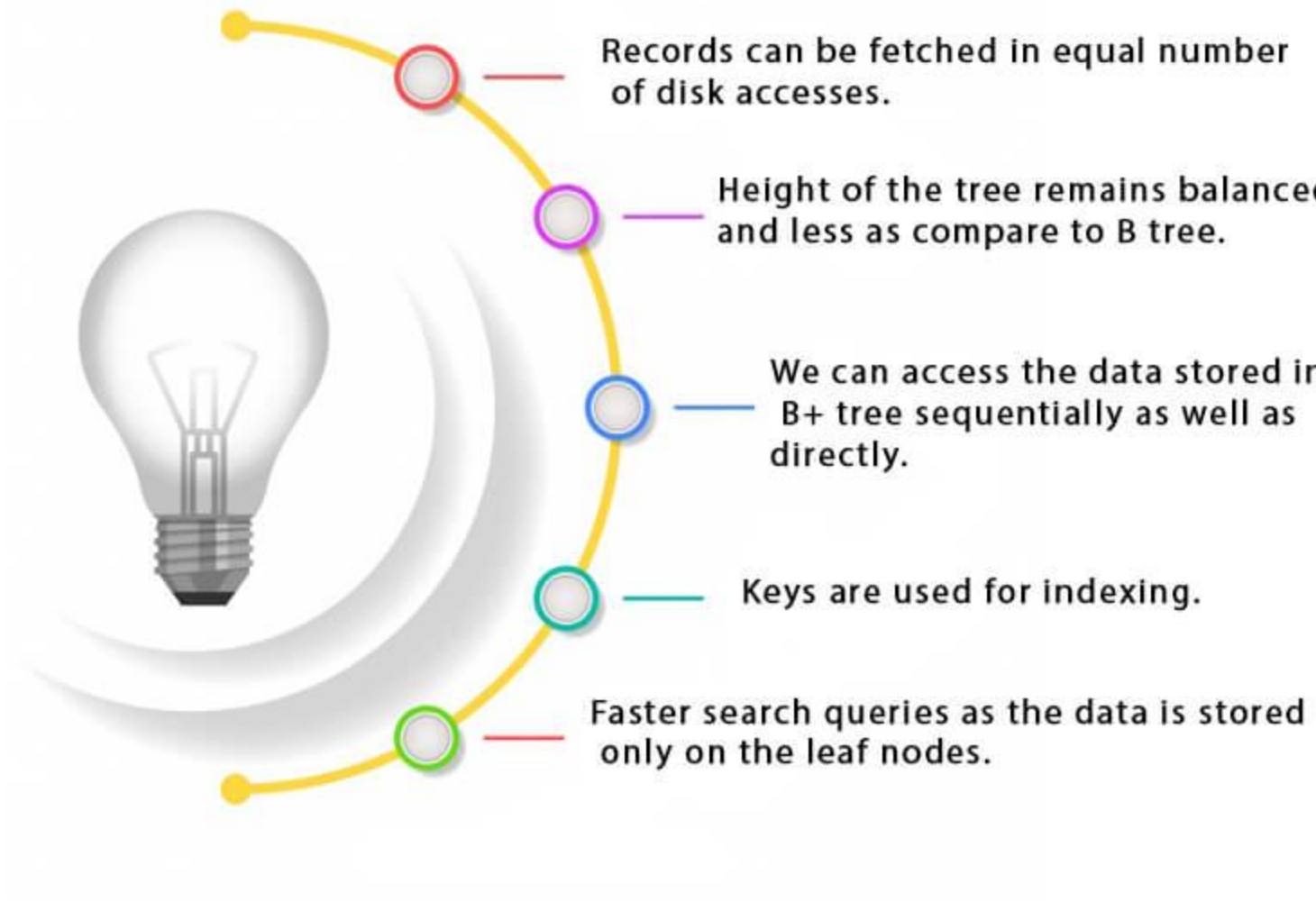
B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.



B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Insertion in B+ Tree

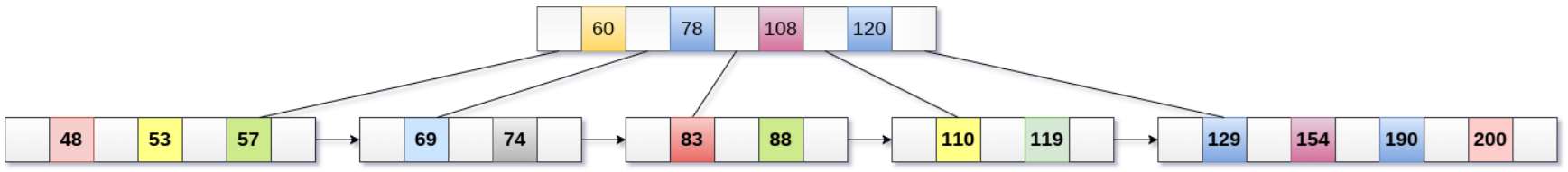
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

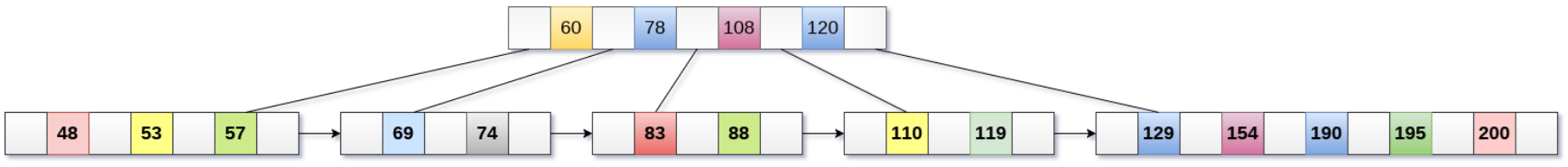
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

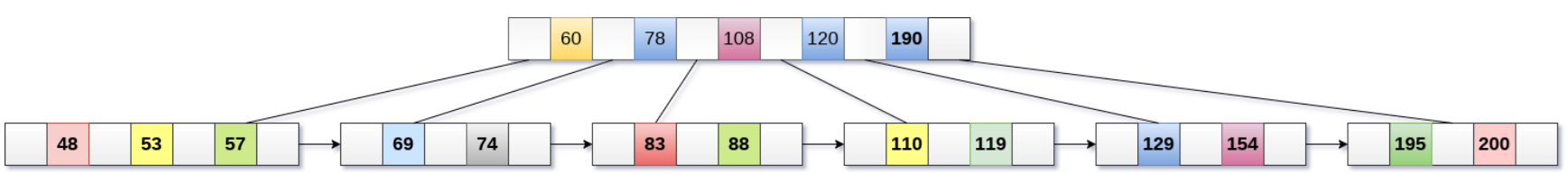
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



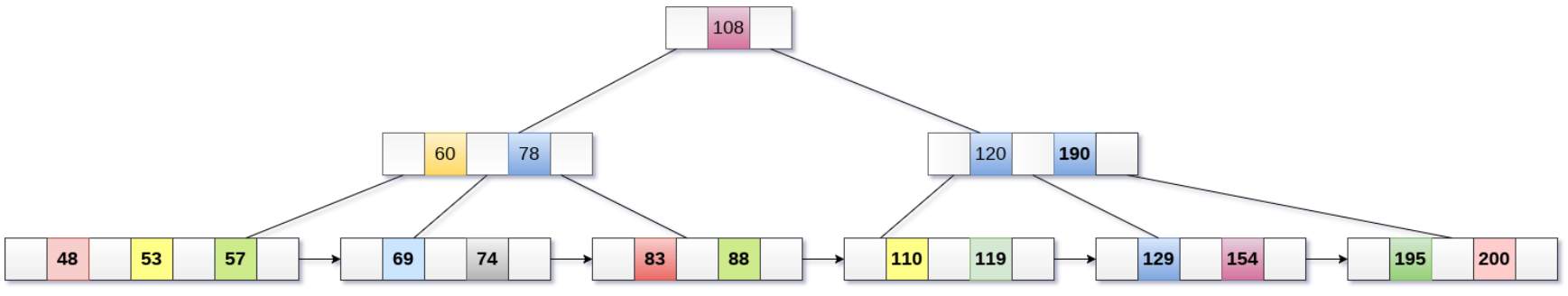
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

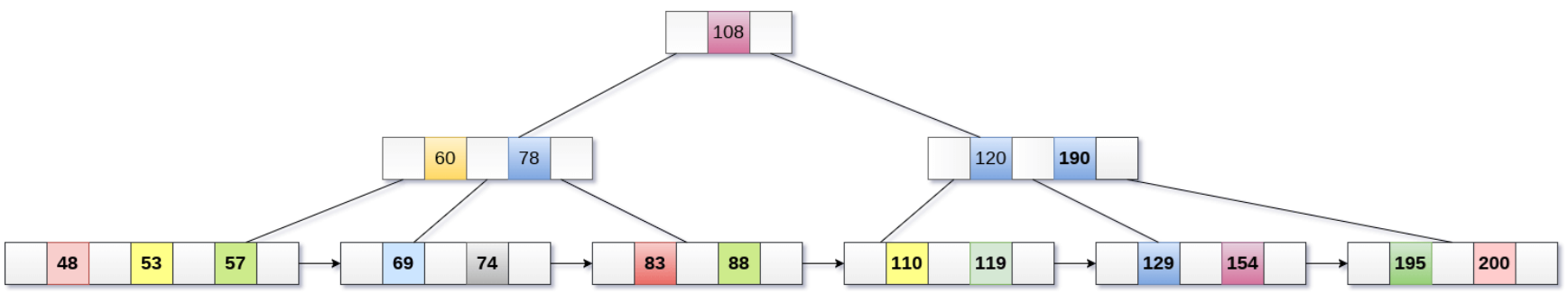
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

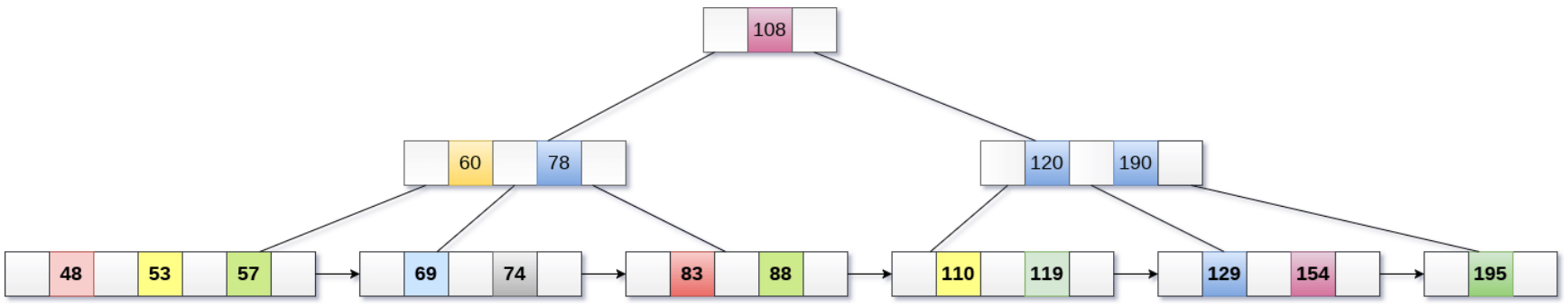
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

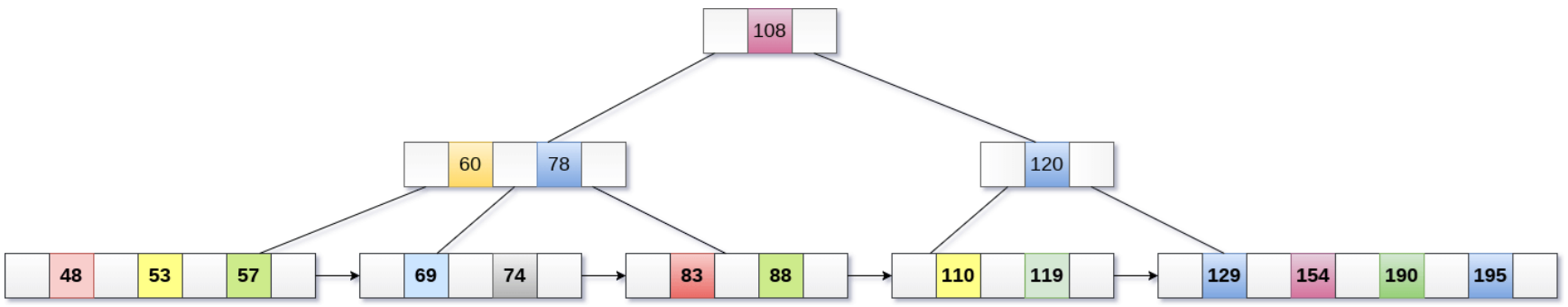
Delete the key 200 from the B+ Tree shown in the following figure.



200 is present in the right sub-tree of 190, after 195. delete it.

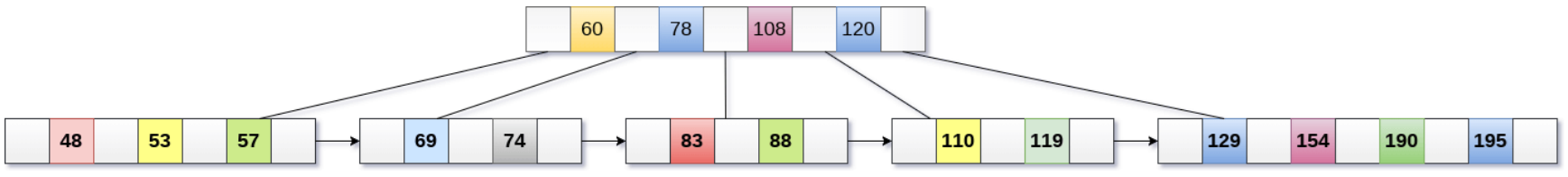


Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.



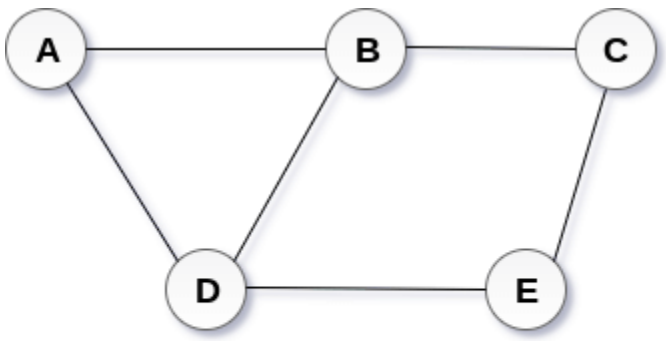
Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



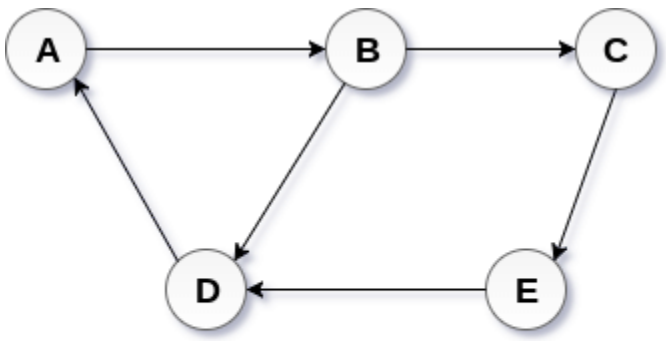
Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

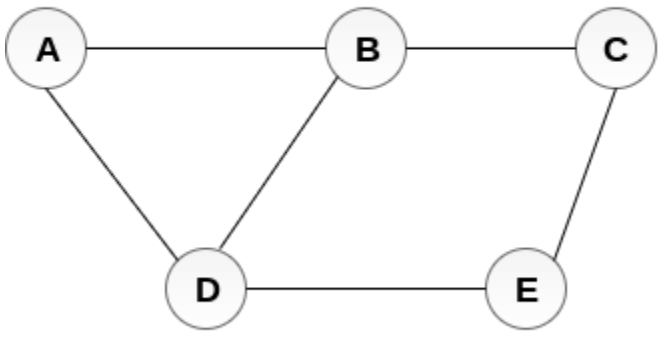
There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

1. Sequential Representation

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension **n x n**.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j.

An undirected graph and its adjacency matrix representation is shown in the following figure.



Undirected Graph

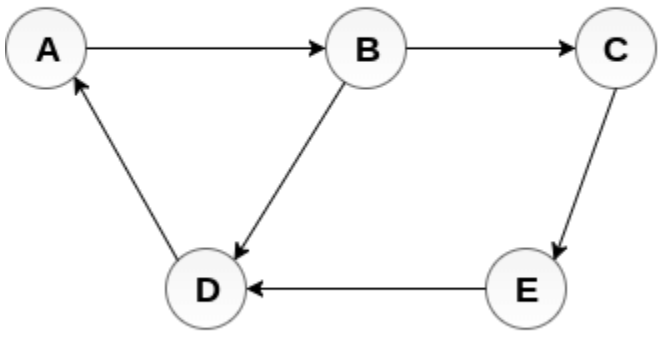
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

in the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

A directed graph and its adjacency matrix representation is shown in the following figure.



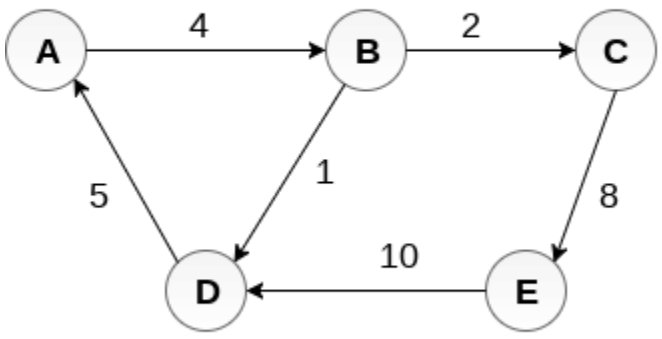
Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



Weighted Directed Graph

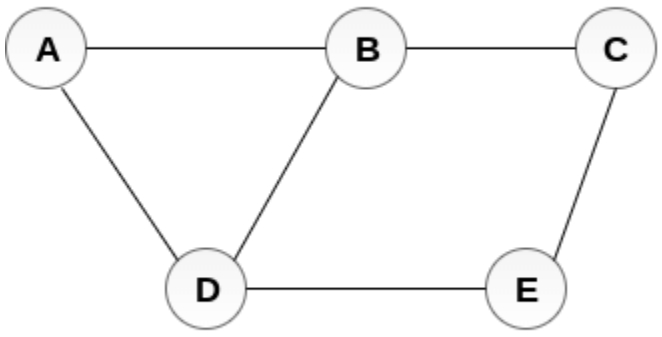
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

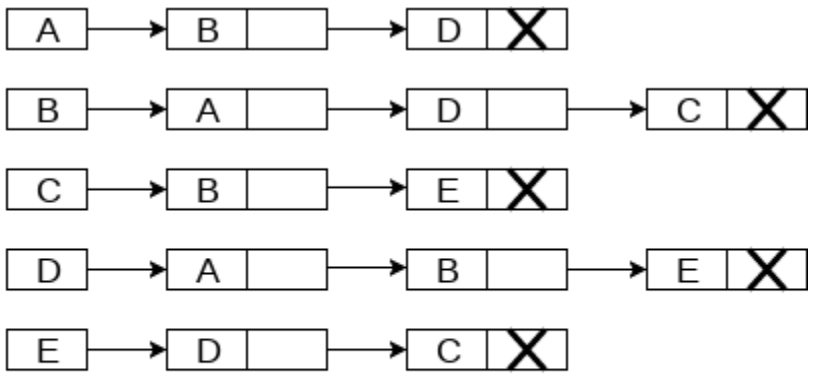
Linked Representation

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.



