UNIT-I

Introduction to Linear Data Structures: Definition and importance of linear data structures, Abstract data types (ADTs) and their implementation, Overview of time and space complexity analysis for linear data structures. Searching Techniques: Linear & Binary Search, Sorting Techniques: Bubble sort, Selection sort, Insertion Sort.

What is data structure?

A **data structure** is a **data** organization, management and storage format that enable efficient access and modification. a **data structure** is a collection of **data** values, the relationships among them, and the functions or operations that can be applied to the **data.**

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing 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, Artificial 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 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 value.

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 complex and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount 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

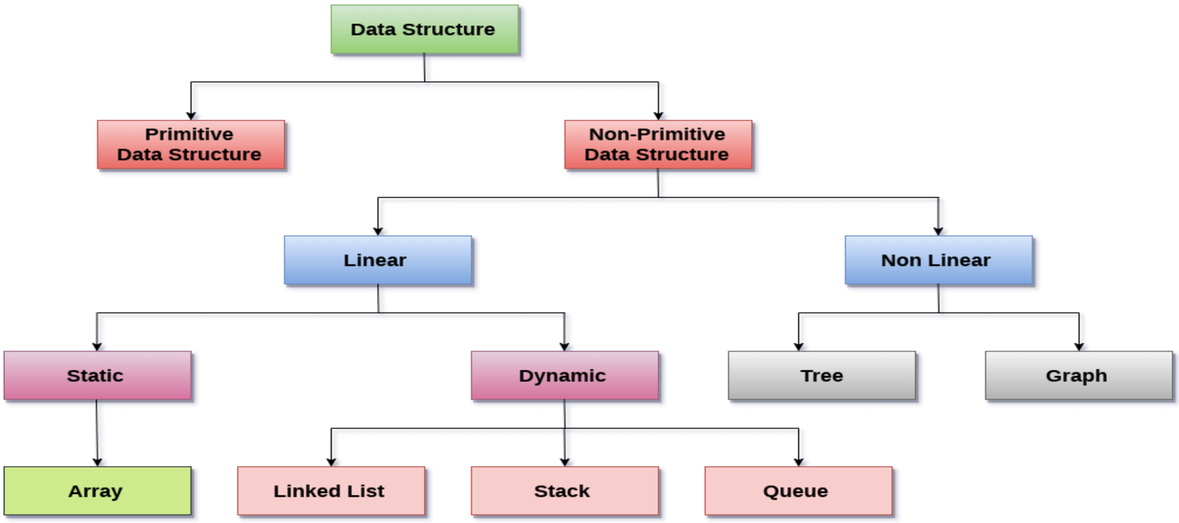
Efficiency: Efficiency of a program depends upon the choice of data structures.

For example: suppose, we have some data and we need to perform the search for a particular 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.

- It's a data structure that stores and manages data in a linear order.
- The data elements of the sequence are linked from one to the next.
- Implementing the linear structure of data within a computer's RAM is simple, provided that the data are organized sequentially.
- Array, queue. Stack, linked list, etc., are a few examples of such a structure.
- Only one relationship exists between the data elements in the data structure.
- Because the data elements are stored on a single level, it is possible to traverse the data elements in one run.
- If a linear data storage structure is used, it is not well utilized.
- The complexity of the structure's time increases with an increase in its size.

Characteristics of Linear Data Structure:

- **Sequential Organization:** In linear data structures, data elements are arranged sequentially, one after the other. Each element has a unique predecessor (except for the first element) and a unique successor (except for the last element)
- **Order Preservation:** The order in which elements are added to the data structure is preserved. This means that the first element added will be the first one to be accessed or removed, and the last element added will be the last one to be accessed or removed.
- **Fixed or Dynamic Size:** Linear data structures can have either fixed or dynamic sizes. Arrays typically have a fixed size when they are created, while other structures like linked lists, stacks, and queues can dynamically grow or shrink as elements are added or removed.
- **Efficient Access:** Accessing elements within a linear data structure is typically efficient. For example, arrays offer constant-time access to elements using their index.

Types of Linear Data Structures:

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.

Advantages of Linear Data Structures

- **Efficient data access:** Elements can be easily accessed by their position in the sequence.
- **Dynamic sizing:** Linear data structures can dynamically adjust their size as elements are added or removed.
- **Ease of implementation:** Linear data structures can be easily implemented using arrays or linked lists.
- **Versatility:** Linear data structures can be used in various applications, such as searching, sorting, and manipulation of data.
- **Simple algorithms:** Many algorithms used in linear data structures are simple and straightforward.

Disadvantages of Linear Data Structures

- **Limited data access:** Accessing elements not stored at the end or the beginning of the sequence can be time-consuming.
- **Memory overhead:** Maintaining the links between elements in linked lists and pointers in stacks and queues can consume additional memory.
- **Complex algorithms:** Some algorithms used in linear data structures, such as searching and sorting, can be complex and time-consuming.
- **Inefficient use of memory:** Linear data structures can result in inefficient use of memory if there are gaps in the memory allocation.
- **Unsuitable for certain operations:** Linear data structures may not be suitable for operations that require constant random access to elements, such as searching for an element in a large dataset.

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:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy 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 child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories.

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 cannot 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 subjects, we need to traverse the complete array of marks and calculate the total sum, then we will divide 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.

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

Characteristics of a Data Structure

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

Abstract data types (ADTs) and their implementation

What is abstract data type?

An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy

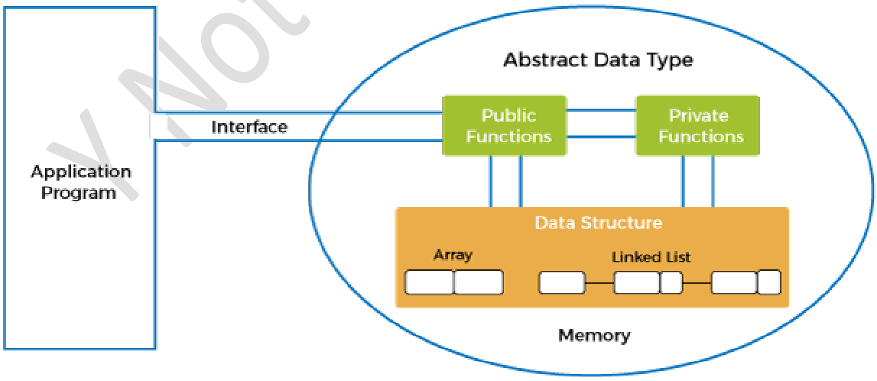
For example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.

For example, a List is an abstract data type that is implemented using a dynamic array and linked list. A queue is implemented using linked list-based queue, array-based queue, and stack-based queue. A Map is implemented using Tree map, hash map, or hash table.

Abstract data type model

Before knowing about the abstract data type model, we should know about abstraction and encapsulation.

- **Abstraction:** It is a technique of hiding the internal details from the user and only showing the necessary details to the user.
- **Encapsulation:** It is a technique of combining the data and the member function in a single unit is known as encapsulation.



- The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function.
- The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT.
- Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program.