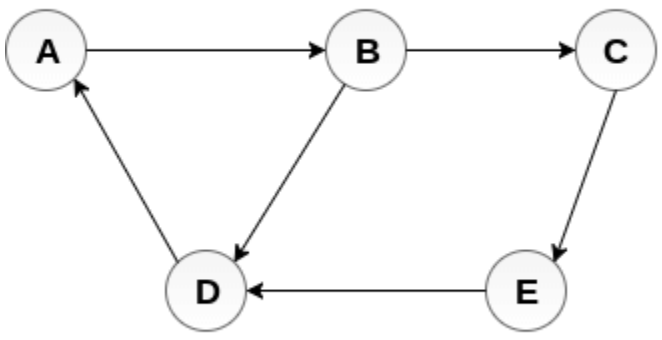
Undirected Graph



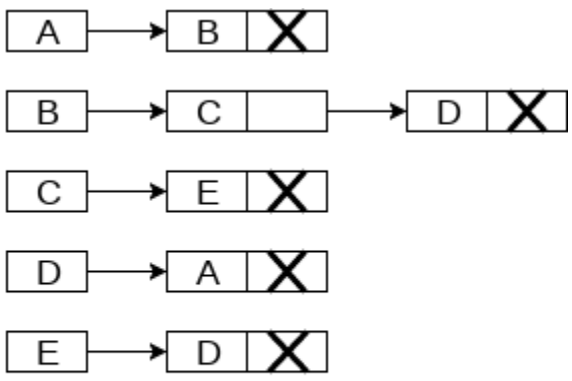
Adjacency List

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.



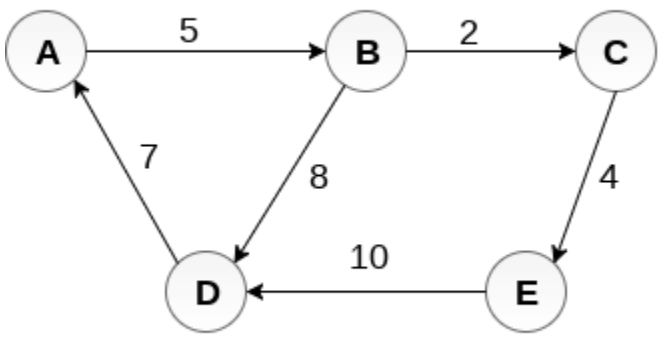
Directed Graph



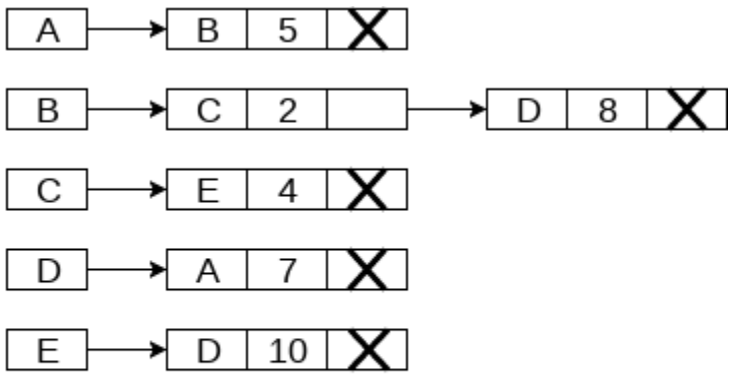
Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

Graph Traversal Algorithm

In this part of the tutorial we will discuss the techniques by using which, we can traverse all the vertices of the graph.

Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs. Lets discuss each one of them in detail.

- Breadth First Search
- Depth First Search

Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

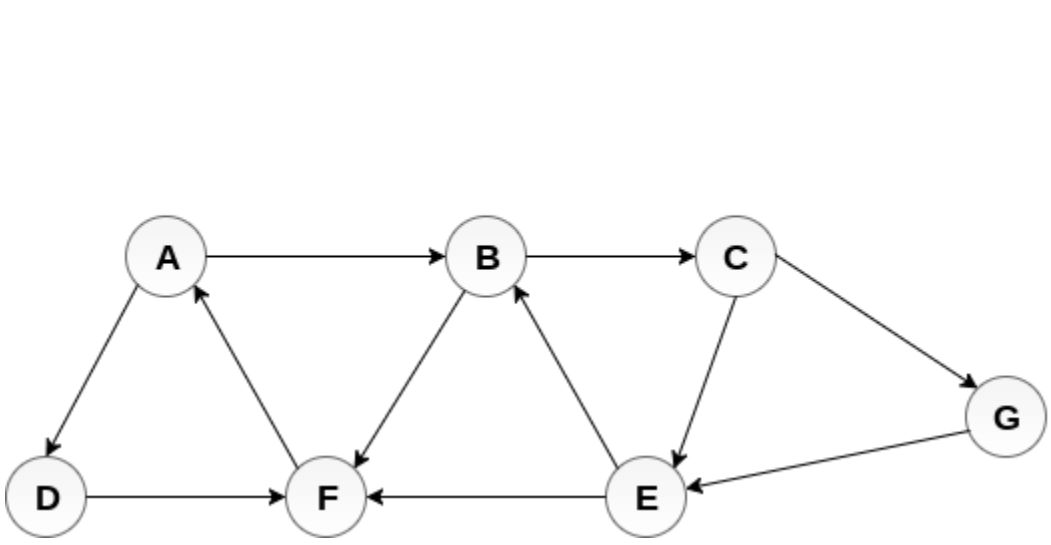
The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Enqueue the starting node A and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until QUEUE is empty
- **Step 4:** Dequeue a node N. Process it and set its STATUS = 3 (processed state).
- **Step 5:** Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]
- **Step 6:** EXIT

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

Lets start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

1. QUEUE1 = {C, F}
2. QUEUE2 = { A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Now, backtrack from E to A, using the nodes available in QUEUE2.

The minimum path will be **A** → **B** → **C** → **E**.

Depth First Search (DFS) Algorithm

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

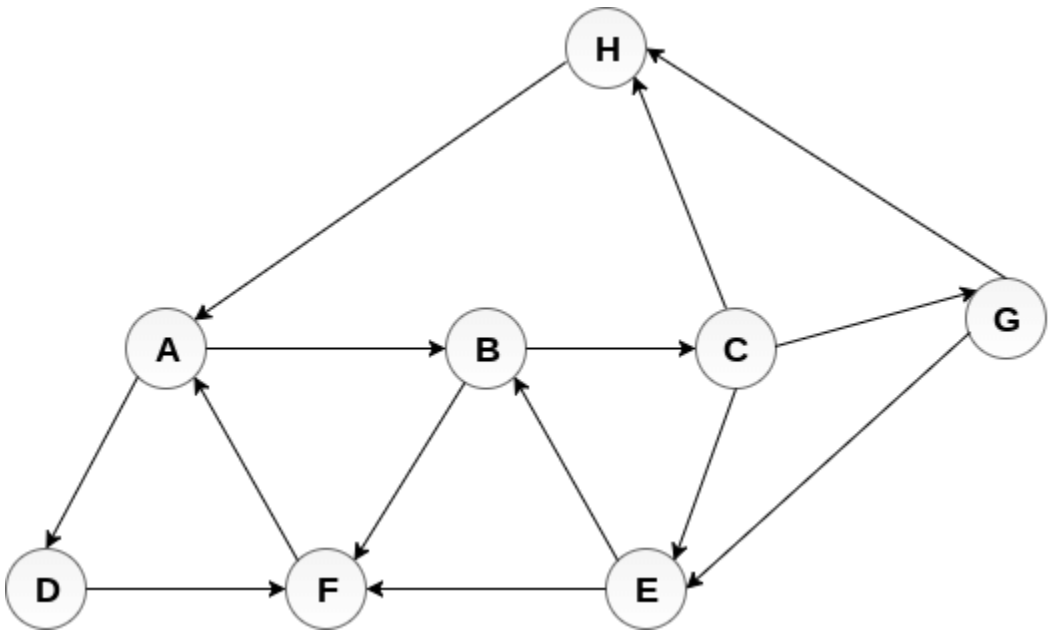
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
  [END OF LOOP]
- **Step 6:** EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

Push H onto the stack

1. STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

1. Print H
2. STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

1. Print A
2. Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

1. Print D
2. Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

1. Print F
2. Stack : B

Pop the top of the stack i.e. B and push all the neighbours

1. Print B
2. Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

1. Print C
2. Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

1. Print G
2. Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

1. Print E
2. Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

1. H → A → D → F → B → C → G → E

Spanning Tree

In this topic, we will learn about the ***spanning tree*** and ***minimum spanning tree***. Before knowing about the spanning trees, we should know about some graphs.

The following are the types of graphs:

- **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional.
  It can also be defined as a graph in which set of V vertices and set of E edges, each edge connecting two different vertices.
- **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
- **Directed graph:** A directed graph is defined as a graph in which set of V vertices and set of Edges, each connecting two different vertices, but it is not mandatory that node points in the opposite direction also.

What is a Spanning tree?

If we have a graph containing V vertices and E edges, then the graph can be represented as:

G(V, E)

If we create the spanning tree from the above graph, then the spanning tree would have the same number of vertices as the graph, but the vertices are not equal. The edges in the spanning tree would be equal to the number of edges in the graph minus 1.

Suppose the spanning tree is represented as:

G`(V`, E`)

where,

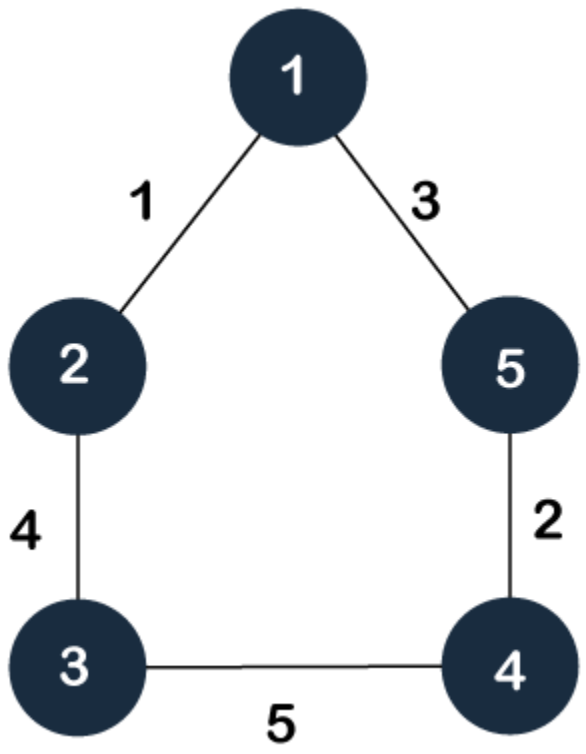
V=V`

E`∈ E -1

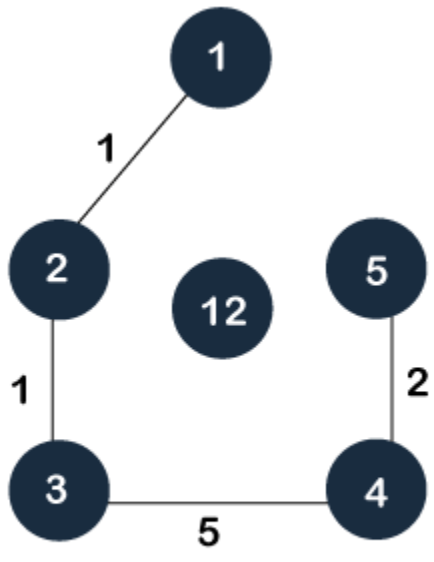
E`=|V| - 1

Let's understand through an example.

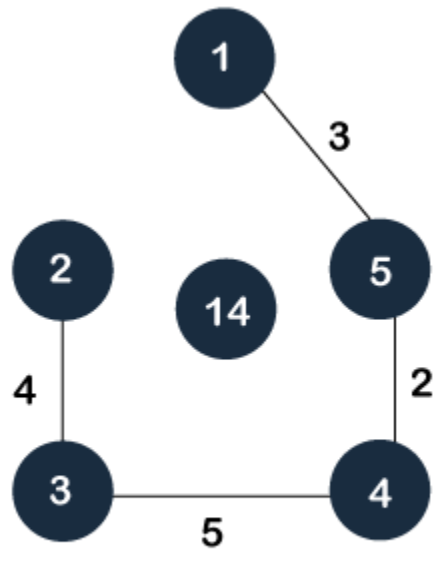
Suppose we want to create the spanning tree of the graph, which is shown below:



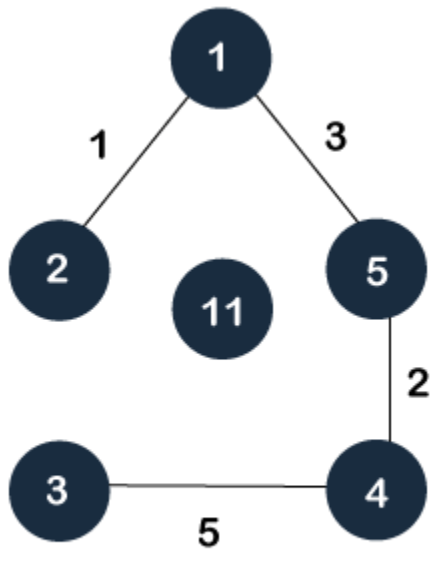
As we know, that spanning tree contains the same number of vertices as the graph, so the total number of vertices in the graph is 5; therefore, the spanning tree will also contain the 5 vertices. The edges in the spanning tree are equal to the number of vertices in the graph minus 1; therefore, the number of edges is 4. Three spanning trees can be created, which are shown below:



Spanning tree 1



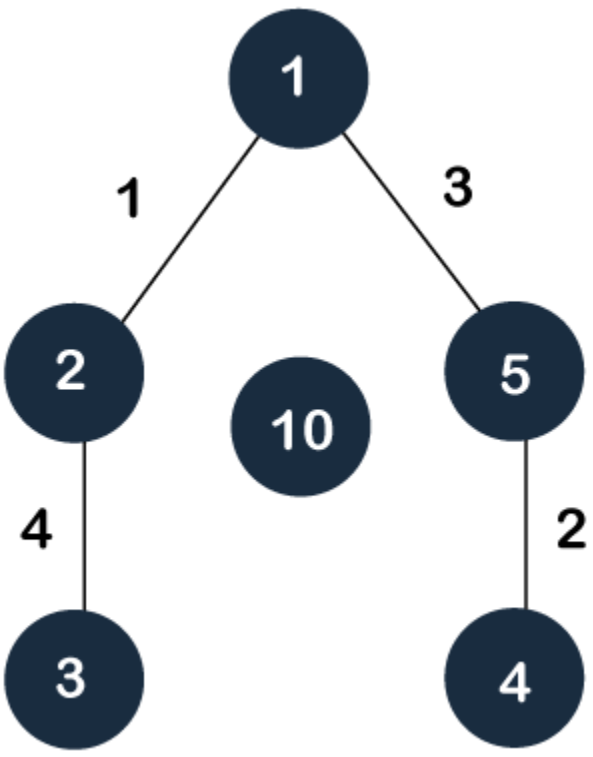
Spanning tree 2



Spanning tree 3

Minimum Spanning Trees

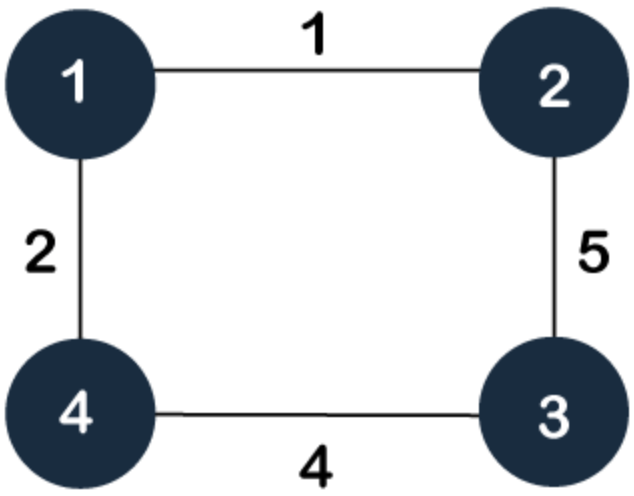
The minimum spanning tree is the tree whose sum of the edge weights is minimum. From the above spanning trees, the total edge weight of the spanning tree 1 is 12, the total edge weight of the spanning tree 2 is 14, and the total edge weight of the spanning tree 3 is 11; therefore, the total edge weight of the spanning tree 3 is minimum. From the above graph, we can also create one more spanning tree as shown below:



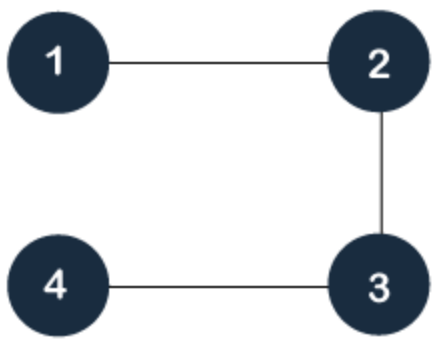
In the above tree, the total edge weight is 10 which is less than the above spanning trees; therefore, the minimum spanning tree is a tree which is having an edge weight, i.e., 10.

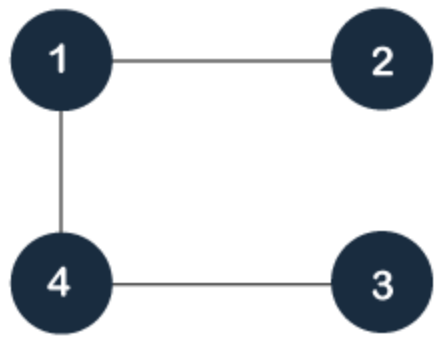
Properties of Spanning tree

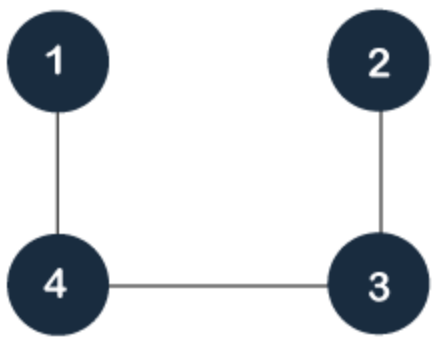
- A connected graph can contain more than one spanning tree. The spanning trees which are minimally connected or we can say that the tree which is having a minimum total edge weight would be considered as the minimum spanning tree.
- All the possible spanning trees that can be created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- The spanning tree does not contain any cycle. Let's understand this property through an example. Suppose we have the graph which is given below:

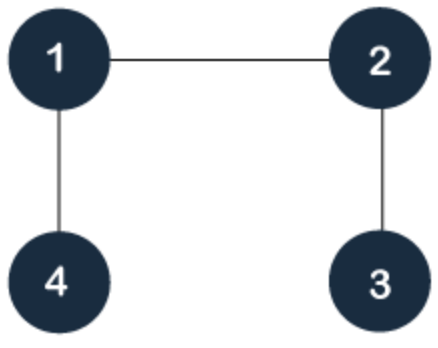


We can create the spanning trees of the above graph, which are shown below:



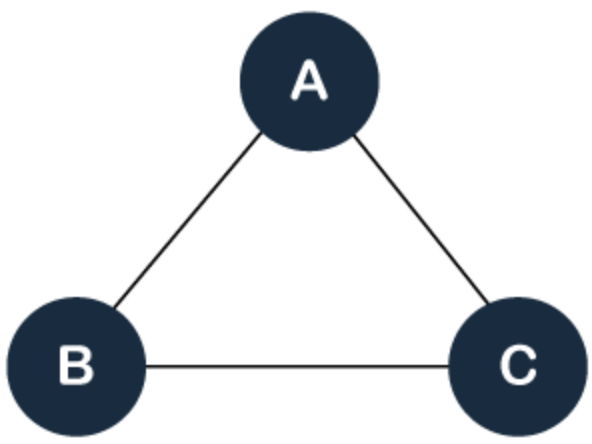




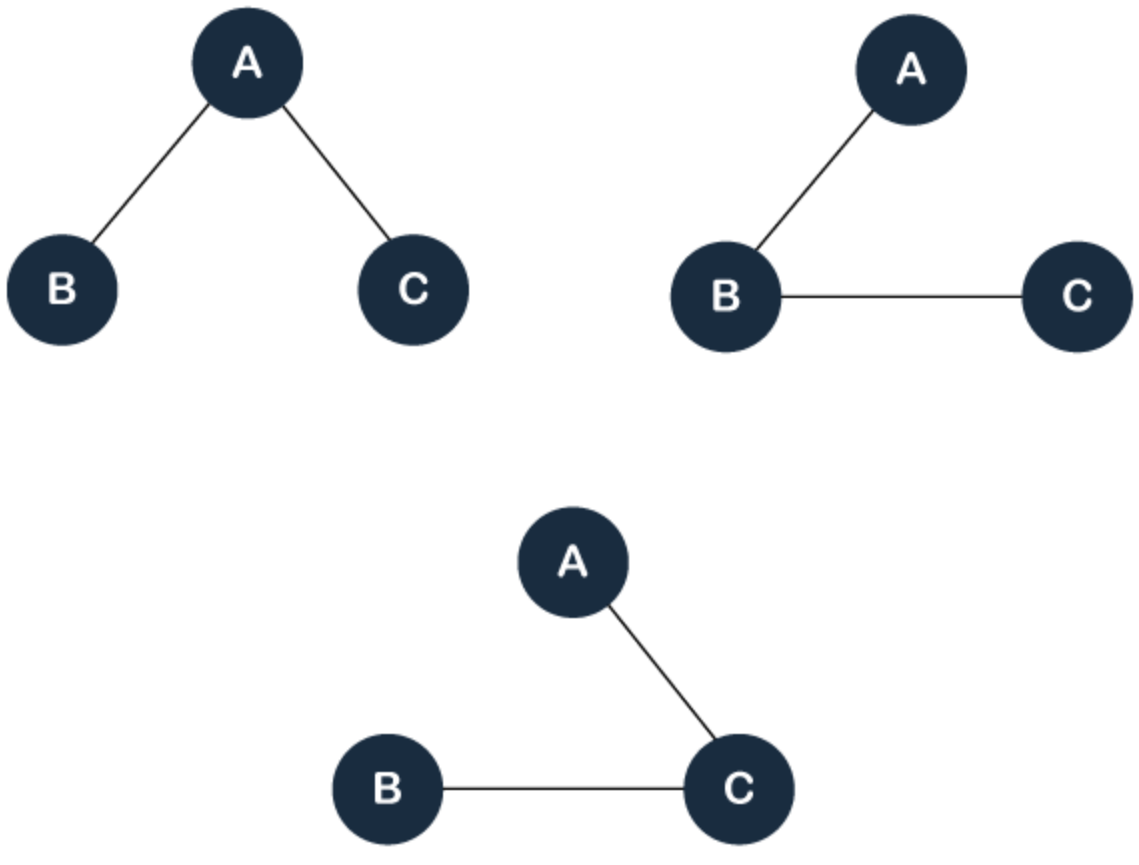


As we can observe in the above spanning trees that one edge has been removed. If we do not remove one edge from the graph, then the tree will form a cycle, and that tree will not be considered as the spanning tree.

- The spanning tree cannot be disconnected. If we remove one more edge from any of the above spanning trees as shown below:
  The above tree is not a spanning tree because it is disconnected now.
- If two or three edges have the same edge weight, then there would be more than two minimum spanning trees. If each edge has a distinct weight, then there will be only one or unique minimum spanning tree.
- A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where n is the number of vertices in the graph. For example, the value of n is 5 then the number of spanning trees would be equal to 125.
- Each connected and undirected graph contains at least one spanning tree.
- The disconnected graph does not contain any spanning tree, which we have already discussed.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges. Let's understand this property through an example.
  A complete graph is a graph in which each pair of vertices are connected. Consider the complete graph having 3 vertices, which is shown below:



We can create three spanning trees from the above graph shown as below:



According to this property, the maximum number of edges from the graph can be formulated as (e-n+1) where e is the number of edges, n is the number of vertices. When we substitute the value of e and n in the formula, then we get 1 value. It means that we can remove maximum 1 edge from the graph to make a spanning tree. In the above spanning trees, the one edge has been removed.

Applications of Spanning trees

The following are the applications of the spanning trees:

- **Building a network:** Suppose there are many routers in the network connected to each other, so there might be a possibility that it forms a loop. So, to avoid the formation of the loop, we use the tree data structure to connect the routers, and a minimum spanning tree is used to minimally connect them.
- **Clustering:** Here, clustering means that grouping the set of objects in such a way that similar objects belong to the same group than to the different group. Our goal is to divide the n objects into k groups such that the distance between the different groups gets maximized.

How can clustering be achieved?

First, we divide the n objects into k groups.

Secondly, we will combine these clusters iteratively by adding an edge between them.

Stop when we reach the k clusters.

Approach

One of the approaches that can be used to obtain clustering is to compute a minimum spanning tree. The minimum spanning tree can be simply calculated by dropping the k-1 most heavily weighted edge from the graph.

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows.

Algorithm

- LINEAR_SEARCH(A, N, VAL)
- **Step 1:** [INITIALIZE] SET POS = -1
- **Step 2:** [INITIALIZE] SET I = 1
- **Step 3:** Repeat Step 4 while I<=N
- **Step 4:** IF A[I] = VAL
  SET POS = I
  PRINT POS
  Go to Step 6
  [END OF IF]
  SET I = I + 1
  [END OF LOOP]
- **Step 5:** IF POS = -1
  PRINT " VALUE IS NOT PRESENTIN THE ARRAY "
  [END OF IF]
- **Step 6:** EXIT

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

C Program

```
1.  #include<stdio.h>
2.  void main ()
3.  {
4.      int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
5.      int item, i,flag;
6.      printf("\nEnter Item which is to be searched\n");
7.      scanf("%d",&item);
8.      for (i = 0; i< 10; i++)
9.      {
10.         if(a[i] == item)
11.         {
12.             flag = i+1;
```

```
13.             break;
14.         }
15.         else
16.         flag = 0;
17.     }
18.     if(flag != 0)
19.     {
20.         printf("\nItem found at location %d\n",flag);
21.     }
22.     else
23.     {
24.         printf("\nItem not found\n");
25.     }
26. }
```

Output:

```
Enter Item which is to be searched
20
Item not found
Enter Item which is to be searched
23
Item found at location 2
```

Java Program

```
1.  import java.util.Scanner;
2.
3.  public class Leniear_Search {
4.  public static void main(String[] args) {
5.      int[] arr = {10, 23, 15, 8, 4, 3, 25, 30, 34, 2, 19};
6.      int item,flag=0;
7.      Scanner sc = new Scanner(System.in);
8.      System.out.println("Enter Item ?");
9.      item = sc.nextInt();
10.     for(int i = 0; i<10; i++)
11.     {
12.         if(arr[i]==item)
13.         {
14.             flag = i+1;
15.             break;
16.         }
17.         else
18.             flag = 0;
19.     }
20.     if(flag != 0)
21.     {
22.         System.out.println("Item found at location" + flag);
23.     }
24.     else
25.         System.out.println("Item not found");
26.
27. }
28. }
```

Output:

```
Enter Item ?
23
Item found at location 2
Enter Item ?
22
Item not found
```

C# Program

```
1.  using System;
2.
3.  public class LinearSearch
4.  {
5.      public static void Main()
6.      {
7.          int item, flag = 0;
8.          int[]  a= {10, 23, 5, 90, 89, 34, 12, 34, 1, 78};
9.          Console.WriteLine("Enter the item value");
10.         item = Convert.ToInt32(Console.ReadLine());
11.         for(int i=0;i<10;i++)
12.         {
13.             if(item == a[i])
14.             {
15.                 flag = i+1;
16.                 break;
17.             }
18.             else
19.                 flag = 0;
20.         }
21.         if(flag != 0 )
22.         {
23.             Console.WriteLine("Item Found at Location " + flag);
24.         }
25.         else
26.             Console.WriteLine("Item Not Found");
27.
28.     }
29. }
```

Output:

```
Enter the item value
78
Item Found at Location 10

Enter the item value
22
Item not found
```

Python Program

```
1.  arr = [10,2,3,4,23,5,21,45,90,100];
2.  item = int(input("Enter the item which you want to search "));
3.  for i in range (0,len(arr)):
4.      if arr[i] == item:
5.          flag = i+1;
6.          break;
7.      else:
8.          flag = 0;
9.  if flag != 0:
10.     print("Item found at location %d" % (flag));
11. else :
12.     print("Item not found");
```

Output:

```
Enter the item which you want to search 2
Item found at location 2
Enter the item which you want to search 101
Item not found
```

Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

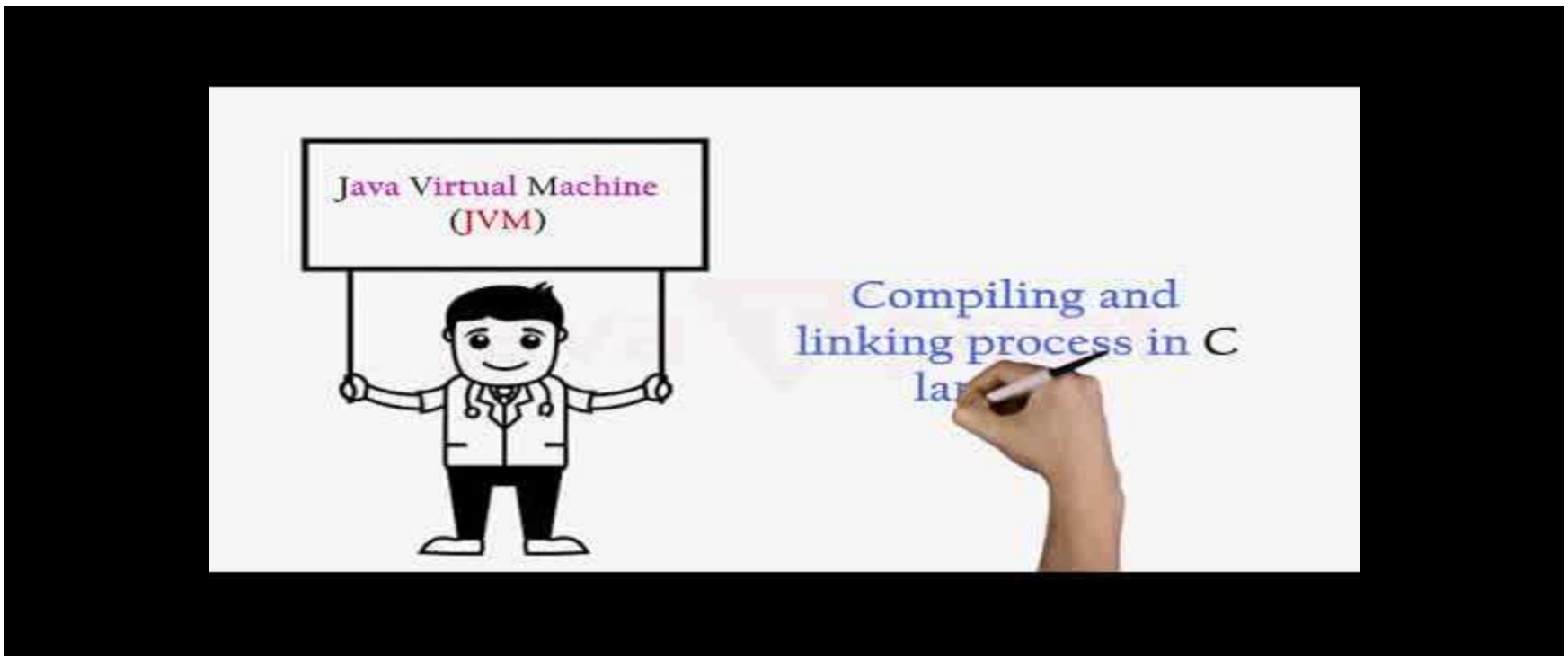
- **Step 1:** [INITIALIZE] SET BEG = lower_bound
  END = upper_bound, POS = - 1
- **Step 2:** Repeat Steps 3 and 4 while BEG <=END
- **Step 3:** SET MID = (BEG + END)/2
- **Step 4:** IF A[MID] = VAL
  SET POS = MID
  PRINT POS
  Go to Step 6
  ELSE IF A[MID] > VAL
  SET END = MID - 1
  ELSE
  SET BEG = MID + 1
  [END OF IF]
  [END OF LOOP]
- **Step 5:** IF POS = -1
  PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
  [END OF IF]
- **Step 6:** EXIT

Complexity

SN	Performance	Complexity
1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)
4	Worst case space complexity	O(1)

Example

Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. Find the location of the item 23 in the array.



In 1st step :

1. BEG = 0
2. END = 8ron
3. MID = 4
4. a[mid] = a[4] = 13 < 23, therefore

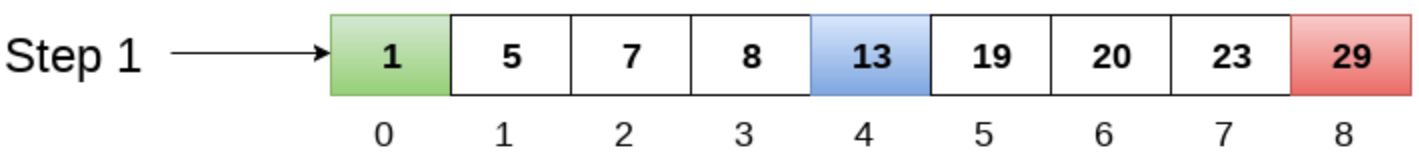
in Second step:

1. Beg = mid +1 = 5
2. End = 8
3. mid = 13/2 = 6
4. a[mid] = a[6] = 20 < 23, therefore;

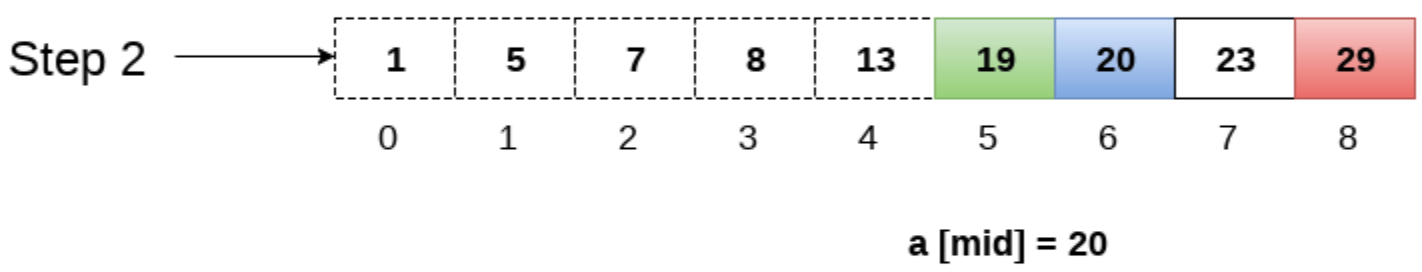
in third step:

1. beg = mid + 1 = 7
2. End = 8
3. mid = 15/2 = 7
4. a[mid] = a[7]
5. a[7] = 23 = item;
6. therefore, set location = mid;
7. The location of the item will be 7.