Let's understand the abstract data type with a real-world example.

If we consider the smartphone. We look at the high specifications of the smartphone, such as:

- 4 GB RAM
- Snapdragon 2.2ghz processor
- 5-inch LCD screen
- Dual camera
- Android 13.0

The above specifications of the smartphone are the data, and we can also perform the following operations on the smartphone:

- call (): We can call through the smartphone.
- text (): We can text a message.
- photo (): We can click a photo.
- video (): We can also make a video.

The smartphone is an entity whose data or specifications and operations are given above. The abstract/logical view and operations are the abstract or logical views of a smartphone.

The implementation view of the above abstract/logical view is given below:

```
class Smartphone
{
  private:
          int ramSize;
          string processorName;
          float screenSize;
          int cameraCount;
          string androidVersion;
  public:
           void call();
           void text();
           void photo();
           void video();
}
```

The above code is the implementation of the specifications and operations that can be performed on the smartphone. The implementation view can differ because the syntax of programming languages is different, but the abstract/logical view of the data structure would remain the same. Therefore, we can say that the abstract/logical view is independent of the implementation view.

Features of ADT:

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- Robust: The program is robust and has the ability to catch errors.
- **Encapsulation:** ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction:** ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence:** ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity:** ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner. Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development.

Advantages:

- **Encapsulation:** ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- **Abstraction:** ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- **Data Structure Independence:** ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- **Information Hiding:** ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
- **Modularity:** ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

- **Overhead:** Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- **Complexity:** ADTs can be complex to implement, especially for large and complex data structures.
- **Learning Curve:** Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.

- **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- **Cost:** Implementing ADTs may require additional resources and investment, which can increase the cost of development.

Abstract Data Types Overview

Abstract Data Type	Other Common Names	Commonly Implemented with
List	Sequence	Array, Linked List
Queue		Array, Linked List
Double-ended Queue	Dequeue, Deque	Array, Doubly-linked List
Stack		Array, Linked List
Associative Array	Dictionary, Hash Map, Hash, Map**	Hash Table
Set		Red-black Tree, Hash Table
Priority Queue	Heap	Heap

Overview of time and space complexity analysis for linear data structures.

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements.

Def 1: -Performance of an algorithm is a process of making evaluative judgement about algorithms.

Def 2: -Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

- Whether that algorithm is providing the exact solution for the problem?
- Whether it is easy to understand?
- Whether it is easy to implement?
- How much space (memory) it requires to solve the problem?
- How much time it takes to solve the problem? Etc.,

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.

Based on this information, performance analysis of an algorithm can also be defined

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

- Space required to complete the task of that algorithm **(Space Complexity)**. It includes program space and data space
- Time required to complete the task of that algorithm **(Time Complexity)**

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

- To store program instructions.
- To store constant values.
- To store variable values.
- And for few other things like function calls, jumping statements etc,.

Space complexity of an algorithm can be defined as

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons.

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

Note - When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack.

That means we calculate only the memory required to store Variables, Constants, Structures, etc., To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

Consider the following piece of code...

```
int square (int a)
{
        return a*a;
}
```

In the above piece of code, it requires 2 bytes of memory to store variable **'a'** and another 2 bytes of memory is used for **return value**.

That means, totally it requires 4 bytes of memory to complete its execution. And these 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be Constant Space Complexity.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Consider the following piece of code...

```
int sum(int A[ ], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++)
    sum = sum + A[i];
  return sum;
}
```

In the above piece of code it requires

'n*2' bytes of memory to store array variable **'a[ ]'**

2 bytes of memory for integer parameter **'n'**

4 bytes of memory for local integer variables **'sum'** and **'i'** (2 bytes each)

2 bytes of memory for **return value**.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be *Linear Space Complexity*.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

Space Complexity:

The space needed by algorithm is the sum of following components

- A fixed part→that is independent of the characteristics of input and output. Example Number & size. In this includes instructions space [space for code] + Space for simple variables + Fixed-size component variables + Space for constant & soon.

- A variable part→ that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved + The space needed by referenced variables + Recursion stack space.

The space requirement S(P) of any algorithm P can be written as

$$\boxed{S(P)=C+ S_P}$$

C→Constant
S_P→ Instance characteristics.

Find space complexity of iterative Algorithm of sum of 'n' numbers.	**Find space complexity for Recursive algorithm:**

Algorithm:
Algorithm sum(a,n)
//Here a is array of Size n
{
S:=0;
if(n≤0)) then return 0;
for i:=1 to n do
s:=s+a[i];
return s;
}
	Algorithm:
	Algorithm RSUM(a,n)
	//a is an array of size n
	{
	If(n≤0) then return 0;
	Else return a[n]+RSUM(a,n-1);
	}

Space complexity S(P): s→1 word i→1 word n→1 word a→n word $\boxed{S(P)\geq n+3}$ Word is a String of bits stored in computer memory, its size is a 4 to 64 bits.	**Space Complexity:** The recursion stack space includes space for the formal parameters, the local variables and the return address. Return address requires→1 word memory Each call to RSUM requires→ at least 3words (It includes space for the values of n, the return address and a pointer to a[]). The depth recursion is n+1 The recursion stack space needed is ≥3(n+1)

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

The time complexity of an algorithm can be defined as.

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine.
3. **Read** and **Write** speed of the machine.
4. The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.,
5. **Input** data

Note - When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system. To solve this problem, we must assume a model machine with a specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine
2. It is a 32-bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above-defined model machine...

Consider the following piece of code...

```
int sum(int a, int b)
{
  return a+b;
}
```

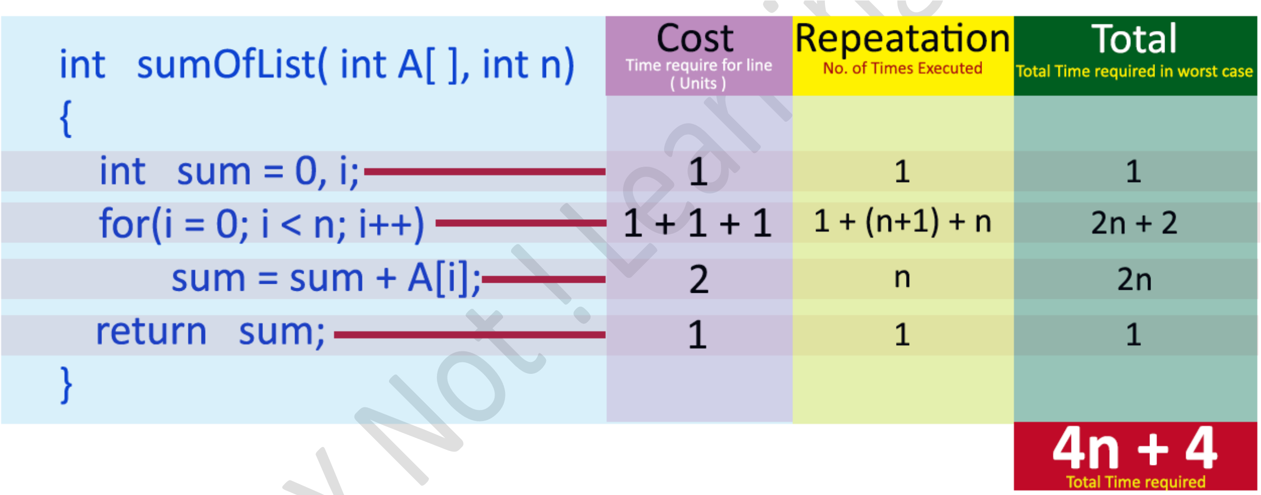
In the above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires the same amount of time i.e. 2 units.