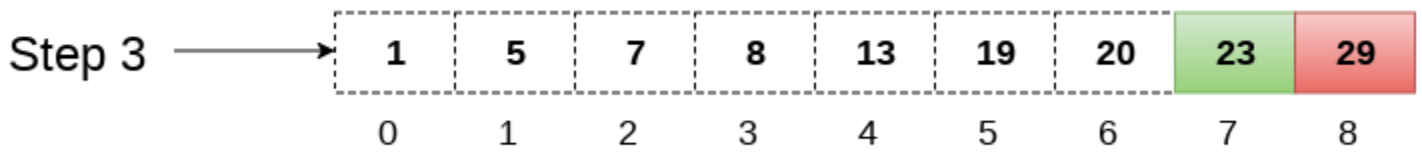
Item to be searched = 23



a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6



a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7



a [mid] = 23
23 = 23
loc = mid

Return location 7

Binary Search Program using Recursion

C program

```
1. #include<stdio.h>
2. int binarySearch(int[], int, int, int);
3. void main ()
4. {
5.     int arr[10] = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
6.     int item, location=-1;
7.     printf("Enter the item which you want to search ");
8.     scanf("%d",&item);
9.     location = binarySearch(arr, 0, 9, item);
10.    if(location != -1)
11.    {
12.        printf("Item found at location %d",location);
13.    }
14.    else
15.    {
16.        printf("Item not found");
17.    }
18. }
19. int binarySearch(int a[], int beg, int end, int item)
20. {
```

```
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }

    }
    return -1;
}
```

Output:

```
Enter the item which you want to search
19
Item found at location 2
```

Java

```
import java.util.*;
public class BinarySearch {
public static void main(String[] args) {
    int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
    int item, location = -1;
    System.out.println("Enter the item which you want to search");
    Scanner sc = new Scanner(System.in);
    item = sc.nextInt();
    location = binarySearch(arr,0,9,item);
    if(location != -1)
    System.out.println("the location of the item is "+location);
    else
        System.out.println("Item not found");
    }
public static int binarySearch(int[] a, int beg, int end, int item)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
```

```
            return binarySearch(a,beg,mid-1,item);
        }

    }
    return -1;
}
}
```

Output:

```
Enter the item which you want to search
45
the location of the item is 5
```

C#

```
using System;

public class LinearSearch
{
    public static void Main()
    {
    int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
    int location=-1;
    Console.WriteLine("Enter the item which you want to search ");
    int item = Convert.ToInt32(Console.ReadLine());
    location = binarySearch(arr, 0, 9, item);
    if(location != -1)
    {
        Console.WriteLine("Item found at location "+ location);
    }
    else
    {
        Console.WriteLine("Item not found");
    }
}
public static int binarySearch(int[] a, int beg, int end, int item)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }

    }
    return -1;

    }
```

```
44. }
```

Output:

```
Enter the item which you want to search
20
Item found at location 3
```

Python

```
1.  def binarySearch(arr,beg,end,item):
2.      if end >= beg:
3.          mid = int((beg+end)/2)
4.          if arr[mid] == item :
5.              return mid+1
6.          elif arr[mid] < item :
7.              return binarySearch(arr,mid+1,end,item)
8.          else:
9.              return binarySearch(arr,beg,mid-1,item)
10.     return -1
11.
12.
13. arr=[16, 19, 20, 23, 45, 56, 78, 90, 96, 100];
14. item = int(input("Enter the item which you want to search ?"))
15. location = -1;
16. location = binarySearch(arr,0,9,item);
17. if location != -1:
18.     print("Item found at location %d" %(location))
19. else:
20.     print("Item not found")
```

Output:

```
Enter the item which you want to search ?
96
Item found at location 9

Enter the item which you want to search ?
101
Item not found
```

Binary Search function using Iteration

```
1.  int binarySearch(int a[], int beg, int end, int item)
2.  {
3.      int mid;
4.      while(end >= beg)
5.      {
6.          mid = (beg + end)/2;
7.          if(a[mid] == item)
8.          {
9.              return mid+1;
10.         }
11.         else if(a[mid] < item)
12.         {
13.             beg = mid + 1;
14.         }
15.         else
16.         {
17.             end = mid - 1;
18.         }
19.
20.     }
21.     return -1;
22. }
```

Bubble Sort

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

Algorithm :

- **Step 1**: Repeat Step 2 For i = 0 to N-1
- **Step 2**: Repeat For J = i + 1 to N - I
- **Step 3**: IF A[J] > A[i] SWAP A[J] and A[i] [END OF INNER LOOP] [END OF OUTER LOOP
- **Step 4**: EXIT

Complexity

Scenario	Complexity
Space	O(1)
Worst case running time	$O(n^2)$
Average case running time	O(n)
Best case running time	$O(n^2)$

C Program

```c
#include<stdio.h>
void main ()
{
    int i, j,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(i = 0; i<10; i++)
   {
      for(j = i+1; j<10; j++)
       {
          if(a[j] > a[i])
           {
              temp = a[i];
               a[i] = a[j];
              a[j] = temp;
           }
      }
   }
   printf("Printing Sorted Element List ...\n");
   for(i = 0; i<10; i++)
   {
       printf("%d\n",a[i]);
   }
```

```
23. }
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101
```

C++ Program

```
1.  #include<iostream>
2.  using namespace std;
3.  int main ()
4.  {
5.      int i, j,temp;
6.      int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
7.      for(i = 0; i<10; i++)
8.      {
9.          for(j = i+1; j<10; j++)
10.         {
11.             if(a[j] < a[i])
12.             {
13.                 temp = a[i];
14.                 a[i] = a[j];
15.                 a[j] = temp;
16.             }
17.         }
18.     }
19.     cout <<"Printing Sorted Element List ...\n";
20.     for(i = 0; i<10; i++)
21.     {
22.         cout <<a[i]<<"\n";
23.     }
24.     return 0;
25. }
```

Output:

```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```

Java Program

```
1.  public class BubbleSort {
2.      public static void main(String[] args) {
3.      int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
4.      for(int i=0;i<10;i++)
5.      {
6.          for (int j=0;j<10;j++)
7.          {
8.              if(a[i]<a[j])
```

```
9.              {
10.               int temp = a[i];
11.                a[i]=a[j];
12.               a[j] = temp;
13.             }
14.         }
15.     }
16.     System.out.println("Printing Sorted List ...");
17.     for(int i=0;i<10;i++)
18.     {
19.         System.out.println(a[i]);
20.     }
21. }
22. }
```

Output:

```
Printing Sorted List . . .
7
9
10
12
23
34
34
44
78
101
```

C# Program

```
1.  using System;
2.
3.  public class Program
4.  {
5.      public static void Main()
6.      {
7.          int i, j,temp;
8.      int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
9.      for(i = 0; i<10; i++)
10.     {
11.         for(j = i+1; j<10; j++)
12.         {
13.             if(a[j] > a[i])
14.             {
15.                 temp = a[i];
16.                a[i] = a[j];
17.                 a[j] = temp;
18.             }
19.         }
20.     }
21.     Console.WriteLine("Printing Sorted Element List ...\n");
22.     for(i = 0; i<10; i++)
23.     {
24.         Console.WriteLine(a[i]);
25.     }
26.     }
27. }
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101
```

Python Program

```
a=[10, 9, 7, 101, 23, 44, 12, 78, 34, 23]
for i in range(0,len(a)):
    for j in range(i+1,len(a)):
        if a[j]<a[i]:
            temp = a[j]
            a[j]=a[i]
            a[i]=temp
print("Printing Sorted Element List...")
for i in a:
    print(i)
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101
```

Rust Program

```
fn main()
{
    let mut temp;
 let mut a: [i32; 10] = [10, 9, 7, 101, 23, 44, 12, 78, 34, 23];
    for i in 0..10
    {
        for j in (i+1)..10
        {
            if a[j] < a[i]
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
 println!("Printing Sorted Element List ...\n");
 for i in &a
 {
     println!("{} ",i);
 }
}
```

Output:

```
Printing Sorted Element List . . .
7
9
10
```

```
12
23
34
34
44
78
101
4
```

JavaScript Program

```
<html>
<head>
<title>
Bubble Sort
</title>
</head>
<body>
<script>
    var a = [10, 9, 7, 101, 23, 44, 12, 78, 34, 23];
   for(i=0;i<10;i++)
   {
      for (j=0;j<10;j++)
      {
         if(a[i]<a[j])
          {
              temp = a[i];
              a[i]=a[j];
             a[j] = temp;
          }
      }
   }
   txt = "<br>";
   document.writeln("Printing Sorted Element List ..."+txt);
   for(i=0;i<10;i++)
   {
      document.writeln(a[i]);
       document.writeln(txt);
   }
   </script>
   </body>
</html>
```

Output:

```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```

PHP Program

```
<html>
<head>
<title>Bubble Sort</title>
</head>
<body>
<?php
    $a = array(10, 9, 7, 101, 23, 44, 12, 78, 34, 23);
    for($i=0;$i<10;$i++)
```

```
9.      {
10.        for ($j=0;$j<10;$j++)
11.        {
12.           if($a[$i]<$a[$j])
13.           {
14.              $temp = $a[$i];
15.              $a[$i]=$a[$j];
16.             $a[$j] = $temp;
17.           }
18.        }
19.     }
20.     echo "Printing Sorted Element List ...\n";
21.     for($i=0;$i<10;$i++)
22.     {
23.         echo $a[$i];
24.        echo "\n";
25.     }
26. ?>
27. </body>
28. </html>
```

Output:

```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```

Bucket Sort

Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. Buckets are sorted individually by using different sorting algorithm.

Complexity of Bucket Sort

Algorithm	Complexity
Space	O(1)
Worst Case	$O(n^2)$
Best Case	Ω(n+k)
Average Case	θ(n+k)

Algorithm

- Step 1 START
- Step 2 Set up an array of initially empty "buckets".
- Step 3 Scatter: Go over the original array, putting each object in its bucket.
- Step 4 Sort each non-empty bucket.
- Step 5 Gather: Visit the buckets in order and put all elements back into the original array.
- Step 6 STOP

Program

```
1. #include <stdio.h>
```

```
2.  void Bucket_Sort(int array[], int n)
3.  {
4.      int i, j;
5.      int count[n];
6.      for (i = 0; i < n; i++)
7.          count[i] = 0;
8.
9.      for (i = 0; i < n; i++)
10.         (count[array[i]])++;
11.
12.     for (i = 0, j = 0; i < n; i++)
13.         for(; count[i] > 0; (count[i])--)
14.             array[j++] = i;
15. }
16. /* End of Bucket_Sort() */
17.
18. /* The main() begins */
19. int main()
20. {
21.     int array[100], i, num;
22.
23.     printf("Enter the size of array : ");
24.     scanf("%d", &num);
25.     printf("Enter the %d elements to be sorted:\n",num);
26.     for (i = 0; i < num; i++)
27.         scanf("%d", &array[i]);
28.     printf("\nThe array of elements before sorting : \n");
29.     for (i = 0; i < num; i++)
30.         printf("%d ", array[i]);
31.     printf("\nThe array of elements after sorting : \n");
32.     Bucket_Sort(array, num);
33.     for (i = 0; i < num; i++)
34.         printf("%d ", array[i]);
35.     printf("\n");
36.     return 0;
37. }
```

COMB SORT

Comb Sort is the advance form of Bubble Sort. Bubble Sort compares all the adjacent values while comb sort removes all the turtle values or small values near the end of the list.

Factors affecting comb sort are:

- It improves on bubble sort by using gap of size more than 1.
- Gap starts with large value and shrinks by the factor of 1.3.
- Gap shrinks till value reaches 1.

Complexity

Algorithm	Complexity
Worst Case Complexity	$O(n^2)$
Best Case Complexity	$\theta(n \log n)$
Average Case Complexity	$\Omega(n^2/2^p)$ where p is number of increments.
Worst Case Space Complexity	O(1)

Algorithm

- STEP 1 START
- STEP 2 Calculate the gap value if gap value==1 goto step 5 else goto step 3
- STEP 3 Iterate over data set and compare each item with gap item then goto step 4.
- STEP 4 Swap the element if require else goto step 2
- STEP 5 Print the sorted array.
- STEP 6 STOP

Program

```
#include <stdio.h>
#include <stdlib.h>
 int newgap(int gap)
{
    gap = (gap * 10) / 13;
    if (gap == 9 || gap == 10)
        gap = 11;
    if (gap < 1)
        gap = 1;
    return gap;
}

void combsort(int a[], int aSize)
{
   int gap = aSize;
   int temp, i;
   for (;;)
   {
      gap = newgap(gap);
      int swapped = 0;
      for (i = 0; i < aSize - gap; i++)
      {
         int j = i + gap;
         if (a[i] > a[j])
         {
            temp = a[i];
             a[i] = a[j];
            a[j] = temp;
             swapped  =  1;
         }
      }
      if (gap  ==  1 && !swapped)
         break;
   }
}
int main ()
{
   int n, i;
   int *a;
   printf("Please insert the number of elements to be sorted: ");
   scanf("%d", &n);       // The total number of elements
   a  =  (int *)calloc(n, sizeof(int));
   for (i = 0;i< n;i++)
   {
      printf("Input element %d :", i);
      scanf("%d", &a[i]); // Adding the elements to the array
```

```
47.    }
48.    printf("unsorted list");      // Displaying the unsorted array
49.    for(i = 0;i < n;i++)
50.    {
51.         printf("%d", a[i]);
52.    }
53.    combsort(a, n);
54.    printf("Sorted list:\n"); // Display the sorted array
55.    for(i = 0;i < n;i++)
56.    {
57.        printf("%d ", (a[i]));
58.    }
59.    return 0;
60. }
```

Counting Sort

It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.

Complexity

Time Complexity: O(n+k) is worst case where n is the number of element and k is the range of input.

Space Complexity: O(k) k is the range of input.

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n+k)	θ(n+k)	O(n+k)
Space Complexity			O(k)

Limitation of Counting Sort

- It is effective when range is not greater than number of object.
- It is not comparison based complexity.
- It is also used as sub-algorithm for different algorithm.
- It uses partial hashing technique to count the occurrence.
- It is also used for negative inputs.

Algorithm

- STEP 1 START
- STEP 2 Store the input array
- STEP 3 Count the key values by number of occurrence of object
- STEP 4 Update the array by adding previous key elements and assigning to objects
- STEP 5 Sort by replacing the object into new array and key= key-1
- STEP 6 STOP

Program

```
1. #include <stdio.h>
2. void counting_sort(int A[], int k, int n)
3. {
4.     int i, j;
5.     int B[15], C[100];
6.     for (i = 0; i <= k; i++)
```

```
7.          C[i] = 0;
8.      for (j = 1; j <= n; j++)
9.          C[A[j]] = C[A[j]] + 1;
10.     for (i = 1; i <= k; i++)
11.         C[i] = C[i] + C[i-1];
12.     for (j = n; j >= 1; j--)
13.     {
14.        B[C[A[j]]] = A[j];
15.         C[A[j]] = C[A[j]] - 1;
16.     }
17.     printf("The Sorted array is : ");
18.     for (i = 1; i <= n; i++)
19.         printf("%d ", B[i]);
20. }
21. /*  End of counting_sort()  */
22.
23. /*  The main() begins  */
24. int main()
25. {
26.     int n, k = 0, A[15], i;
27.     printf("Enter the number of input : ");
28.     scanf("%d", &n);
29.     printf("\nEnter the elements to be sorted :\n");
30.     for (i = 1; i <= n; i++)
31.     {
32.         scanf("%d", &A[i]);
33.         if (A[i] > k) {
34.             k = A[i];
35.         }
36.     }
37.     counting_sort(A, k, n);
38.     printf("\n");
39.     return 0;
40. }
```

Heap Sort

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

Complexity

Complexity	Best Case	Average Case	Worst case
Time Complexity	Ω(n log (n))	θ(n log (n))	O(n log (n))
Space Complexity			O(1)

Algorithm

HEAP_SORT(ARR, N)

- **Step 1**: [Build Heap H]
  Repeat for i=0 to N-1
  CALL INSERT_HEAP(ARR, N, ARR[i])
  [END OF LOOP]
- **Step 2**: Repeatedly Delete the root element
  Repeat while N > 0
  CALL Delete_Heap(ARR,N,VAL)
  SET N = N+1
  [END OF LOOP]
- **Step 3**: END

C Program

```
#include<stdio.h>
int temp;

void heapify(int arr[], int size, int i)
{
int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;

if (left < size && arr[left] >arr[largest])
largest = left;

if (right < size && arr[right] > arr[largest])
largest = right;

if (largest != i)
{
temp = arr[i];
    arr[i]= arr[largest];
    arr[largest] = temp;
heapify(arr, size, largest);
}
}

void heapSort(int arr[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
{
temp = arr[0];
    arr[0]= arr[i];
    arr[i] = temp;
heapify(arr, i, 0);
}
}

void main()
{
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
int i;
int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);
```

```
46.
47. printf("printing sorted elements\n");
48. for (i=0; i<size; ++i)
49. printf("%d\n",arr[i]);
50. }
```

Output:

```
printing sorted elements

1
1
2
2
2
3
4
10
23
100
```

Java Program

```
1.  #include<stdio.h>
2.  int temp;
3.
4.  void heapify(int arr[], int size, int i)
5.  {
6.  int largest = i;
7.  int left = 2*i + 1;
8.  int right = 2*i + 2;
9.
10. if (left < size && arr[left] >arr[largest])
11. largest = left;
12.
13. if (right < size && arr[right] > arr[largest])
14. largest = right;
15.
16. if (largest != i)
17. {
18. temp = arr[i];
19.     arr[i]= arr[largest];
20.     arr[largest] = temp;
21. heapify(arr, size, largest);
22. }
23. }
24.
25. void heapSort(int arr[], int size)
26. {
27. int i;
28. for (i = size / 2 - 1; i >= 0; i--)
29. heapify(arr, size, i);
30. for (i=size-1; i>=0; i--)
31. {
32. temp = arr[0];
33.     arr[0]= arr[i];
34.     arr[i] = temp;
35. heapify(arr, i, 0);
36. }
37. }
38.
39. void main()
40. {
41. int arr[] = {1, 10, 2, 3, 4, 1, 2, 100, 23, 2};
```

```
42. int i;
43. int size = sizeof(arr)/sizeof(arr[0]);
44.
45. heapSort(arr, size);
46.
47. printf("printing sorted elements\n");
48. for (i=0; i<size; ++i)
49. printf("%d\n",arr[i]);
50. }
```

Output:

```
printing sorted elements

1
1
2
2
2
3
4
10
23
100
```

C# program

```
1.  using System;
2.  public class HeapSorting {
3.  static void heapify(int[] arr, int size, int i)
4.  {
5.  int largest = i;
6.  int left = 2*i + 1;
7.  int right = 2*i + 2;
8.  int temp;
9.  if (left < size && arr[left] > arr[largest])
10. largest = left;
11.
12. if (right < size && arr[right] > arr[largest])
13. largest = right;
14.
15. if (largest != i)
16. {
17. temp = arr[i];
18.     arr[i]= arr[largest];
19.     arr[largest] = temp;
20. heapify(arr, size, largest);
21. }
22. }
23.
24. static void heapSort(int[] arr, int size)
25. {
26. int i;
27. int temp;
28. for (i = size / 2 - 1; i >= 0; i--)
29. heapify(arr, size, i);
30. for (i=size-1; i>=0; i--)
31. {
32. temp = arr[0];
33.     arr[0]= arr[i];
34.     arr[i] = temp;
35. heapify(arr, i, 0);
36. }
37. }
```

```
38.
39. public void Main()
40. {
41. int[] arr = {1, 10, 2, 3, 4, 1, 2, 100, 23, 2};
42. int i;
43. heapSort(arr, 10);
44. Console.WriteLine("printing sorted elements");
45. for (i=0; i<10; ++i)
46. Console.WriteLine(arr[i]);
47. }
48. }
```

Output:

```
printing sorted elements

1
1
2
2
2
3
4
10
23
100
```

Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Complexity

Complexity	Best Case	Average Case	Worst Case
Time	Ω(n)	θ(n^2)	o(n^2)
Space			o(1)

Algorithm

- **Step 1**: Repeat Steps 2 to 5 for K = 1 to N-1
- **Step 2**: SET TEMP = ARR[K]
- **Step 3**: SET J = K - 1
- **Step 4**: Repeat while TEMP <=ARR[J]
  SET ARR[J + 1] = ARR[J]
  SET J = J - 1
  [END OF INNER LOOP]
- **Step 5**: SET ARR[J + 1] = TEMP
  [END OF LOOP]
- **Step 6**: EXIT

C Program

```c
#include<stdio.h>
void main ()
{
    int i,j, k,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    printf("\nprinting sorted elements...\n");
    for(k=1; k<10; k++)
    {
        temp = a[k];
       j= k-1;
        while(j>=0 && temp <= a[j])
        {
            a[j+1] = a[j];
           j = j-1;
        }
       a[j+1] = temp;
   }
    for(i=0;i<10;i++)
    {
        printf("\n%d\n",a[i]);
    }
}
```

Output:

```
Printing Sorted Elements . . .
7
9
10
12
23
23
34
44
78
101
```

C++ Program

```cpp
#include<iostream>
using namespace std;
int main ()
{
    int i,j, k,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    cout<<"\nprinting sorted elements...\n";
    for(k=1; k<10; k++)
    {
        temp = a[k];
        j= k-1;
        while(j>=0 && temp <= a[j])
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = temp;
    }
    for(i=0;i<10;i++)
    {
        cout <<a[i]<<"\n";
    }
}
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Java Program

```
1.  public class InsertionSort {
2.  public static void main(String[] args) {
3.      int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
4.      for(int k=1; k<10; k++)
5.      {
6.          int temp = a[k];
7.          int j= k-1;
8.          while(j>=0 && temp <= a[j])
9.          {
10.             a[j+1] = a[j];
11.             j = j-1;
12.         }
13.         a[j+1] = temp;
14.     }
15.     System.out.println("printing sorted elements ...");
16.     for(int i=0;i<10;i++)
17.     {
18.         System.out.println(a[i]);
19.     }
20. }
21. }
```

Output:

```
Printing sorted elements . . .
7
9
10
12
23
23
34
44
78
101
```

C# Program

```
1.  using System;
2.
3.  public class Program
4.  {
5.
6.      public static void Main()
7.      {
8.          int i,j, k,temp;
9.          int[] a = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
10.     Console.WriteLine("\nprinting sorted elements...\n");
11.     for(k=1; k<10; k++)
12.     {
13.         temp = a[k];
14.         j= k-1;
```

```
15.         while(j>=0 && temp <= a[j])
16.         {
17.             a[j+1] = a[j];
18.            j = j-1;
19.         }
20.         a[j+1] = temp;
21.     }
22.     for(i=0;i<10;i++)
23.     {
24.         Console.WriteLine(a[i]);
25.     }
26. }
27. }
```

Output:

```
printing sorted elements . . .
7
9
10
12
23
23
34
44
78
101
```

Python Program

```
1.  a=[10, 9, 7, 101, 23, 44, 12, 78, 34, 23]
2.  for k in range(1,10):
3.      temp = a[k]
4.      j = k-1
5.      while j>=0 and temp <=a[j]:
6.          a[j+1]=a[j]
7.          j = j-1
8.      a[j+1] = temp
9.  print("printing sorted elements...")
10. for i in a:
11.     print(i)
```

Output:

```
printing sorted elements . . .
7
9
10
12
23
23
34
44
78
101
```

Swift Program

```
1.  import Foundation
2.  import Glibc
3.      var a = [10, 9, 7, 101, 23, 44, 12, 78, 34, 23];
4.      print("\nprinting sorted elements...\n");
5.      for k in 1...9
6.      {
7.          let temp = a[k];
8.          var j = k-1;
9.          while j>=0 && temp <= a[j]
10.         {
```

```
            a[j+1] = a[j];
          j = j-1;
       }

       a[j+1] = temp;
   }
   for i in a
   {
       print(i);
   }
```

Output:

```
printing sorted elements...

7
9
10
12
23
23
34
44
78
101
```

JavaScript Program

```
<html>
<head>
<title>
   Insertion Sort
</title>
   </head>
   <body>
      <script>
            var txt = "<br>";
        var a = [10, 9, 7, 101, 23, 44, 12, 78, 34, 23];
         document.writeln("printing sorted elements ... "+txt);
        for(k=0;k<10;k++)
         {
           var temp = a[k]
            j=k-1;
           while (j>=0 && temp <= a[j])
            {
              a[j+1] = a[j];
                jj = j-1;
           }
            a[j+1] = temp;
       }

       for(i=0;i<10;i++)
        {
           document.writeln(a[i]);
            document.writeln(txt);
       }
      </script>
   </body>
</html>
```

Output:

```
printing sorted elements ...
7
9
```

```
10
12
23
23
34
44
78
101
```

PHP Program

```
1.  <html>
2.  <head>
3.  <title>Insertion Sort</title>
4.  </head>
5.  <body>
6.  <?php
7.   $a = array(10, 9, 7, 101, 23, 44, 12, 78, 34, 23);
8.            echo("printing sorted elements ... \n");
9.             for($k=0;$k<10;$k++)
10.           {
11.               $temp = $a[$k];
12.              $j=$k-1;
13.               while ($j>=0 && $temp <= $a[$j])
14.              {
15.                  $a[$j+1] = $a[$j];
16.                $j = $j-1;
17.               }
18.              $a[$j+1] = $temp;
19.           }
20.          for($i=0;$i<10;$i++)
21.           {
22.              echo $a[$i];
23.               echo "\n";
24.          }
25. ?>
26. </body>
27. </html>
```

Output:

```
printing sorted elements ...
7
9
10
12
23
23
34
44
78
101
```

Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

Complexity

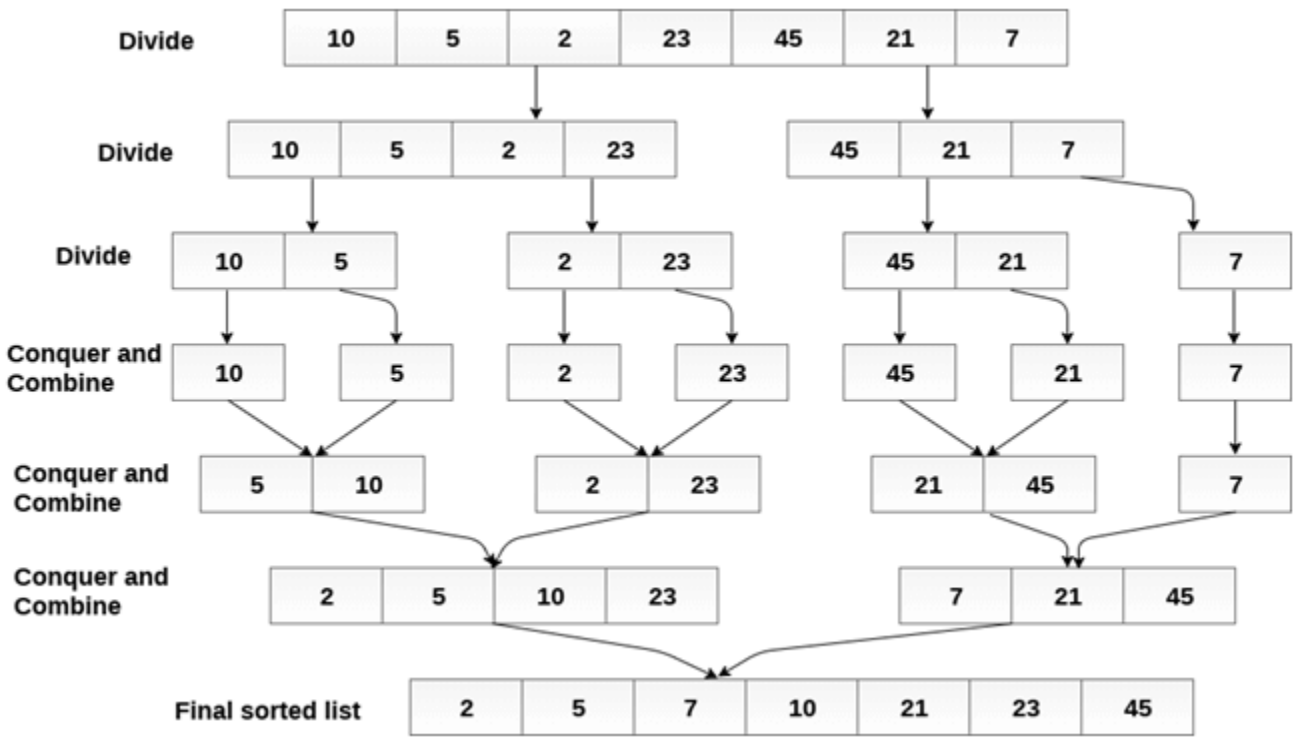
Complexity	Best case	Average Case	Worst Case

Time Complexity	O(n log n)	O(n log n)	O(n log n)
Space Complexity			O(n)

Example :

Consider the following array of 7 elements. Sort the array by using merge sort.

1. A = {10, 5, 2, 23, 45, 21, 7}



Algorithm

- **Step 1**: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
- **Step 2**: Repeat while (I <= MID) AND (J<=END)
  IF ARR[I] < ARR[J]
  SET TEMP[INDEX] = ARR[I]
  SET I = I + 1
  ELSE
  SET TEMP[INDEX] = ARR[J]
  SET J = J + 1
  [END OF IF]
  SET INDEX = INDEX + 1
  [END OF LOOP]
  Step 3: [Copy the remaining elements of right sub-array, if any]
  IF I > MID
  Repeat while J <= END
  SET TEMP[INDEX] = ARR[J]
  SET INDEX = INDEX + 1, SET J = J + 1
  [END OF LOOP]
  [Copy the remaining elements of left sub-array, if any]
  ELSE
  Repeat while I <= MID
  SET TEMP[INDEX] = ARR[I]
  SET INDEX = INDEX + 1, SET I = I + 1
  [END OF LOOP]
  [END OF IF]
- **Step 4**: [Copy the contents of TEMP back to ARR] SET K = 0
- **Step 5**: Repeat while K < INDEX
  SET ARR[K] = TEMP[K]

SET K = K + 1
[END OF LOOP]

- **Step 6**: Exit

MERGE_SORT(ARR, BEG, END)

- **Step 1**: IF BEG < END
SET MID = (BEG + END)/2
CALL MERGE_SORT (ARR, BEG, MID)
CALL MERGE_SORT (ARR, MID + 1, END)
MERGE (ARR, BEG, MID, END)
[END OF IF]
- **Step 2**: END

C Program

```
1. #include<stdio.h>
2. void mergeSort(int[],int,int);
3. void merge(int[],int,int,int);
4. void main ()
5. {
6.     int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
7.     int i;
8.     mergeSort(a,0,9);
9.     printf("printing the sorted elements");
10.    for(i=0;i<10;i++)
11.    {
12.        printf("\n%d\n",a[i]);
13.    }
14.
15. }
16. void mergeSort(int a[], int beg, int end)
17. {
18.    int mid;
19.    if(beg<end)
20.    {
21.        mid = (beg+end)/2;
22.        mergeSort(a,beg,mid);
23.        mergeSort(a,mid+1,end);
24.        merge(a,beg,mid,end);
25.    }
26. }
27. void merge(int a[], int beg, int mid, int end)
28. {
29.    int i=beg,j=mid+1,k,index = beg;
30.    int temp[10];
31.    while(i<=mid && j<=end)
32.    {
33.        if(a[i]<a[j])
34.        {
35.            temp[index] = a[i];
36.            i = i+1;
37.        }
38.        else
39.        {
40.            temp[index] = a[j];
41.            j = j+1;
42.        }
43.        index++;
```

```
44.    }
45.    if(i>mid)
46.    {
47.        while(j<=end)
48.        {
49.            temp[index] = a[j];
50.            index++;
51.            j++;
52.        }
53.    }
54.    else
55.    {
56.        while(i<=mid)
57.        {
58.            temp[index] = a[i];
59.            index++;
60.            i++;
61.        }
62.    }
63.    k = beg;
64.    while(k<index)
65.    {
66.        a[k]=temp[k];
67.        k++;
68.    }
69. }
```

Output:

```
printing the sorted elements

7
9
10
12
23
23
34
44
78
101
```

Java Program

```
1.  public class MyMergeSort
2.  {
3.  void merge(int arr[], int beg, int mid, int end)
4.  {
5.
6.  int l = mid - beg + 1;
7.  int r = end - mid;
8.
9.  intLeftArray[] = new int [l];
10. intRightArray[] = new int [r];
11.
12. for (int i=0; i<l; ++i)
13. LeftArray[i] = arr[beg + i];
14.
15. for (int j=0; j<r; ++j)
16. RightArray[j] = arr[mid + 1+ j];
17.
18.
19. int i = 0, j = 0;
20. int k = beg;
```

```
21. while (i<l&&j<r)
22. {
23. if (LeftArray[i] <= RightArray[j])
24. {
25. arr[k] = LeftArray[i];
26. i++;
27. }
28. else
29. {
30. arr[k] = RightArray[j];
31. j++;
32. }
33. k++;
34. }
35. while (i<l)
36. {
37. arr[k] = LeftArray[i];
38. i++;
39. k++;
40. }
41.
42. while (j<r)
43. {
44. arr[k] = RightArray[j];
45. j++;
46. k++;
47. }
48. }
49.
50. void sort(int arr[], int beg, int end)
51. {
52. if (beg<end)
53. {
54. int mid = (beg+end)/2;
55. sort(arr, beg, mid);
56. sort(arr , mid+1, end);
57. merge(arr, beg, mid, end);
58. }
59. }
60. public static void main(String args[])
61. {
62. intarr[] = {90,23,101,45,65,23,67,89,34,23};
63. MyMergeSort ob = new MyMergeSort();
64. ob.sort(arr, 0, arr.length-1);
65.
66. System.out.println("\nSorted array");
67. for(int i =0; i<arr.length;i++)
68. {
69.     System.out.println(arr[i]+"");
70. }
71. }
72. }
```

Output:

```
Sorted array
23
23
23
34
```

```
45
65
67
89
90
101
```

C# Program

```
1.  using System;
2.  public class MyMergeSort
3.  {
4.  void merge(int[] arr, int beg, int mid, int end)
5.  {
6.
7.  int l = mid - beg + 1;
8.  int r = end - mid;
9.          int i,j;
10.
11. int[] LeftArray = new int [l];
12. int[] RightArray = new int [r];
13.
14. for (i=0; i<l; ++i)
15. LeftArray[i] = arr[beg + i];
16.
17. for (j=0; j<r; ++j)
18. RightArray[j] = arr[mid + 1+ j];
19.
20.
21. i = 0; j = 0;
22. int k = beg;
23. while (i < l && j < r)
24. {
25. if (LeftArray[i] <= RightArray[j])
26. {
27. arr[k] = LeftArray[i];
28. i++;
29. }
30. else
31. {
32. arr[k] = RightArray[j];
33. j++;
34. }
35. k++;
36. }
37. while (i < l)
38. {
39. arr[k] = LeftArray[i];
40. i++;
41. k++;
42. }
43.
44. while (j < r)
45. {
46. arr[k] = RightArray[j];
47. j++;
48. k++;
49. }
50. }
51.
52. void sort(int[] arr, int beg, int end)
```

```
53. {
54. if (beg < end)
55. {
56. int mid = (beg+end)/2;
57. sort(arr, beg, mid);
58. sort(arr , mid+1, end);
59. merge(arr, beg, mid, end);
60. }
61. }
62. public static void Main()
63. {
64. int[] arr = {90,23,101,45,65,23,67,89,34,23};
65. MyMergeSort ob = new MyMergeSort();
66. ob.sort(arr, 0, 9);
67.
68. Console.WriteLine("\nSorted array");
69. for(int i =0; i<10;i++)
70. {
71.     Console.WriteLine(arr[i]+"");
72. }
73. }
74. }
```

Output:

```
Sorted array
23
23
23
34
45
65
67
89
90
101
```

Quick Sort

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. left = 0, loc = 0, en d = n - 1, where n is the length of the array.
2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.

   Ensure that, a[loc] is less than a[right].

   1. If this is the case, then continue with the comparison until right becomes equal to the loc.
   2. If a[loc] > a[right], then swap the two values. And go to step 3.
   3. Set, loc = right

1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]
   1. if this is the case, then continue with the comparison until loc becomes equal to left.
   2. [loc] < a[right], then swap the two values and go to step 2.
   3. Set, loc = left.

Complexity

Complexity	Best Case	Average Case	Worst Case

Time Complexity	O(n) for 3 way partition or O(n log n) simple partition	O(n log n)	$O(n^2)$
Space Complexity			O(log n)

Algorithm

PARTITION (ARR, BEG, END, LOC)

- **Step 1**: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
- **Step 2**: Repeat Steps 3 to 6 while FLAG =
- **Step 3**: Repeat while ARR[LOC] <=ARR[RIGHT] AND LOC != RIGHT
  SET RIGHT = RIGHT - 1
  [END OF LOOP]
- **Step 4**: IF LOC = RIGHT
  SET FLAG = 1
  ELSE IF ARR[LOC] > ARR[RIGHT]
  SWAP ARR[LOC] with ARR[RIGHT]
  SET LOC = RIGHT
  [END OF IF]
- **Step 5**: IF FLAG = 0
  Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
  SET LEFT = LEFT + 1
  [END OF LOOP]
- **Step 6**:IF LOC = LEFT
  SET FLAG = 1
  ELSE IF ARR[LOC] < ARR[LEFT]
  SWAP ARR[LOC] with ARR[LEFT]
  SET LOC = LEFT
  [END OF IF]
  [END OF IF]
- **Step 7**: [END OF LOOP]
- **Step 8**: END

QUICK_SORT (ARR, BEG, END)

- **Step 1**: IF (BEG < END)
  CALL PARTITION (ARR, BEG, END, LOC)
  CALL QUICKSORT(ARR, BEG, LOC - 1)
  CALL QUICKSORT(ARR, LOC + 1, END)
  [END OF IF]
- **Step 2**: END

C Program

```c
#include <stdio.h>
int partition(int a[], int beg, int end);
void quickSort(int a[], int beg, int end);
void main()
{
    int i;
    int arr[10]={90,23,101,45,65,23,67,89,34,23};
    quickSort(arr, 0, 9);
    printf("\n The sorted array is: \n");
    for(i=0;i<10;i++)
    printf(" %d\t", arr[i]);
```

```
12. }
13. int partition(int a[], int beg, int end)
14. {
15.
16.    int left, right, temp, loc, flag;
17.    loc = left = beg;
18.    right = end;
19.    flag = 0;
20.    while(flag != 1)
21.    {
22.       while((a[loc] <= a[right]) && (loc!=right))
23.        right--;
24.       if(loc==right)
25.        flag =1;
26.       else if(a[loc]>a[right])
27.       {
28.          temp = a[loc];
29.           a[loc] = a[right];
30.          a[right] = temp;
31.           loc = right;
32.       }
33.        if(flag!=1)
34.       {
35.           while((a[loc] >= a[left]) && (loc!=left))
36.          left++;
37.           if(loc==left)
38.          flag =1;
39.           else if(a[loc] <a[left])
40.          {
41.              temp = a[loc];
42.             a[loc] = a[left];
43.              a[left] = temp;
44.             loc = left;
45.           }
46.       }
47.    }
48.    return loc;
49. }
50. void quickSort(int a[], int beg, int end)
51. {
52.
53.    int loc;
54.    if(beg<end)
55.    {
56.       loc = partition(a, beg, end);
57.        quickSort(a, beg, loc-1);
58.       quickSort(a, loc+1, end);
59.    }
60. }
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
```

Java Program

```
1.  public class QuickSort {
2.  public static void main(String[] args) {
3.          int i;
4.         int[] arr={90,23,101,45,65,23,67,89,34,23};
5.          quickSort(arr, 0, 9);
6.         System.out.println("\n The sorted array is: \n");
7.          for(i=0;i<10;i++)
8.         System.out.println(arr[i]);
9.      }
10.    public static int partition(int a[], int beg, int end)
11.    {
12.
13.         int left, right, temp, loc, flag;
14.        loc = left = beg;
15.         right = end;
16.        flag = 0;
17.       while(flag != 1)
18.      {
19.          while((a[loc] <= a[right]) && (loc!=right))
20.         right--;
21.          if(loc==right)
22.         flag =1;
23.          elseif(a[loc]>a[right])
24.         {
25.              temp = a[loc];
26.             a[loc] = a[right];
27.              a[right] = temp;
28.             loc = right;
29.          }
30.         if(flag!=1)
31.          {
32.             while((a[loc] >= a[left]) && (loc!=left))
33.              left++;
34.             if(loc==left)
35.              flag =1;
36.             elseif(a[loc] <a[left])
37.              {
38.                 temp = a[loc];
39.                  a[loc] = a[left];
40.                a[left] = temp;
41.                 loc = left;
42.             }
43.          }
44.      }
45.       returnloc;
46.   }
47.    static void quickSort(int a[], int beg, int end)
48.   {
49.
50.       int loc;
51.        if(beg<end)
52.       {
53.           loc = partition(a, beg, end);
54.          quickSort(a, beg, loc-1);
55.           quickSort(a, loc+1, end);
```

```
56.         }
57.     }
58. }
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

C# Program

```
1.  using System;
2.  public class QueueSort{
3.  public static void Main() {
4.          int i;
5.           int[] arr={90,23,101,45,65,23,67,89,34,23};
6.          quickSort(arr, 0, 9);
7.           Console.WriteLine("\n The sorted array is: \n");
8.          for(i=0;i<10;i++)
9.           Console.WriteLine(arr[i]);
10.    }
11.    public static int partition(int[] a, int beg, int end)
12.    {
13.
14.       int left, right, temp, loc, flag;
15.        loc = left = beg;
16.       right = end;
17.        flag = 0;
18.       while(flag != 1)
19.       {
20.          while((a[loc] <= a[right]) && (loc!=right))
21.           right--;
22.          if(loc==right)
23.           flag =1;
24.          else if(a[loc]>a[right])
25.          {
26.             temp = a[loc];
27.              a[loc] = a[right];
28.             a[right] = temp;
29.              loc = right;
30.          }
31.           if(flag!=1)
32.          {
33.              while((a[loc] >= a[left]) && (loc!=left))
34.             left++;
35.              if(loc==left)
36.             flag =1;
37.              else if(a[loc] <a[left])
38.             {
39.                 temp = a[loc];
40.                a[loc] = a[left];
41.                 a[left] = temp;
42.                loc = left;
43.              }
44.          }
```

```
45.         }
46.         return loc;
47.     }
48.     static void quickSort(int[] a, int beg, int end)
49.     {
50.
51.         int loc;
52.         if(beg<end)
53.         {
54.             loc = partition(a, beg, end);
55.             quickSort(a, beg, loc-1);
56.             quickSort(a, loc+1, end);
57.         }
58.     }
59. }
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

Radix Sort

Radix sort processes the elements the same way in which the names of the students are sorted according to their alphabetical order. There are 26 radix in that case due to the fact that, there are 26 alphabets in English. In the first pass, the names are grouped according to the ascending order of the first letter of names.