If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

```
int sum (int A [], int n)
{
  int sum = 0, i;
  for (i = 0; i < n; i++)
    sum = sum + A[i];
  return sum;
}
```

For the above code, time complexity can be calculated as



	Cost Time require for line (Units)	Repeatation No. of Times Executed	Total Total Time required in worst case
int sumOfList(int A[ ], int n)			
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
			4n + 4 Total Time required

- ➢ In above calculation
- ➢ **Cost** is the amount of computer time required for a single operation in each line.
- ➢ **Repetition** is the amount of computer time required by each operation for all its repeatations.
- ➢ **Total** is the amount of computer time required by each operation to execute.
- ➢ So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is *Linear Time Complexity*.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity.

Here RUN means Compile + Execution.

Time Complexity $\boxed{T(P)=t_c+t_p}$

But we are neglecting t_c Because the compile time does not depends on the instance characteristics. The compiled program will be run several times without recompilation.

So $T(P)= t_p$

Here t_p → instance characteristics.

For example:

The program **p** do some operations like ADD, SUB, MUL etc.

If we knew the characteristics of the compiler to be used, we could process to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores and so on.

We obtain $t_p(n)$ express as follow:

$t_p(n)=C_aADD(n)+ C_SSUB(n)+ C_mMUL(n)+ C_dDIV(n)+$………..

n→ Instance characteristics

C_a, C_S, C_m, C_d, …… → time needed for an addition, Subtraction, multiplication, division, and etc.

ADD, SUBB, MUL, DIV → Are functions, whose values are the numbers of additions, subtractions, multiplications,.. etc, that are performed when the code for p is used on an instance with characteristics.

Performance Measurement

Program step: Program step is the syntactically or semantically meaningful segment of a program. And it has an execution time that is independent of the instance characteristics.

Example:

- For comment→//-- zero steps
- For assignment statements (Which does not involve any calls to other algorithms)

  := →one step
- For iterative statements such as "for, while and until-repeat" statements, we consider the step counts only for control part of the statement.

  For while loop "while (<expr>) do " : the step count for control part of a while stmt is → Number of step counts for assignable to <expr>

  For for loop ie for i:=<expr> to <expr1> do: The step count of control part of "for" statement is→ Sum of the count of <expr> & <expr1> N and remaining execution of the "for" statement, i.e., one.

We determine the number of steps needed by a program to solve a particular problem. For this there are two methods.

1) **First method** is, introduce a new variable **"count"** in to the program for finding number of steps in program.

2) **Second method** is, building a "**table**" in which we list the total number of steps contributed by each statement.

Example for 1st method:

Find time complexity of **Iterative** algorithm of sum of ‘n’ numbers**.**	Find time complexity for **Recursive** algorithm:

Algorithm:
Algorithm sum(a,n)
{
// count is global it is initially zero
S:=0;
Count:=count+1; // count for assignment
for i:=1 to n do
{
Count :=count+1; //for “for” loop
s:=s+a[i];
count :=count+1; //for assingment
}
Count :=count+1; //for last time of for loop
Count :=count+1; // for return stmt
return s;
}
	Algorithm:
	Algorithm RSUM(a,n)
	{
	Count :=count+1; // for if condition
	If(n≤0) then
	{
	Count:=count+1; // for return stmt
	return 0;
	}
	Else {
	Count :=count+1;
	//for the addition, function invocation & return
	return a[n]+RSUM(a,n-1);
	}
	}

Finally count values is =2n+3; So total number of steps= **2n+3**	If n=0 then $t_{Rsum}(0)=2$ If n>0 then increases by 2, ie., 2+ $t_{Rsum}(n-1)$ Means $t_{Rsum}(n)=2+t_{Rsum}(n-1)$ $=2+2+t_{Rsum}(n-2)$ $=2+2+2+t_{Rsum}(n-3)$ . . $=2(n)+t_{Rsum}(n-n)$ $=2n+t_{Rsum}(0)=$**2n+2**

In the above example the recursive algorithm has **less** time complexity then iterative algorithm.

Example for 2nd method:

Find time complexity of Algorithm of sum of ‘n’ numbers.			
Statements	s/e	Frequency	Total steps

1. Algorithm sum(a,n)	0	_	0
2. {	0	_	0
3. S:=0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. s:=s+a[i];	1	n	n
6. return s;	1	1	1
7. }	0	_	0
			2n+3

Find time complexity for Recursive algorithm					
Statements	s/e	Frequency		Total steps	
		n=0	n>0	n=0	n>0
1. Algorithm RSUM(a,n)	0	_	_	_	_
2. {	0	_	_	_	_
3. If(n≤0) then	1	1	1	1	1
4. return 0;	1	1	0	1	0
5. Else	0				
6. return a[n]+RSUM(a,n-1);	1+x	0	1	0	1+x
7. }	0	_	_	0	0
				2	2+x

Here x= t_{Rsum}(n-1)

Asymptotic Notation:

A problem may have numerous (many) algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run.

Asymptotic notation of an algorithm is a mathematical representation of its complexity

Asymptotic notation is used to judge the best algorithm among numerous algorithms for a particular problem.

Asymptotic complexity is a way of expressing the main component of algorithms like

- Cost
- Time complexity
- Space complexity

Some Asymptotic notations are

1. Big oh→O
2. Omega→Ω
3. Theta →θ
4. Little oh→o
5. Little Omega→ω

1. Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

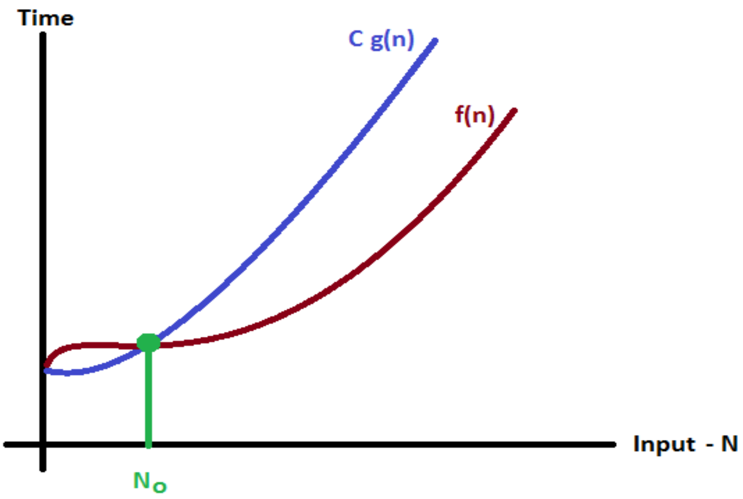
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be **defined** as follows...