In the second pass, the names are grouped according to the ascending order of the second letter. The same process continues until we find the sorted list of names. The bucket are used to store the names produced in each pass. The number of passes depends upon the length of the name with the maximum letter.

In the case of integers, radix sort sorts the numbers according to their digits. The comparisons are made among the digits of the number from LSB to MSB. The number of passes depend upon the length of the number with the most number of digits.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n+k)	θ(nk)	O(nk)
Space Complexity			O(n+k)

Example

Consider the array of length 6 given below. Sort the array by using Radix sort.

A = {10, 2, 901, 803, 1024}

Pass 1: (Sort the list according to the digits at 0's place)

10, 901, 2, 803, 1024.

Pass 2: (Sort the list according to the digits at 10's place)

02, 10, 901, 803, 1024

Pass 3: (Sort the list according to the digits at 100's place)

02, 10, 1024, 803, 901.

Pass 4: (Sort the list according to the digits at 1000's place)

02, 10, 803, 901, 1024

Therefore, the list generated in the step 4 is the sorted list, arranged from radix sort.

Algorithm

- **Step 1**:Find the largest number in ARR as LARGE
- **Step 2**: [INITIALIZE] SET NOP = Number of digits in LARGE
- **Step 3**: SET PASS =0
- **Step 4**: Repeat Step 5 while PASS <= NOP-1
- **Step 5**: SET I = 0 and INITIALIZE buckets
- **Step 6**:Repeat Steps 7 to 9 while I<n-1< li=""></n-1<>
- **Step 7**: SET DIGIT = digit at PASSth place in A[I]
- **Step 8**: Add A[I] to the bucket numbered DIGIT
- **Step 9**: INCREMENT bucket count for bucket numbered DIGIT [END OF LOOP]
- **Step 10**: Collect the numbers in the bucket [END OF LOOP]
- **Step 11**: END

C Program

```c
#include <stdio.h>
int largest(int a[]);
void radix_sort(int a[]);
void main()
{
    int i;
    int a[10]={90,23,101,45,65,23,67,89,34,23};
    radix_sort(a);
    printf("\n The sorted array is: \n");
    for(i=0;i<10;i++)
        printf(" %d\t", a[i]);
}

int largest(int a[])
{
    int larger=a[0], i;
    for(i=1;i<10;i++)
    {
        if(a[i]>larger)
        larger = a[i];
    }
    return larger;
}
void radix_sort(int a[])
{
    int bucket[10][10], bucket_count[10];
    int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
    larger = largest(a);
    while(larger>0)
    {
        NOP++;
        larger/=10;
    }
```

```
for(pass=0;pass<NOP;pass++) // Initialize the buckets
{
   for(i=0;i<10;i++)
    bucket_count[i]=0;
   for(i=0;i<10;i++)
    {
      // sort the numbers according to the digit at passth place
       remainder = (a[i]/divisor)%10;
      bucket[remainder][bucket_count[remainder]] = a[i];
       bucket_count[remainder] += 1;
    }
    // collect the numbers after PASS pass
    i=0;
    for(k=0;k<10;k++)
    {
        for(j=0;j<bucket_count[k];j++)
       {
            a[i] = bucket[k][j];
            i++;
       }
    }
    divisor *= 10;

}
}
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

Java Program

```
1.  public class Radix_Sort {
2.  public static void main(String[] args) {
3.          int i;
4.          Scanner sc = new Scanner(System.in);
5.          int[] a = {90,23,101,45,65,23,67,89,34,23};
6.          radix_sort(a);
7.          System.out.println("\n The sorted array is: \n");
8.          for(i=0;i<10;i++)
9.              System.out.println(a[i]);
10.     }
11.
12.     static int largest(inta[])
13.     {
14.         int larger=a[0], i;
15.         for(i=1;i<10;i++)
16.         {
17.             if(a[i]>larger)
18.             larger = a[i];
19.         }
20.         returnlarger;
21.     }
22.     static void radix_sort(inta[])
```

```
23.    {
24.        int bucket[][]=newint[10][10];
25.         int bucket_count[]=newint[10];
26.        int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
27.         larger = largest(a);
28.        while(larger>0)
29.         {
30.            NOP++;
31.             larger/=10;
32.        }
33.         for(pass=0;pass<NOP;pass++) // Initialize the buckets
34.        {
35.             for(i=0;i<10;i++)
36.            bucket_count[i]=0;
37.             for(i=0;i<10;i++)
38.            {
39.                 // sort the numbers according to the digit at passth place
40.                remainder = (a[i]/divisor)%10;
41.                 bucket[remainder][bucket_count[remainder]] = a[i];
42.                bucket_count[remainder] += 1;
43.            }
44.            // collect the numbers after PASS pass
45.             i=0;
46.            for(k=0;k<10;k++)
47.             {
48.                for(j=0;j<bucket_count[k];j++)
49.                 {
50.                    a[i] = bucket[k][j];
51.                     i++;
52.                }
53.            }
54.            divisor *= 10;
55.        }
56.    }
57. }
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

C# Program

```
1.  using System;
2.  public class Radix_Sort {
3.  public static void Main()
4.  {
5.          int i;
6.          int[] a = {90,23,101,45,65,23,67,89,34,23};
7.          radix_sort(a);
8.          Console.WriteLine("\n The sorted array is: \n");
9.          for(i=0;i<10;i++)
10.             Console.WriteLine(a[i]);
11.    }
12.
```

```
13.  static int largest(int[] a)
14.  {
15.      int larger=a[0], i;
16.     for(i=1;i<10;i++)
17.      {
18.         if(a[i]>larger)
19.          larger = a[i];
20.     }
21.      return larger;
22.  }
23.  static void radix_sort(int[] a)
24.  {
25.      int[,] bucket=new int[10,10];
26.     int[] bucket_count=new int[10];
27.      int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
28.     larger = largest(a);
29.      while(larger>0)
30.     {
31.         NOP++;
32.        larger/=10;
33.     }
34.     for(pass=0;pass<NOP;pass++) // Initialize the buckets
35.     {
36.        for(i=0;i<10;i++)
37.         bucket_count[i]=0;
38.        for(i=0;i<10;i++)
39.         {
40.            // sort the numbers according to the digit at passth place
41.             remainder = (a[i]/divisor)%10;
42.            bucket[remainder,bucket_count[remainder]] = a[i];
43.             bucket_count[remainder] += 1;
44.        }
45.         // collect the numbers after PASS pass
46.        i=0;
47.         for(k=0;k<10;k++)
48.        {
49.             for(j=0;j<bucket_count[k];j++)
50.            {
51.                 a[i] = bucket[k,j];
52.                i++;
53.             }
54.         }
55.          divisor *= 10;
56.      }
57.   }
58. }
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
- In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	1	2	10	3	90	43	56
2	2	2	3	10	90	43	56
3	2	2	3	10	90	43	56
4	4	2	3	10	43	90	56
5	5	**2**	**3**	**10**	**43**	**56**	**90**

Sorted A = {2, 3, 10, 43, 56, 90}

Complexity

Complexity	Best Case	Average Case	Worst Case
Time	Ω(n)	$\theta(n^2)$	$o(n^2)$
Space			o(1)

Algorithm

SELECTION SORT(ARR, N)

- **Step 1**: Repeat Steps 2 and 3 for K = 1 to N-1
- **Step 2**: CALL SMALLEST(ARR, K, N, POS)
- **Step 3**: SWAP A[K] with ARR[POS] [END OF LOOP]
- **Step 4**: EXIT

SMALLEST (ARR, K, N, POS)

- **Step 1**: [INITIALIZE] SET SMALL = ARR[K]
- **Step 2**: [INITIALIZE] SET POS = K
- **Step 3**: Repeat for J = K+1 to N -1 IF SMALL > ARR[J]

SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]

- **Step 4**: RETURN POS

C Program

```c
#include<stdio.h>
int smallest(int[],int,int);
void main ()
{
    int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int i,j,k,pos,temp;
    for(i=0;i<10;i++)
    {
        pos = smallest(a,10,i);
        temp = a[i];
        a[i]=a[pos];
        a[pos] = temp;
    }
    printf("\nprinting sorted elements...\n");
    for(i=0;i<10;i++)
    {
        printf("%d\n",a[i]);
    }
}
int smallest(int a[], int n, int i)
{
    int small,pos,j;
    small = a[i];
    pos = i;
    for(j=i+1;j<10;j++)
    {
        if(a[j]<small)
        {
            small = a[j];
            pos=j;
        }
    }
    return pos;
}
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

C++ Program

```cpp
#include<iostream>
using namespace std;
int smallest(int[],int,int);
int main ()
{
```

```
6.      int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
7.      int i,j,k,pos,temp;
8.      for(i=0;i<10;i++)
9.      {
10.         pos = smallest(a,10,i);
11.         temp = a[i];
12.         a[i]=a[pos];
13.         a[pos] = temp;
14.     }
15.     cout<<"\n printing sorted elements...\n";
16.     for(i=0;i<10;i++)
17.     {
18.         cout<<a[i]<<"\n";
19.     }
20.     return 0;
21. }
22. int smallest(int a[], int n, int i)
23. {
24.     int small,pos,j;
25.     small = a[i];
26.     pos = i;
27.     for(j=i+1;j<10;j++)
28.     {
29.         if(a[j]<small)
30.         {
31.             small = a[j];
32.             pos=j;
33.         }
34.     }
35.     return pos;
36. }
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Java Program

```
1.  public class SelectionSort {
2.  public static void main(String[] args) {
3.      int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
4.      int i,j,k,pos,temp;
5.      for(i=0;i<10;i++)
6.      {
7.          pos = smallest(a,10,i);
8.          temp = a[i];
9.          a[i]=a[pos];
10.         a[pos] = temp;
11.     }
12.     System.out.println("\nprinting sorted elements...\n");
13.     for(i=0;i<10;i++)
14.     {
15.         System.out.println(a[i]);
```

```
16.     }
17. }
18. public static int smallest(int a[], int n, int i)
19. {
20.     int small,pos,j;
21.     small = a[i];
22.     pos = i;
23.     for(j=i+1;j<10;j++)
24.     {
25.         if(a[j]<small)
26.         {
27.             small = a[j];
28.             pos=j;
29.         }
30.     }
31.     return pos;
32. }
33. }
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

C# Program

```
1.  using System;
2.  public class Program
3.  {
4.
5.
6.  public void Main() {
7.      int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
8.      int i,pos,temp;
9.      for(i=0;i<10;i++)
10.     {
11.         pos = smallest(a,10,i);
12.         temp = a[i];
13.         a[i]=a[pos];
14.         a[pos] = temp;
15.     }
16.     Console.WriteLine("\nprinting sorted elements...\n");
17.     for(i=0;i<10;i++)
18.     {
19.         Console.WriteLine(a[i]);
20.     }
21. }
22. public static int smallest(int[] a, int n, int i)
23. {
24.     int small,pos,j;
25.     small = a[i];
26.     pos = i;
27.     for(j=i+1;j<10;j++)
28.     {
```

```
29.         if(a[j]<small)
30.        {
31.             small = a[j];
32.            pos=j;
33.         }
34.     }
35.     return pos;
36. }
37. }
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Python Program

```
1.  def smallest(a,i):
2.      small = a[i]
3.      pos=i
4.      for j in range(i+1,10):
5.          if a[j] < small:
6.              small = a[j]
7.              pos = j
8.      return pos
9.
10. a=[10, 9, 7, 101, 23, 44, 12, 78, 34, 23]
11. for i in range(0,10):
12.     pos = smallest(a,i)
13.     temp = a[i]
14.     a[i]=a[pos]
15.     a[pos]=temp
16. print("printing sorted elements...")
17. for i in a:
18.     print(i)
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Rust Program

```
1.  fn main ()
2.  {
3.      let mut a: [i32;10] = [10, 9, 7, 101, 23, 44, 12, 78, 34, 23];
4.
5.      for i in 0..10
6.      {
7.          let mut small = a[i];
```

```
8.          let mut pos = i;
9.          for j in (i+1)..10
10.         {
11.         if a[j]<small
12.         {
13.             small = a[j];
14.            pos=j;
15.         }
16.     }
17.         let mut temp = a[i];
18.         a[i]=a[pos];
19.         a[pos] = temp;
20.     }
21.     println!("\nprinting sorted elements...\n");
22.     for i in &a
23.     {
24.         println!("{}",i);
25.     }
26. }
```

Output:

```
printing sorted elements...

7
9
10
12
23
23
34
44
78
101
```

JavaScript Program

```
1. <html>
2. <head>
3. <title>
4.     Selection Sort
5. </title>
6.     </head>
7.     <body>
8.     <script>
9. function smallest(a,  n, i)
10. {
11.
12.     var small = a[i];
13.     var pos = i;
14.     for(j=i+1;j<10;j++)
15.     {
16.         if(a[j]<small)
17.         {
18.             small = a[j];
19.             pos=j;
20.         }
21.     }
22.     return pos;
23. }
24. var a = [10, 9, 7, 101, 23, 44, 12, 78, 34, 23];
25.     for(i=0;i<10;i++)
26.     {
```

```
27.         pos = smallest(a,10,i);
28.        temp = a[i];
29.         a[i]=a[pos];
30.        a[pos] = temp;
31.    }
32.    document.writeln("printing sorted elements ...\n"+"<br>");
33.    for(i=0;i<10;i++)
34.    {
35.        document.writeln(a[i]+"<br>");
36.    }
37.        </script>
38.    </body>
39. </html>
```

Output:

```
printing sorted elements ...
7
9
10
12
23
23
34
44
78
101
```

PHP Program

```
1.  <html>
2.  <head>
3.  <title>Selection sort</title>
4.  </head>
5.  <body>
6.  <?php
7.  function smallest($a,  $n, $i)
8.  {
9.
10.    $small = $a[$i];
11.    $pos = $i;
12.    for($j=$i+1;$j<10;$j++)
13.    {
14.        if($a[$j]<$small)
15.        {
16.            $small = $a[$j];
17.            $pos=$j;
18.        }
19.    }
20.    return $pos;
21. }
22. $a = array(10, 9, 7, 101, 23, 44, 12, 78, 34, 23);
23.    for($i=0;$i<10;$i++)
24.    {
25.        $pos = smallest($a,10,$i);
26.        $temp = $a[$i];
27.        $a[$i]=$a[$pos];
28.        $a[$pos] = $temp;
29.    }
30.    echo "printing sorted elements ...\n";
31.    for($i=0;$i<10;$i++)
32.    {
33.        echo $a[$i];
```

```
34.         echo "\n";
35.     }
36. ?>
37. </body>
38. </html>
```

Output:

```
printing sorted elements ...
7
9
10
12
23
23
34
44
78
101
```

Shell Sort

Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions. In general, Shell sort performs the following steps.

- **Step 1**: Arrange the elements in the tabular form and sort the columns by using insertion sort.
- **Step 2**: Repeat Step 1; each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$\Omega(n \log(n))$	$\theta(n \log(n)^2)$	$O(n \log(n)^2)$
Space Complexity			O(1)

Algorithm

Shell_Sort(Arr, n)

- **Step 1**: SET FLAG = 1, GAP_SIZE = N
- **Step 2**: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
- **Step 3**:SET FLAG = 0
- **Step 4**:SET GAP_SIZE = (GAP_SIZE + 1) / 2
- **Step 5**:Repeat Step 6 for I = 0 to I < (N -GAP_SIZE)
- **Step 6**:IF Arr[I + GAP_SIZE] > Arr[I]
  SWAP Arr[I + GAP_SIZE], Arr[I]
  SET FLAG = 0
- **Step 7**: END

C Program

```
1. #include <stdio.h>
2. void shellsort(int arr[], int num)
3. {
4. int i, j, k, tmp;
5. for (i = num / 2; i > 0; i = i / 2)
6. {
7. for (j = i; j < num; j++)
8. {
9. for(k = j - i; k >= 0; k = k - i)
10. {
11. if (arr[k+i] >= arr[k])
12. break;
```

```
13. else
14. {
15. tmp = arr[k];
16. arr[k] = arr[k+i];
17. arr[k+i] = tmp;
18. }
19. }
20. }
21. }
22. }
23. int main()
24. {
25. int arr[30];
26. int k,  num;
27. printf("Enter total no. of elements : ");
28. scanf("%d", &num);
29. printf("\nEnter %d numbers: ", num);
30.
31. for (k =  0 ; k < num; k++)
32. {
33. scanf("%d", &arr[k]);
34. }
35. shellsort(arr, num);
36. printf("\n Sorted array is: ");
37. for (k = 0; k < num; k++)
38. printf("%d ", arr[k]);
39. return 0;
40. }
```

Output:

```
Enter total no. of elements : 6

Enter 6 numbers: 3
2
4
10
2
1

Sorted array is: 1 2 2 3 4 10
```

Java Program

```
1. import java.util.*;
2. public class Shell_Sort
3. {
4. static void shellsort(int[] arr, intnum)
5. {
6. int i, j, k, tmp;
7. for (i = num / 2; i> 0; i = i / 2)
8. {
9. for (j = i; j<num; j++)
10. {
11. for(k = j - i; k>= 0; k = k - i)
12. {
13. if (arr[k+i] >= arr[k])
14. break;
15. else
16. {
17. tmp = arr[k];
18. arr[k] = arr[k+i];
19. arr[k+i] = tmp;
```

```
20. }
21. }
22. }
23. }
24. }
25. public static void main(String[] args)
26. {
27. Scanner sc = new Scanner(System.in);
28. int arr[]= newint[30];
29. intk,  num;
30. System.out.println("Enter total no. of elements : ");
31. num = sc.nextInt();
32. for (k =  0 ; k<num; k++)
33. {
34. arr[k]=sc.nextInt();
35. }
36. shellsort(arr, num);
37. System.out.println("\n Sorted array is: ");
38. for (k = 0; k<num; k++)
39. System.out.println(arr[k]);
40. }
41. }
```

Output:

```
Enter total no. of elements : 6
3
2
4
10
2
1

Sorted array is: 1 2 2 3 4 10
```

C# Program

```
1. using System;
2. public class Shell_Sort
3. {
4. static void shellsort(int[] arr, int num)
5. {
6. int i, j, k, tmp;
7. for (i = num / 2; i > 0; i = i / 2)
8. {
9. for (j = i; j < num; j++)
10. {
11. for(k = j - i; k >= 0; k = k - i)
12. {
13. if (arr[k+i] >= arr[k])
14. break;
15. else
16. {
17. tmp = arr[k];
18. arr[k] = arr[k+i];
19. arr[k+i] = tmp;
20. }
21. }
22. }
23. }
```

```
24. }
25. public void Main()
26. {
27.     int[] arr=new int[10];
28. int k,  num;
29. Console.WriteLine("Enter total no. of elements : ");
30. num = Convert.ToInt32(Console.ReadLine());
31. for (k =  0 ; k < num; k++)
32. {
33. arr[k]=Convert.ToInt32(Console.ReadLine());
34. }
35. shellsort(arr, num);
36. Console.WriteLine("\n Sorted array is: ");
37. for (k = 0; k < num; k++)
38. Console.WriteLine(arr[k]);
39. }
40. }
```

Output:

```
Enter total no. of elements : 6

3
2
4
10
2
1

Sorted array is: 1 2 2 3 4 10
```

Bitonic Sort

Bitonic sort is a parallel sorting algorithm which performs O(n2 log n) comparisons. Although, the number of comparisons are more than that in any other popular sorting algorithm, It performs better for the parallel implementation because elements are compared in predefined sequence which must not be depended upon the data being sorted. The predefined sequence is called Bitonic sequence.

What is Bitonic Sequence ?

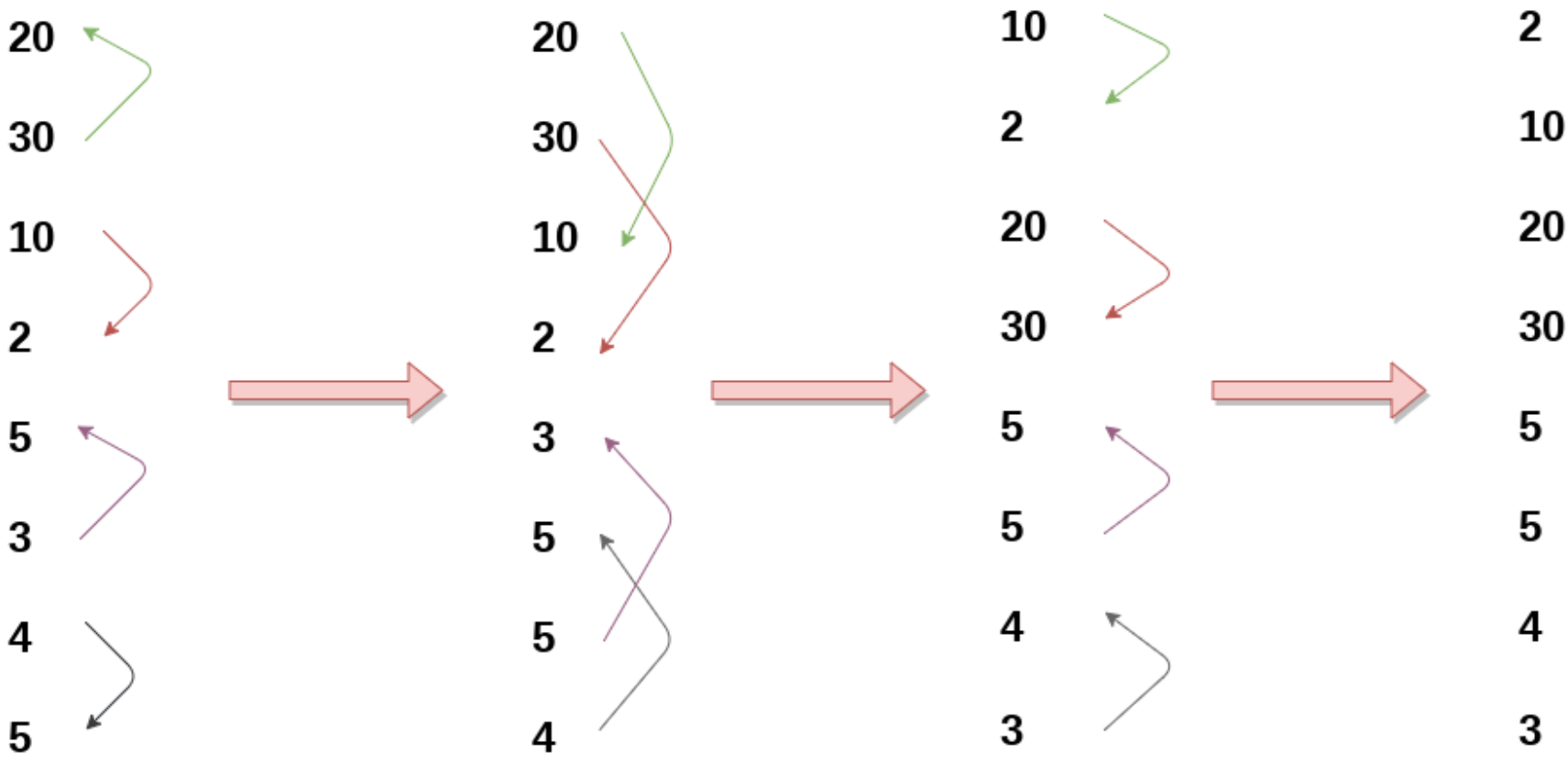
In order to understand Bitonic sort, we must understand Bitonic sequence. Bitonic sequence is the one in which, the elements first comes in increasing order then start decreasing after some particular index. An array A[0... i ... n-1] is called Bitonic if there exist an index i such that,

1. A[0] < A[1] < A[2] .... A[i-1] < A[i] > A[i+1] > A[i+2] > A[i+3] > ... >A[n-1]

where, 0 <= i <= n-1. A rotation of Bitonic sort is also Bitonic.

How to convert Random sequence to Bitonic sequence ?

Consider a sequence A[0 ... n-1] of n elements. First start constructing Bitonic sequence by using 4 elements of the sequence. Sort the first 2 elements in ascending order and the last 2 elements in descending order, concatenate this pair to form a Bitonic sequence of 4 elements. Repeat this process for the remaining pairs of element until we find a Bitonic sequence.



Constructing Bitonic Sequence

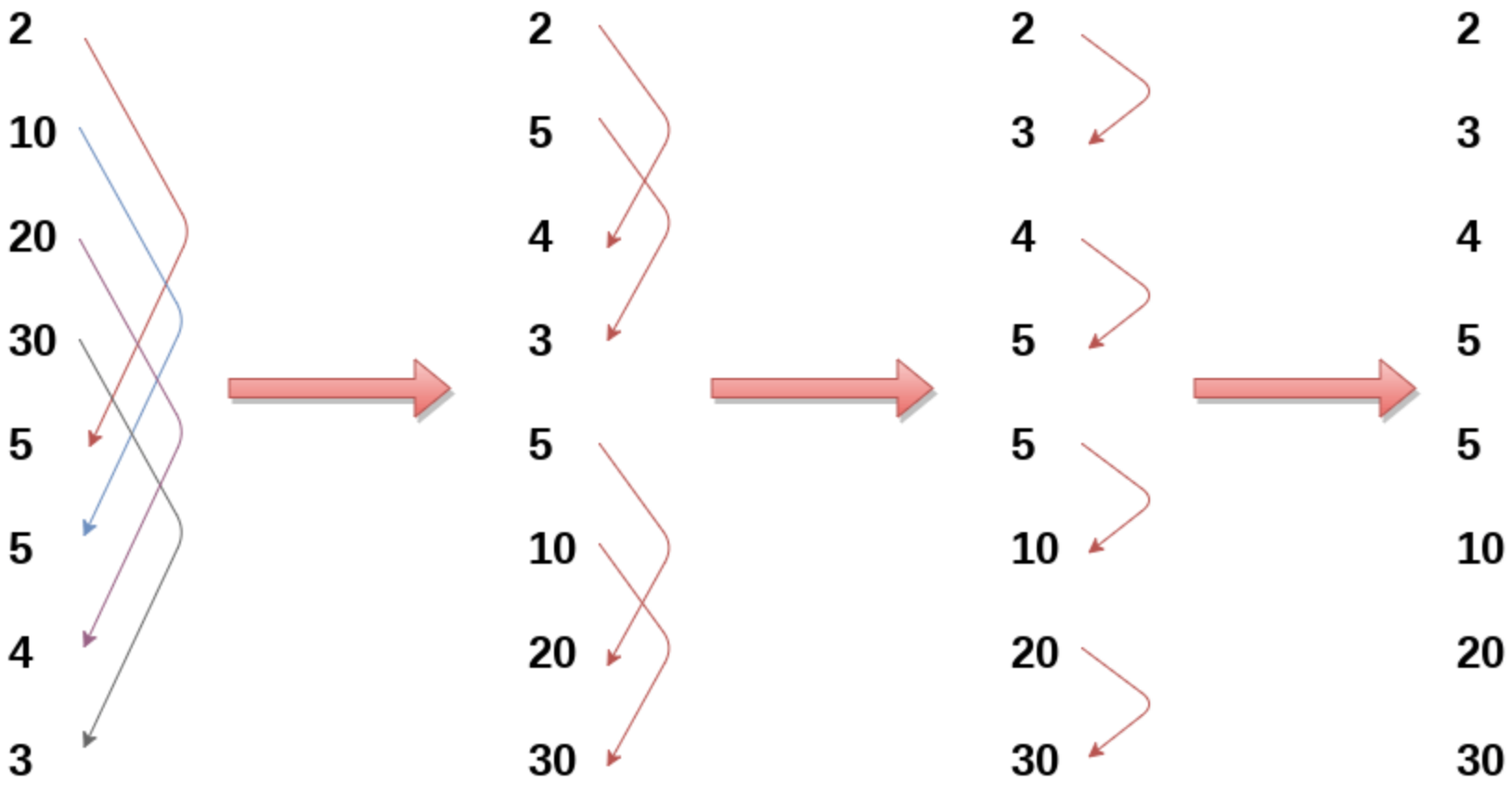
After this step, we get the Bitonic sequence for the given sequence as 2, 10, 20, 30, 5, 5, 4, 3.

Bitonic Sorting :

Bitonic sorting mainly involves the following basic steps.

1. Form a Bitonic sequence from the given random sequence which we have formed in the above step. We can consider it as the first step in our procedure. After this step, we get a sequence whose first half is sorted in ascending order while second step is sorted in descending order.
2. Compare first element of first half with the first element of the second half, then second element of the first half with the second element of the second half and so on. Swap the elements if any element in the second half is found to be smaller.
3. After the above step, we got all the elements in the first half smaller than all the elements in the second half. The compare and swap results into the two sequences of n/2 length each. Repeat the process performed in the second step recursively onto every sequence until we get single sorted sequence of length n.

The whole procedure involved in Bitonic sort is described in the following image.



Constructing Sorted Sequence

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log^2 n)$
Space Complexity			$O(n \log^2 n)$

C Program

```
1. //this program works when size of input is power of 2.
2. #include<stdio.h>
3. void exchange(int arr[], int i, int j, int d)
4. {
5.     int temp;
6.     if (d==(arr[i]>arr[j]))
7.     {
8.         temp = arr[i];
9.         arr[i] = arr[j];
10.        arr[j] = temp;
11.    }
12. }
13. void merge(int arr[], int l, int c, int d)
14. {
15.    int k,i;
16.    if (c>1)
17.    {
18.        k = c/2;
19.        for (i=l; i<l+k; i++)
20.            exchange(arr, i, i+k, d);
21.        merge(arr, l, k, d);
22.        merge(arr, l+k, k, d);
23.    }
24. }
25. void bitonicSort(int arr[],int l, int c, int d)
```

```
{
    int k;
    if (c>1)
    {
        k = c/2;
        bitonicSort(arr, l, k, 1);
        bitonicSort(arr, l+k, k, 0);
        merge(arr,l, c, d);
    }
}

void sort(int arr[], int n, int order)
{
    bitonicSort(arr,0, n, order);
}
int main()
{
    int arr[]= {1, 10, 2, 3, 1, 23, 45, 21};
    int n = sizeof(arr)/sizeof(arr[0]);
    int i;
    int order = 1;
    sort(arr, n, order);

    printf("Sorted array: \n");
    for (i=0; i<n; i++)
        printf("%d ", arr[i]);
}
```

Output:

```
Sorted array:
1 1 2 3 10 21 23 45
```

Java

```
1. //this program works when size of input is power of 2.
2. public class BitonicSort
3. {
4. static void exchange(int arr[], int i, int j, boolean d)
5. {
6.     int temp;
7.     if (d==(arr[i]>arr[j]))
8.     {
9.         temp = arr[i];
10.        arr[i] = arr[j];
11.        arr[j] = temp;
12.    }
13. }
14. static void merge(int arr[], int l, int c, boolean d)
15. {
16.    int k,i;
17.    if (c>1)
18.    {
19.        k = c/2;
20.        for (i=l; i<l+k; i++)
21.            exchange(arr, i, i+k, d);
22.        merge(arr, l, k, d);
23.        merge(arr, l+k, k, d);
24.    }
```

```
25. }
26. static void bitonicSort(int arr[],int l, int c, boolean d)
27. {
28.     int k;
29.     if (c>1)
30.     {
31.         k = c/2;
32.         bitonicSort(arr, l, k, true);
33.         bitonicSort(arr, l+k, k, false);
34.         merge(arr,l, c, d);
35.     }
36. }
37.
38. static void sort(int arr[], int n, boolean order)
39. {
40.     bitonicSort(arr,0, n, order);
41. }
42. public static void main(String[] args)
43. {
44.     int arr[]= {1, 10, 2, 3, 1, 23, 45, 21};
45.     int n = arr.length;
46.     int i;
47.     boolean order = true;
48.     sort(arr, n, order);
49.
50.     System.out.println("Sorted array: \n");
51.     for (i=0; i<n; i++)
52.         System.out.println(arr[i]);
53. }
54. }
```

Output:

```
Sorted array:

1       1       2       3       10      21      23      45
```

C#

```
1.  //this program works when size of input is power of 2.
2.  using System;
3.  public class BitonicSort
4.  {
5.  static void exchange(int[] arr, int i, int j, bool d)
6.  {
7.      int temp;
8.      if (d==(arr[i]>arr[j]))
9.      {
10.         temp = arr[i];
11.         arr[i] = arr[j];
12.         arr[j] = temp;
13.     }
14. }
15. static void merge(int[] arr, int l, int c, bool d)
16. {
17.     int k,i;
18.     if (c>1)
19.     {
20.         k = c/2;
21.         for (i=l; i<l+k; i++)
```

```
22.             exchange(arr, i, i+k, d);
23.         merge(arr, l, k, d);
24.         merge(arr, l+k, k, d);
25.     }
26. }
27. static void bitonicSort(int[] arr,int l, int c, bool d)
28. {
29.     int k;
30.     if (c>1)
31.     {
32.         k = c/2;
33.         bitonicSort(arr, l, k, true);
34.         bitonicSort(arr, l+k, k, false);
35.         merge(arr,l, c, d);
36.     }
37. }
38.
39. static void sort(int[] arr, int n, bool order)
40. {
41.     bitonicSort(arr,0, n, order);
42. }
43. public void Main()
44. {
45.     int[] arr= {1, 10, 2, 3, 1, 23, 45, 21};
46.     int n = arr.Length;
47.     int i;
48.     bool order = true;
49.     sort(arr, n, order);
50.
51.     Console.WriteLine("Sorted array: \n");
52.     for (i=0; i<n; i++)
53.         Console.WriteLine(arr[i]+"   ");
54. }
55. }
```

Cocktail Sort

Cocktail sort is the variation of Bubble Sort which traverses the list in both directions alternatively. It is different from bubble sort in the sense that, bubble sort traverses the list in forward direction only, while this algorithm traverses in forward as well as backward direction in one iteration.

Algorithm

In cocktail sort, one iteration consists of two stages:

1. The first stage loop through the array like bubble sort from left to right. The adjacent elements are compared and if left element is greater than the right element, then we swap those elements. The largest element of the list is placed at the end of the array in the forward pass.
2. The second stage loop through the array from the right most unsorted element to the left. The adjacent elements are compared and if right element is smaller than the left element then, we swap those elements. The smallest element of the list is placed at the beginning of the array in backward pass.

The process continues until the list becomes unsorted.

Example

Consider the array A : {8, 2, 3, 1, 9}. Sort the elements of the array using Cocktail sort.

Iteration 1:

Forward pass :

1. compare 8 with 2; 8>2 → swap(8,2)
2.
3. A={2,8,3,1,9}
4.
5. Compare 8 with 3; 8>3 → swap(8,3)
6.
7. A={2,3,8,1,9}
8.
9. Compare 8 with 1; 8 > 1 → swap(8,1)
10.
11. A = {2,3,1,8,9}
12.
13. Compare 8 with 9; 8<9 → **do** not swap

At the end of first forward pass: the largest element of the list is placed at the end.

1. A={2, 3, 1, 8, 9 }

Backward pass:

1. compare 8 with 1; 8> 1 → **do** not swap
2.
3. A={2, 3, 1, 8, 9 }
4. compare 1 with 3 ; 3>1 → swap(1,3)
5.
6. A={2, 1, 3, 8, 9 }
7.
8. compare 1 with 2 ; 2> 1 → swap(1,2)
9.
10. A={1, 2, 3, 8, 9}

At the end of first backward pass; the smallest element has been placed at the first index of the array.