The function f(n) =O(g(n)) (read as "f of n is big oh of g of n) iff (if and only if) there exit positive constants c and n_0 such that f(n)<=c*g(n) for all n, n>=n_0

f(n)=O(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C x g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) <= C g(n)
⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)

2. Big - Omega Notation (Ω)

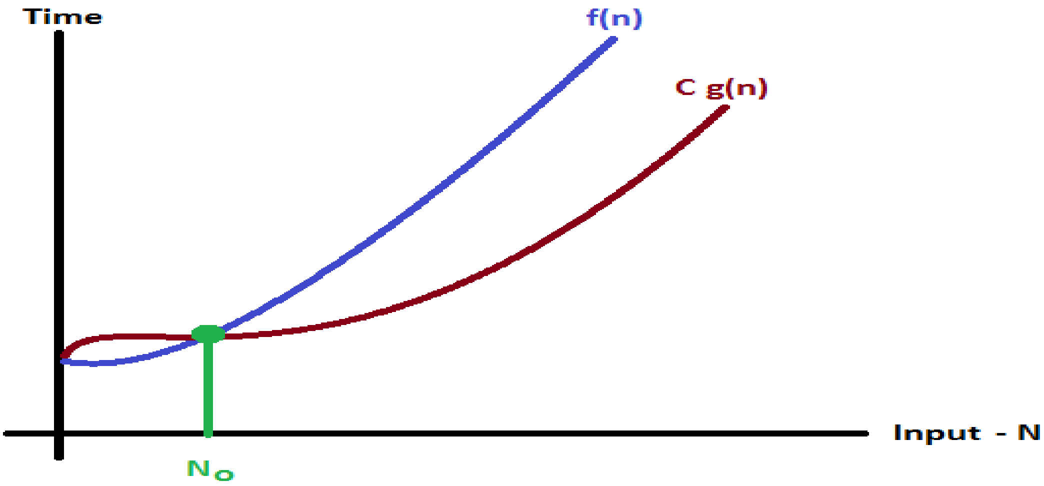
Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

The function f(n) = Ω(g(n)) (read as "f of n is omega of g of n) iff (if and only if) there exit positive constants c and n_0 such that f(n)>=c*g(n) for all n, n>=n_0 f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)
⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
By using Big - Omega notation we can represent the time complexity as follows...
3n + 2 = Ω(n)

3. Big - Theta Notation (Θ)

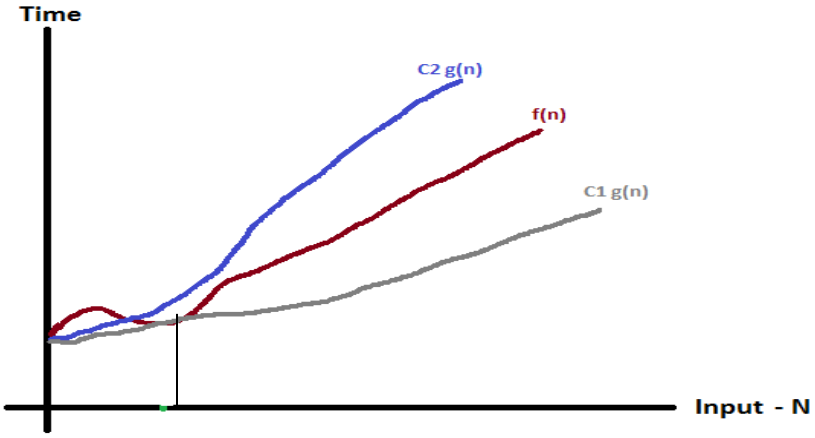
Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

The function f(n) = Θ (g(n)) (read as "f of n is theta of g of n) iff (if and only if) there exist positive constants c_1,c_2 and n_0 such thatc_1*g(n)<= f(n)<=c_2*g(n) for all n, n>=n_0
f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always **C_1 g(n)** is less than **f(n)** and **C_2 g(n)** is greater than **f(n)** which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) >= C_2 g(n)** for all values of **C_1, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) >= ⇒C_2 g(n)
C_1 n <= 3n + 2 >= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 1**.
By using Big - Theta notation we can represent the time complexity as follows...
3n + 2 = Θ (n)

1. Little oh: o

The function f(n)=o(g(n)) (read as “f of n is little oh of g of n”) iff

Lim f(n)/g(n)=0 for all n, n≥0

n→~

Example:

3n+2= o(n^2) as

Lim ((3n+2)/n^2)=0

n→~

2. Little Omega:ω

The function f(n)= ω (g(n)) (read as “f of n is little ohomega of g of n”) iff

Lim g(n)/f(n)=0 for all n, n≥0

n→~

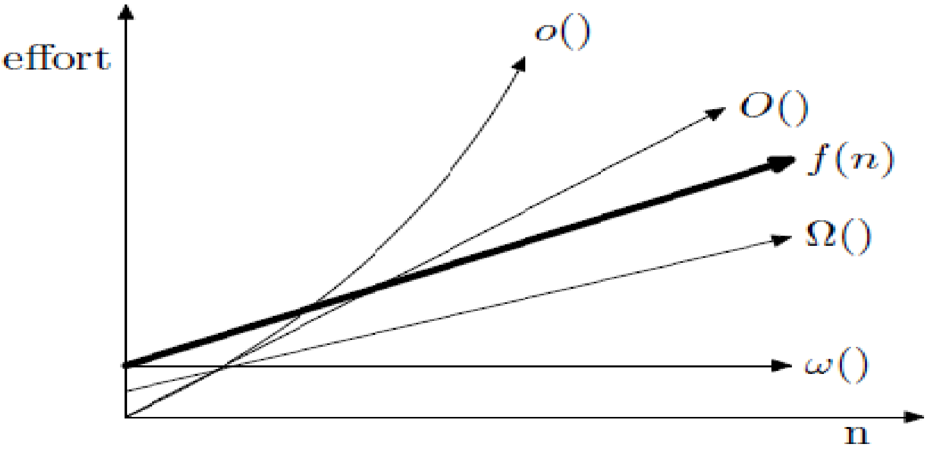
Example:

3n+2= o(n^2) as

Lim (n^2/(3n+2) =0

n→~

Graph for visualize the relationships between these notations



Performance Measurement

Searching Techniques:

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

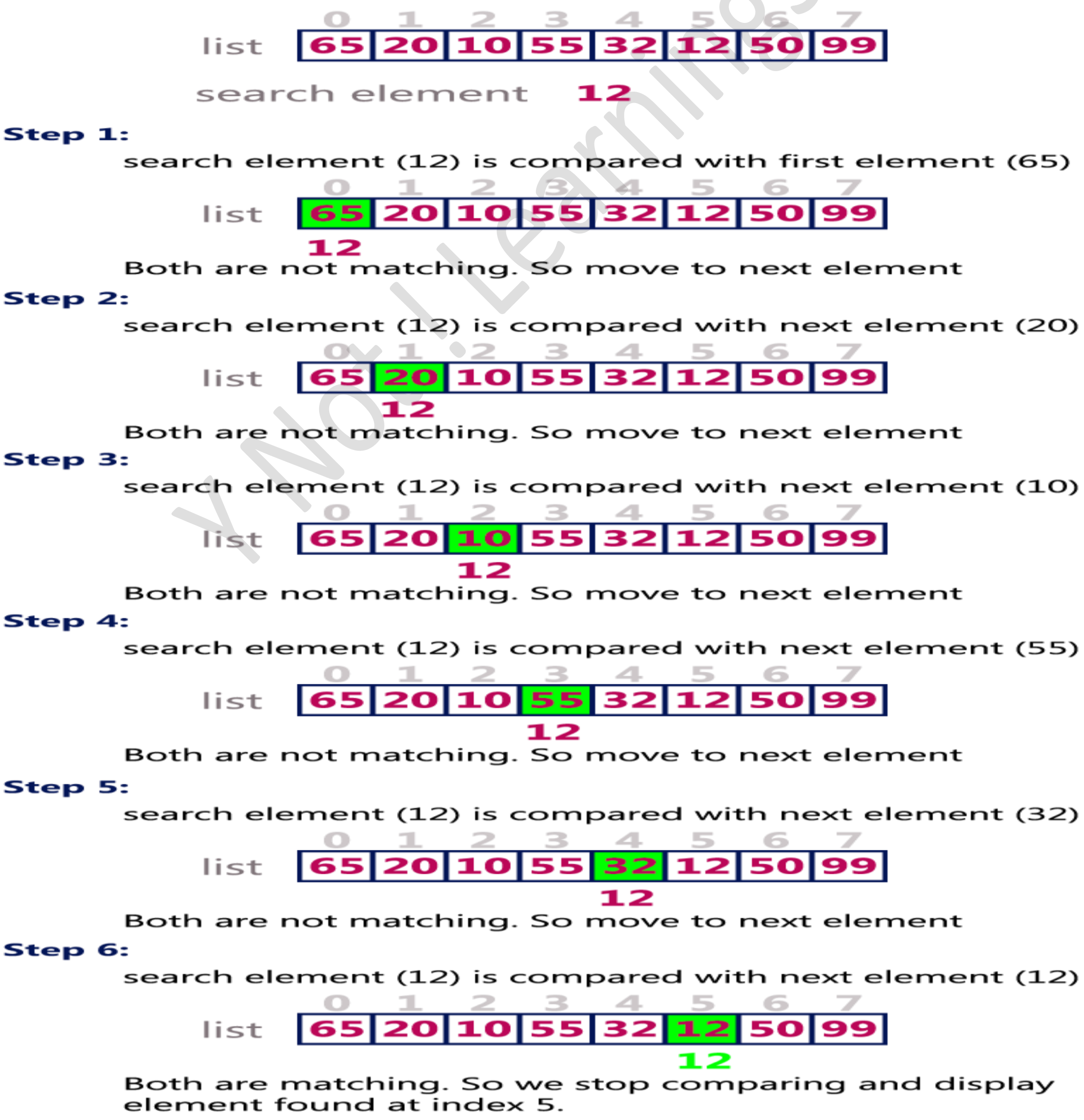
Linear search algorithm finds a given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing search element with the first element in the list. If both are matched then result is element found otherwise search element is compared with the next element in the list. Repeat the same until search element is compared with the last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.
- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Example

Consider the following list of elements and the element to be searched...



0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
search element 12

Step 1:
search element (12) is compared with first element (65)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 2:
search element (12) is compared with next element (20)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 3:
search element (12) is compared with next element (10)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 4:
search element (12) is compared with next element (55)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

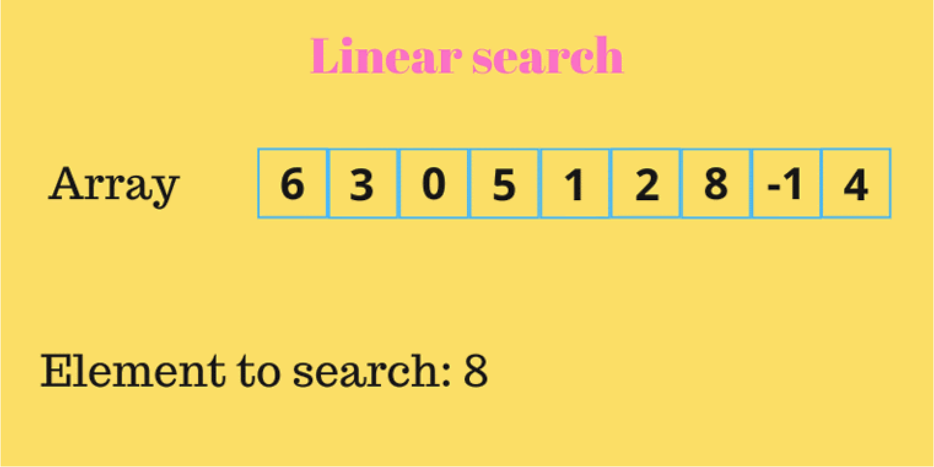
Step 5:
search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:
search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Implementation of Linear Search Algorithm using C



Example

```c
#include <stdio.h>
int main()
{
   int a[10], i, item,n;
   printf("\nEnter number of elements of an array:\n");
   scanf("%d",&n);
   printf("\nEnter elements: \n");
   for (i=0; i<n; i++)
      scanf("%d", &a[i]);
   printf("\nEnter item to search: ");
   scanf("%d", &item);
   for (i=0; i<=9; i++)
      if (item == a[i])
      {
         printf("\nItem found at location %d", i+1);
         break;
      }
   if (i > 9)
      printf("\nItem does not exist.");
   return 0;
}
```

Output

```
Enter number of elements of an
array:
9
Enter elements:
11
2
3
55
33
66
12
677
66

Enter item to search: 12
Item found at location 7
```

Binary Search Algorithm

Binary search algorithm finds a given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only a sorted list of elements. That means the binary search is used only with a list of elements that are already arranged in an order. The binary search cannot be used for a list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for the left sublist of the middle element. If the search element is larger, then we repeat the same process for the right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

Step 1 - Read the search element from the user.

Step 2 - Find the middle element in the sorted list.

Step 3 - Compare the search element with the middle element in the sorted list.

Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.

Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

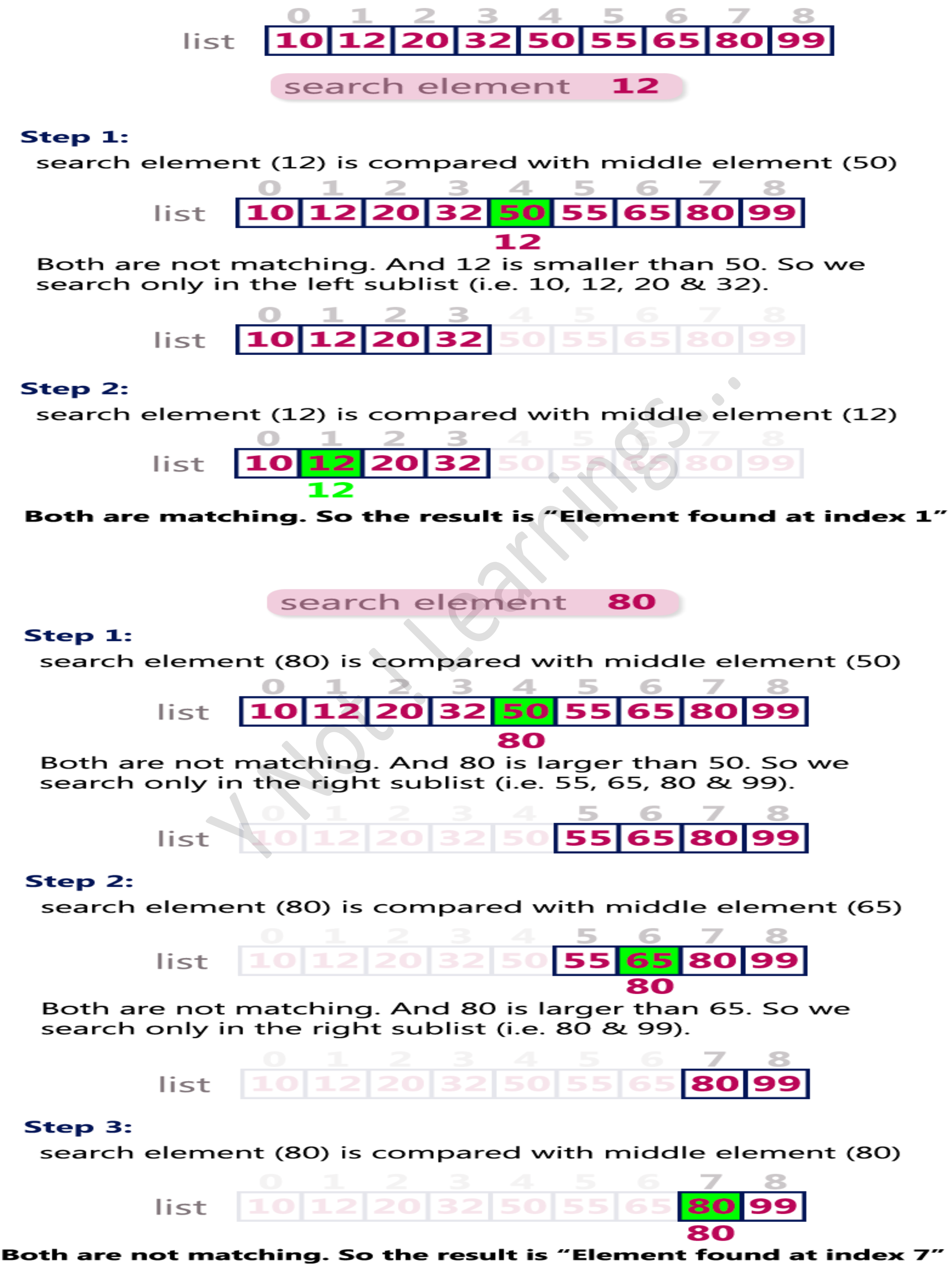
Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example

Consider the following list of elements and the element to be searched...



	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

search element 12

Step 1:

search element (12) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	🟩 50	55	65	80	99
					12				

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (12) is compared with middle element (12)

	0	1	2	3	4	5	6	7	8
list	10	🟩 12	20	32	50	55	65	80	99
		12							

Both are matching. So the result is "Element found at index 1"

search element 80

Step 1:

search element (80) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	🟩 50	55	65	80	99
					80				

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	🟩 65	80	99
							80		

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 3:

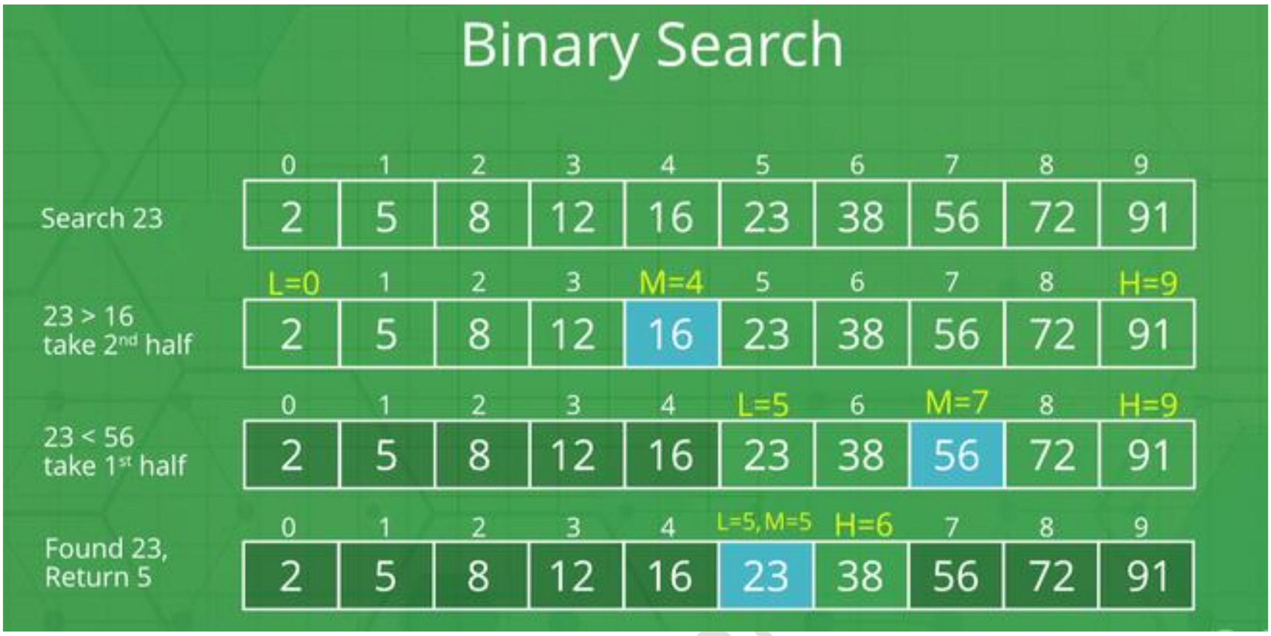
search element (80) is compared with middle element (80)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	🟩 80	99
								80	

Both are not matching. So the result is "Element found at index 7"

Binary Search Algorithm using C

Binary Search is defined as a searching algorithm used in a sorted array by **repeatedly dividing the search interval in half**. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).