If we have a look at the list produced in the first step; we can find that the list has already been sorted in ascending order but the algorithm will process itself completely.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$
Space Complexity			O(1)

C Program

```c
#include <stdio.h>
int temp;
void Cocktail(int a[], int n)
{
    int is_swapped = 1;
    int begin = 0,i;
    int end = n - 1;

    while (is_swapped) {
        is_swapped = 0;
     for (i = begin; i < end; ++i) {
            if (a[i] > a[i + 1]) {
             temp = a[i];
        a[i]=a[i+1];
         a[i+1]=temp;
```

```
        is_swapped = 1;
            }
        }
if (!is_swapped)
            break;
is_swapped = 0;
for (i = end - 1; i >= begin; --i) {
    if (a[i] > a[i + 1])
   {
          temp = a[i];
     a[i]=a[i+1];
     a[i+1]=temp;
     is_swapped = 1;
        }
     }
       ++begin;
   }
}

int main()
{
    int arr[] = {0, 10, 2, 8, 23, 1, 3, 45},i;
    int n = sizeof(arr) / sizeof(arr[0]);
    Cocktail(arr, n);
    printf("printing sorted array :\n");
     for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Output:

```
printing sorted array :
0 1 2 3 8 10 23 45
```

C++ Program

```
#include <iostream>
using namespace std;
int temp;
void Cocktail(int a[], int n)
{
    int is_swapped = 1;
    int begin = 0,i;
    int end = n - 1;

    while (is_swapped) {
        is_swapped = 0;
     for (i = begin; i < end; ++i) {
            if (a[i] > a[i + 1]) {
           temp = a[i];
       a[i]=a[i+1];
      a[i+1]=temp;
       is_swapped = 1;
          }
       }
if (!is_swapped)
            break;
```

```
22.  is_swapped = 0;
23.  for (i = end - 1; i >= begin; --i) {
24.      if (a[i] > a[i + 1])
25.     {
26.          temp = a[i];
27.       a[i]=a[i+1];
28.       a[i+1]=temp;
29.       is_swapped = 1;
30.         }
31.       }
32.         ++begin;
33.     }
34. }
35.
36. int main()
37. {
38.     int arr[] = {0, 10, 2, 8, 23, 1, 3, 45},i;
39.     int n = sizeof(arr) / sizeof(arr[0]);
40.     Cocktail(arr, n);
41.     cout <<"printing sorted array :\n";
42.      for (i = 0; i < n; i++)
43.         cout<<arr[i]<<" ";
44.     cout<<"\n";
45.     return 0;
46. }
```

Output:

```
printing sorted array :
0 1 2 3 8 10 23 45
```

Java Program

```
1.  public class CocktailSort
2.  {
3.  static int temp;
4.  static void Cocktail(int a[], int n)
5.  {
6.      boolean is_swapped = true;
7.      int begin = 0,i;
8.      int end = n - 1;
9.
10.     while (is_swapped) {
11.         is_swapped = false;
12.      for (i = begin; i < end; ++i) {
13.             if (a[i] > a[i + 1]) {
14.             temp = a[i];
15.         a[i]=a[i+1];
16.         a[i+1]=temp;
17.         is_swapped = true;
18.             }
19.         }
20.  if (!is_swapped)
21.             break;
22.  is_swapped = false;
23.  for (i = end - 1; i >= begin; --i) {
24.      if (a[i] > a[i + 1])
25.     {
26.          temp = a[i];
```

```
27.       a[i]=a[i+1];
28.       a[i+1]=temp;
29.       is_swapped = true;
30.         }
31.       }
32.         ++begin;
33.     }
34. }
35. public static void main(String[] args) {
36.
37.     int arr[] = {0, 10, 2, 8, 23, 1, 3, 45},i;
38.     int n = arr.length;
39.     Cocktail(arr, n);
40.     System.out.println("printing sorted array :\n");
41.      for (i = 0; i < n; i++)
42.         System.out.print(arr[i]+" ");
43.     System.out.println();
44.     }
45. }
```

Output:

```
printing sorted array :

0 1 2 3 8 10 23 45
```

C# Program

```
1.  using System;
2.  public class CocktailSort
3.  {
4.  static int temp;
5.  static void Cocktail(int[] a, int n)
6.  {
7.      Boolean is_swapped = true;
8.      int begin = 0,i;
9.      int end = n - 1;
10.
11.     while (is_swapped) {
12.         is_swapped = false;
13.      for (i = begin; i < end; ++i) {
14.             if (a[i] > a[i + 1]) {
15.                 temp = a[i];
16.         a[i]=a[i+1];
17.          a[i+1]=temp;
18.         is_swapped = true;
19.             }
20.         }
21. if (!is_swapped)
22.             break;
23. is_swapped = false;
24. for (i = end - 1; i >= begin; --i) {
25.     if (a[i] > a[i + 1])
26.    {
27.           temp = a[i];
28.       a[i]=a[i+1];
29.       a[i+1]=temp;
30.       is_swapped = true;
31.          }
32.       }
```

```
        ++begin;
    }
}
public void Main() {

    int[] arr = {0, 10, 2, 8, 23, 1, 3, 45};
    int n = arr.Length,i;
    Cocktail(arr, n);
    Console.WriteLine("printing sorted array :\n");
     for (i = 0; i < n; i++)
        Console.Write(arr[i]+" ");

    }
```

Output:

```
printing sorted array :

0 1 2 3 8 10 23 45
```

Python Program

```
def Cocktail(a,n):
    is_swapped = True;
    begin = 0
    end = n - 1;
    while is_swapped:
        is_swapped = False;
        for i in range(begin,end):
            if a[i] > a[i + 1]:
                temp = a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
                is_swapped = True;
        if not(is_swapped):
            break;
        is_swapped = False;
        for i in range(end-1,begin-1,-1):
            if a[i] > a[i + 1]:
                temp = a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
                is_swapped = True;
        ++begin;
arr = [0, 10, 2, 8, 23, 1, 3, 45];
n = len(arr);
Cocktail(arr, n);
print("printing sorted array :\n");
for i in range(0,n):
    print(arr[i]),
```

Output:

```
printing sorted array :

0 1 2 3 8 10 23 45
```

Rust Program

```
fn main()
{
    let mut a :[i32;8] = [0, 10, 2, 8, 23, 1, 3, 45];
```

```
4.      let mut is_swapped = true;
5.       let mut  begin = 0;
6.       let end = 7;
7.
8.       while is_swapped {
9.           is_swapped = false;
10.       for i in begin..end {
11.             if a[i] > a[i + 1] {
12.             let mut temp = a[i];
13.              a[i]=a[i+1];
14.             a[i+1]=temp;
15.              is_swapped = true;
16.             }
17.         }
18.     if !is_swapped
19.     {
20.             break;
21.     }
22.     is_swapped = false;
23.     for i in (begin..(end-1)).rev()
24.     {
25.         if a[i] > a[i + 1]
26.         {
27.           let mut temp = a[i];
28.          a[i]=a[i+1];
29.           a[i+1]=temp;
30.          is_swapped =true;
31.          }
32.       }
33.      begin=begin+1;
34.     }
35.     print!("printing sorted array :\n");
36.     for i in 0..7
37.     {
38.         print!("{}",a[i]);
39.     }
40. }
```

Output:

```
printing sorted array :
0 1 2 3 8 10 23 45
```

JavaScript Program

```
1.  <html>
2.  <head>
3.      <title>Cocktail Sort</title>
4.      </head>
5.          <body>
6.              <script>
7.
8.  function Cocktail( a, n)
9.  {
10.     var temp=0;
11.     var is_swapped = 1;
12.     var begin = 0;
13.     var end = n - 1;
14.
```

```
15.    while (is_swapped) {
16.       is_swapped = 0;
17.     for (i = begin; i < end; ++i) {
18.           if (a[i] > a[i + 1]) {
19.            temp = a[i];
20.       a[i]=a[i+1];
21.        a[i+1]=temp;
22.       is_swapped = 1;
23.           }
24.       }
25.    if (!is_swapped)
26.           break;
27.    is_swapped = 0;
28.    for (i = end - 1; i >= begin; --i) {
29.        if (a[i] > a[i + 1])
30.    {
31.          temp = a[i];
32.     a[i]=a[i+1];
33.        a[i+1]=temp;
34.       is_swapped = 1;
35.          }
36.       }
37.        beginbegin=begin+1;
38.     }
39. }
40.
41.     var txt = "<br>";
42.     var arr = [0, 10, 2, 8, 23, 1, 3, 45];
43.     var n = arr.length;
44.     Cocktail(arr, n);
45.     document.writeln("printing sorted array :\n"+txt);
46.      for (i = 0; i < n; i++)
47.      {
48.          document.writeln(arr[i]+"/xa0");
49.      }
50. </script>
51. </body>
52. </html>
```

Output:

```
printing sorted array :
0
1 2 3 8 10 23 45
```

Cycle Sort

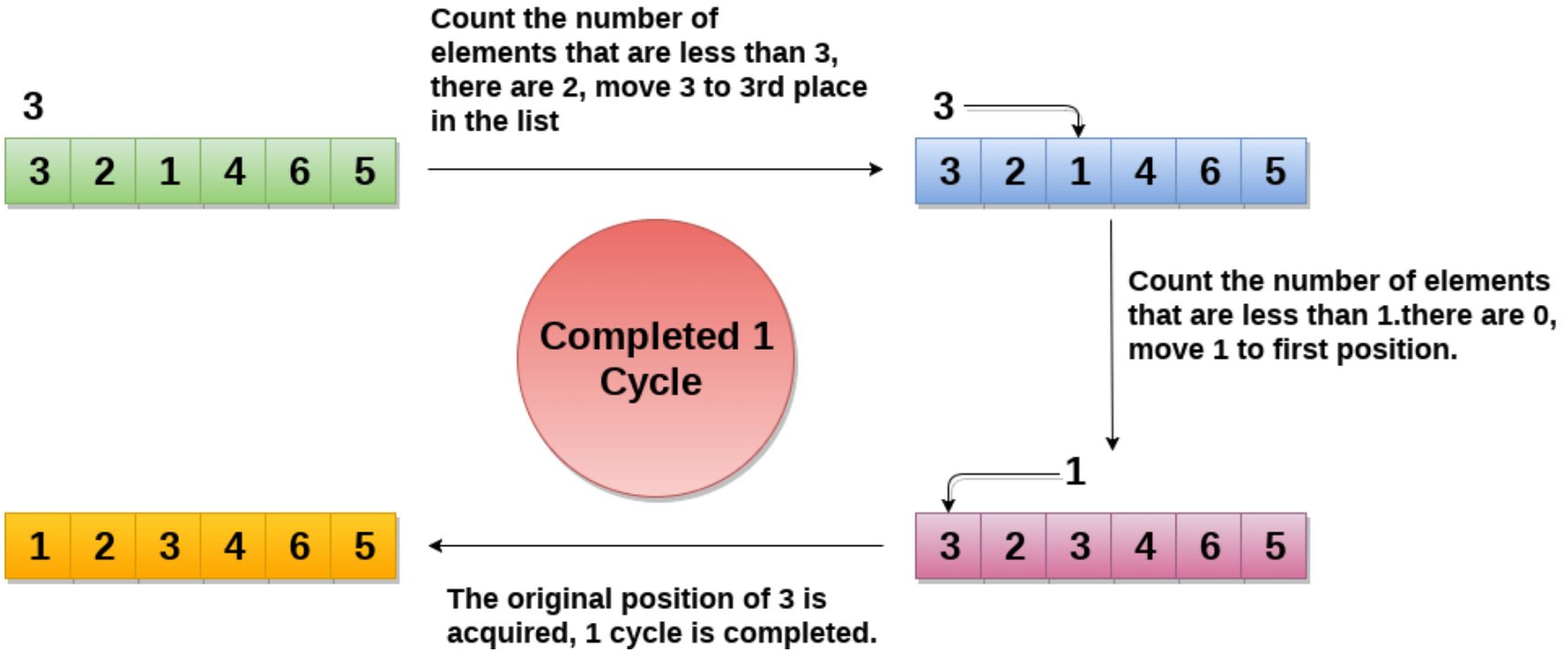
Cycle sort is a comparison sorting algorithm which forces array to be factored into the number of cycles where each of them can be rotated to produce a sorted array. It is theoretically optimal in the sense that it reduces the number of writes to the original array.

Algorithm

Consider an array of n distinct elements. An element a is given, index of a can be calculated by counting the number of elements that are smaller than a.

1. if the element is found to be at its correct position, simply leave it as it is.
2. Otherwise, find the correct position of **a** by counting the total number of elements that are less than **a**. where it must be present in the sorted array. The other element b which is replaced is to be moved to its correct position. This process continues until we got an element at the original position of **a**.

The illustrated process constitutes a cycle. Repeating this cycle for each element of the list. The resulting list will be sorted.



C program

```c
#include<stdio.h>
void sort(int a[], int n)
{
    int writes = 0,start,element,pos,temp,i;

    for (start = 0; start <= n - 2; start++) {
        element = a[start];
        pos = start;
        for (i = start + 1; i < n; i++)
            if (a[i] < element)
                pos++;
        if (pos == start)
            continue;
        while (element == a[pos])
            pos += 1;
        if (pos != start) {
            //swap(element, a[pos]);
        temp = element;
        element = a[pos];
            a[pos] = temp;
            writes++;
        }
        while (pos != start) {
            pos = start;
            for (i = start + 1; i < n; i++)
                if (a[i] < element)
                    pos += 1;
            while (element == a[pos])
                pos += 1;
            if (element != a[pos]) {
              temp = element;
          element = a[pos];
              a[pos] = temp;
                writes++;
            }
        }
    }

}
```

```
40.
41. int main()
42. {
43.     int a[] = {1, 9, 2, 4, 2, 10, 45, 3, 2};
44.     int n = sizeof(a) / sizeof(a[0]),i;
45.     sort(a, n);
46.     printf("After sort, array : ");
47.     for (i = 0; i < n; i++)
48.         printf("%d  ",a[i]);
49.     return 0;
50. }
```

Output:

```
After sort, array :
1
2
2
2
3
4
9
10
45
```

Java Program

```
1.  public class CycleSort
2.  {
3.  static void sort(int a[], int n)
4.  {
5.      int writes = 0,start,element,pos,temp,i;
6.
7.      for (start = 0; start <= n - 2; start++) {
8.          element = a[start];
9.          pos = start;
10.         for (i = start + 1; i < n; i++)
11.             if (a[i] < element)
12.                 pos++;
13.         if (pos == start)
14.             continue;
15.         while (element == a[pos])
16.             pos += 1;
17.       if (pos != start) {
18.           //swap(element, a[pos]);
19.       temp = element;
20.       element = a[pos];
21.           a[pos] = temp;
22.           writes++;
23.       }
24.       while (pos != start) {
25.           pos = start;
26.           for (i = start + 1; i < n; i++)
27.               if (a[i] < element)
28.                   pos += 1;
29.           while (element == a[pos])
30.               pos += 1;
31.           if (element != a[pos]) {
32.             temp = element;
```

```
        element = a[pos];
            a[pos] = temp;
              writes++;
          }
      }
  }
}
public static void main(String[] args)
{
  int a[] = { 1, 9, 2, 4, 2, 10, 45, 3, 2 };
  int n = a.length,i;
  sort(a, n);
  System.out.println("After sort, array : ");
  for (i = 0; i < n; i++)
      System.out.println(a[i]);

}
}
```

Output:

```
After sort, array :
1
2
2
2
3
4
9
10
45
```

C# Program

```
using System;
public class CycleSort
{
static void sort(int[] a, int n)
{
    int writes = 0,start,element,pos,temp,i;

    for (start = 0; start <= n - 2; start++) {
        element = a[start];
        pos = start;
        for (i = start + 1; i < n; i++)
            if (a[i] < element)
                pos++;
        if (pos == start)
            continue;
        while (element == a[pos])
            pos += 1;
        if (pos != start) {
            //swap(element, a[pos]);
        temp = element;
        element = a[pos];
            a[pos] = temp;
            writes++;
        }
        while (pos != start) {
            pos = start;
            for (i = start + 1; i < n; i++)
                if (a[i] < element)
```

```
                pos += 1;
          while (element == a[pos])
             pos += 1;
          if (element != a[pos]) {
            temp = element;
        element = a[pos];
            a[pos] = temp;
              writes++;
          }
      }
  }
}
public void Main()
{
   int[] a = { 1, 9, 2, 4, 2, 10, 45, 3, 2  };
   int n = a.Length,i;
   sort(a, n);
   Console.WriteLine("After sort, array : ");
   for (i = 0; i < n; i++)
       Console.WriteLine(a[i]);

}
}
```

Output:

```
After sort, array :
1
2
2
2
3
4
9
10
45
```

Tim-sort

Tim-sort is a sorting algorithm derived from insertion sort and merge sort. It was designed to perform in an optimal way on different kind of real world data. It is an adaptive sorting algorithm which needs O(n log n) comparisons to sort an array of n elements. It was designed and implemented by Tim Peters in 2002 in python programming language. It has been python's standard sorting algorithm since version 2.3.

Technique

Consider an array of n elements which needs to be sorted. In Tim sort, the array is divided into several parts where each of them is called a Run. These Runs are sorted by using insertion sort one by one and then the sorted runs get combined using a combine function. The idea of Tim sort is originated from the fact that, insertion sort works more optimally on the short lists rather than working on the larger lists.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	O(n)	O(n log n)	O(n log n)
Space Complexity			n

Steps :

1. Divide the array into the number of blocks known as run.
2. Consider size of run either 32 or 64(in the below implementation, size of run is 32.)
3. Sort the individual elements of every run one by one using insertion sort.
4. Merge the sorted runs one by one using merge function of merge sort.
5. Double the size of merged sub-arrays after every iteration.

C program

```
#include<stdio.h>
const int run = 32;
int minimum(int a, int b)
{
    if(a<b)
    return a;
    else
    return b;
}
void insertionSort(int a[], int beg, int end)
{
   int temp, i, j;
   for (i = beg + 1; i <= end; i++)
   {
       temp = a[i];
      j = i - 1;
       while (a[j] > temp && j >= beg)
      {
          a[j+1] = a[j];
         j--;
      }
      a[j+1] = temp;
   }
}

void merge(int a[], int left, int mid, int right)
{
   int len1 = mid - left + 1, len2 = right - mid;
   int beg[len1], end[len2];
   int i,j,k;
   for (i = 0; i < len1; i++)
      beg[i] = a[left + i];
   for (i = 0; i < len2; i++)
      end[i] = a[mid + 1 + i];

i = 0;
j = 0;
k = left;

while (i < len1 && j < len2)
{
   if (beg[i] <= end[j])
   {
      a[k] = beg[i];
       i++;
   }
   else
   {
       a[k] = end[j];
```

```
            j++;
        }
        k++;
    }
    while (i < len1)
    {
        a[k] = beg[i];
        k++;
        i++;
    }

    while (j < len2)
    {
        a[k] = end[j];
        k++;
        j++;
    }
}
void timSort(int a[], int n)
{
    int i,size,beg,mid,end;
    for (i = 0; i < n; i+=run)
        insertionSort(a, i, minimum((i+31), (n-1)));
    for (size = run; size < n; size = 2*size)
    {
        for (beg = 0; beg < n; beg += 2*size)
        {
            mid = beg + size - 1;
            end = minimum((beg + 2*size - 1), (n-1));

            merge(a, beg, mid, end);
        }
    }
}

int main()
{
    int a[] = {12,1,20,2,3,123,54,332},i;
    int n = sizeof(a)/sizeof(a[0]);
    printf("Printing Array elements \n");
    for (i = 0; i < n; i++)
    printf("%d  ", a[i]);
    printf("\n");
    timSort(a, n);

    printf("Printing sorted array elements \n");
    for (i = 0; i < n; i++)
        printf("%d  ", a[i]);
    printf("\n");
    return 0;
        }
```

Output:

```
Printing Array elements
12  1  20  2  3  123  54  332

Printing sorted array elements
1  2  3  12  20  54  123  332
```