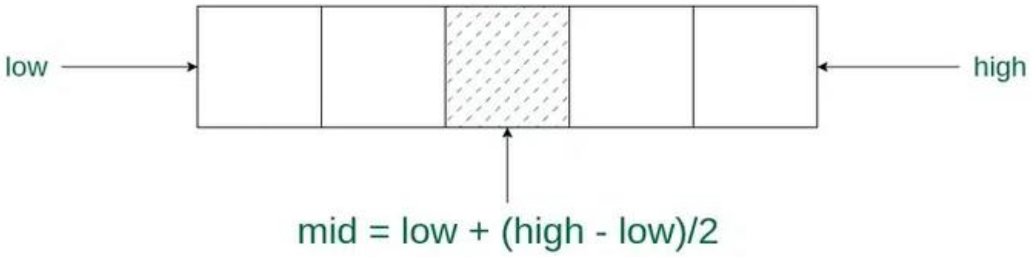
Conditions for when to apply Binary Search in a Data Structure:

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

Binary Search Algorithm:

Divide the search space into two halves by finding the middle index “mid”.



Compare the middle element of the search space with the key.

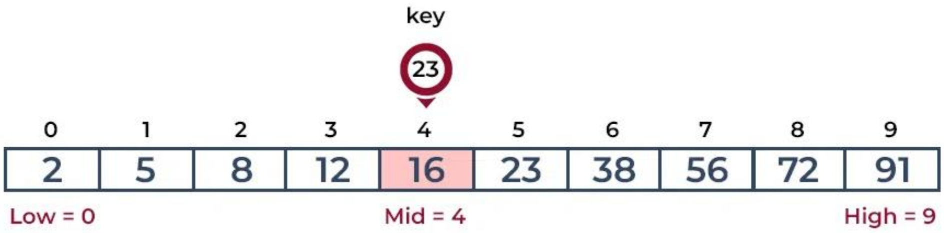
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

How does Binary Search work?

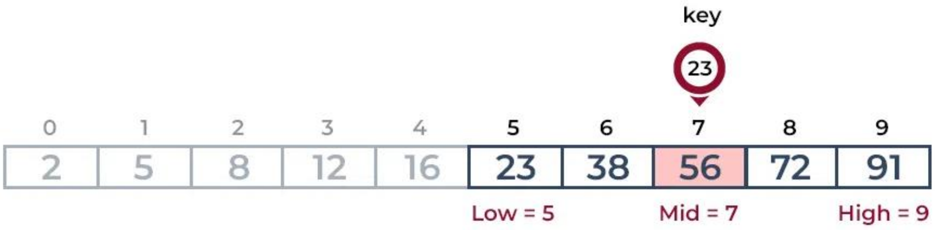
Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.

First Step: Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

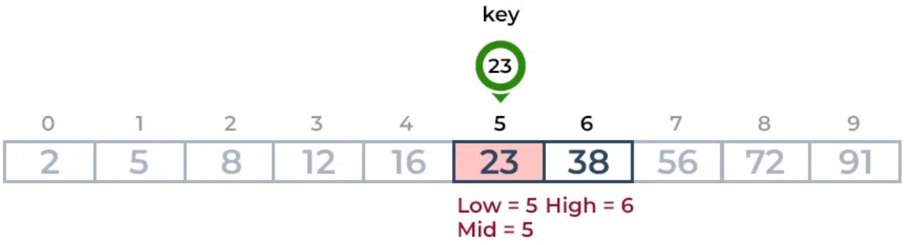
- Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.



- Key is less than the current mid 56. The search space moves to the left.



Second Step: If the key matches the value of the mid element, the element is found and stop search.



How to Implement Binary Search?

The Binary Search Algorithm can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

Given below are the pseudocodes for the approaches.

1. **Iterative Binary Search Algorithm:**

Here we use a while loop to continue the process of comparing the key and splitting the search space in two halves.

// C program to implement iterative Binary Search
#include <stdio.h>
// An iterative binary search function.
int binarySearch(int arr[], int l, int r, int x)
{
        while (l <= r)
	OutPut:- Element is present at index 3

```
{
        int m = l + (r - l) / 2;
        // Check if x is present at mid
        if (arr[m] == x)
            return m;
        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;
        // If x is smaller, ignore right half
        else
            r = m - 1;
    }
    // If we reach here, then element was not present
    return -1;
}
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array"):
    printf("Element is present at index %d",result);
    return 0;
}
```

Time Complexity: O (log N)

Auxiliary Space: O (1)

2. Recursive Binary Search Algorithm:

Create a recursive function and compare the mid of the search space with the key. And based on the result either return the index where the key is found or call the recursive function for the next search space.

```
// C program to implement recursive Binary Search
#include <stdio.h>
// A recursive binary search function. It returns location of x in given array
arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
{
        int mid = l + (r - l) / 2;
// If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;
// If element is smaller than mid, then it can only be present in left subarray
```

Output

```
Element is present at
index 3
```

```
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
// Else the element can only be present in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }
// We reach here when element is not present in array
    return -1;
}
// Driver code
int main()
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

Complexity Analysis of Binary Search:

Time Complexity:

Best Case: O(1)

Average Case: O(log N)

Worst Case: O(log N)

Auxiliary Space: O(1), If the recursive call stack is considered then the auxiliary space will be O(logN).

Advantages of Binary Search:

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

- The array should be sorted.

- ➢ Binary search requires that the data structure being searched be stored in contiguous memory locations.
- ➢ Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

Applications of Binary Search:

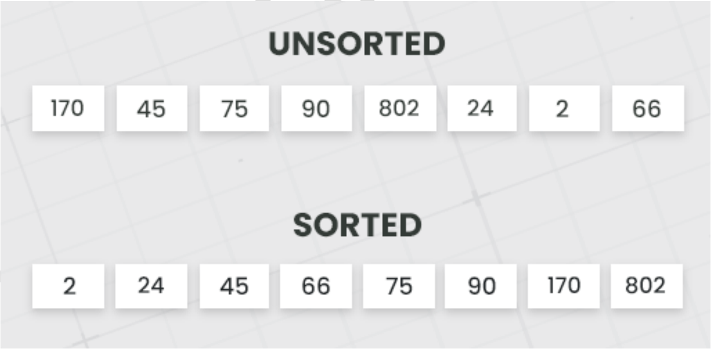
- ➢ Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- ➢ It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
- ➢ It can be used for searching a database.

Sorting Techniques:

What is Sorting?

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

For Example: The below list of characters is sorted in increasing order of their ASCII values. That is, the character with a lesser ASCII value will be placed first than the character with a higher ASCII value.



UNSORTED

170	45	75	90	802	24	2	66

SORTED

2	24	45	66	75	90	170	802

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Bubble Sort Algorithm

- ➢ Traverse from left and compare adjacent elements and the higher one is placed at right side.
- ➢ In this way, the largest element is moved to the rightmost end at first.
- ➢ This process is then continued to find the second largest and place it and so on until the data is sorted.

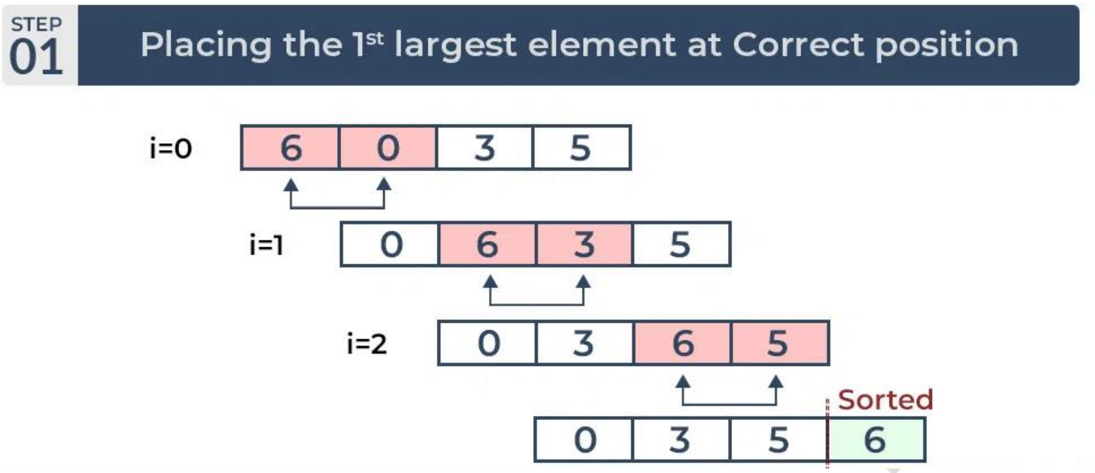
How does Bubble Sort Work?

Let us understand the working of bubble sort with the help of the following illustration:

Input: arr[] = {6, 3, 0, 5}

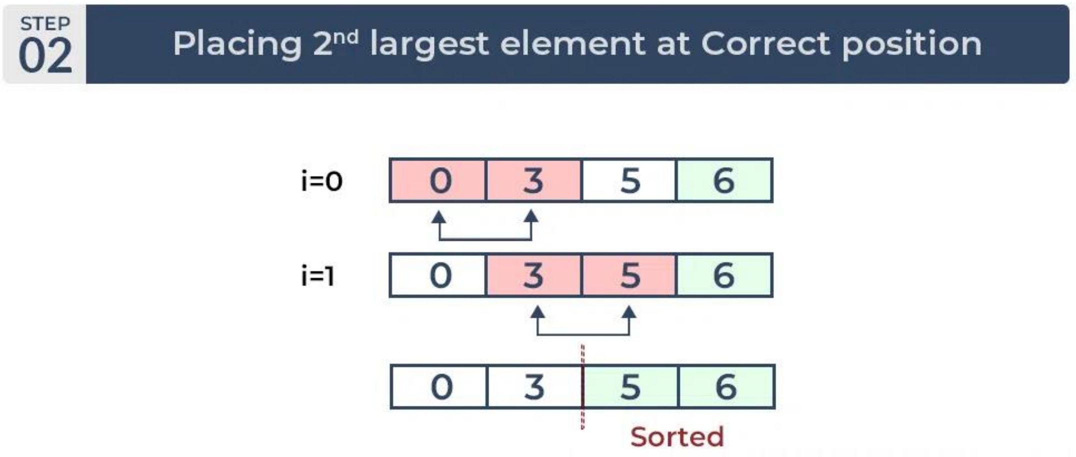
First Pass:

The largest element is placed in its correct position, i.e., the end of the array.



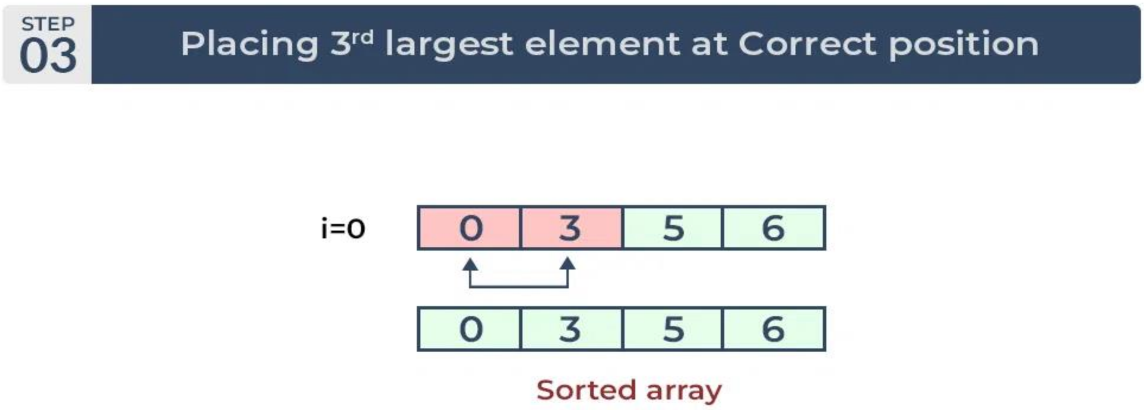
Second Pass:

Place the second largest element at correct position



Third Pass:

Place the remaining two elements at their correct positions.



Total no. of passes: n-1

Total no. of comparisons: n*(n-1)/2

Implementation of Bubble Sort

```
// Optimized implementation of Bubble sort
#include <stdbool.h>
#include <stdio.h>
void swap(int* xp, int* yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}
// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
        int i, j;
        bool swapped;
        for (i = 0; i < n - 1; i++)
        {
                swapped = false;
                for (j = 0; j < n - i - 1; j++)
                {
                        if (arr[j] > arr[j + 1])
                        {
                                swap(&arr[j], &arr[j + 1]);
                                swapped = true;
                        }
                }
// If no two elements were swapped by inner loop, then break
                if (swapped == false)
                        break;
        }
}
// Function to print an array
void printArray(int arr[], int size)
{
        int i;
        for (i = 0; i < size; i++)
                printf("%d ", arr[i]);
}
// Driver program to test above functions
int main()
{
        int arr[] = { 64, 34, 25, 12, 22, 11, 90,2,8,99 };
        int n = sizeof(arr) / sizeof(arr[0]);
        bubbleSort(arr, n);
        printf("Sorted array: \n");
```

Output

```
Sorted array:
2 8 11 12 22 25 34 64 90 99
```

printArray(arr, n);
return 0;
}	

Complexity Analysis of Bubble Sort:

Time Complexity: O(N2)

Auxiliary Space: O (1)

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

Selection Sort Algorithm

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order. Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

Step by Step Process

Step 1 - Select the first element of the list (i.e., Element at first position in the list).

Step 2: Compare the selected element with all the other elements in the list.

Step 3: In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.

Step 4: Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Selection Sort Logic

```
//Selection sort logic
  for(i=0; i<size; i++)
{
      for(j=i+1; j<size; j++)
{
```

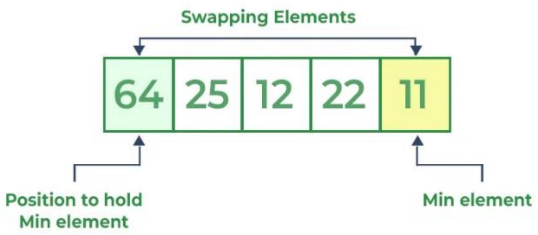
```
                if(list[i] > list[j])
                  {
                  temp=list[i];
                  list[i]=list[j];
                  list[j]=temp;
                }
            }
        }
```

How does Selection Sort Algorithm work?

Let’s consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

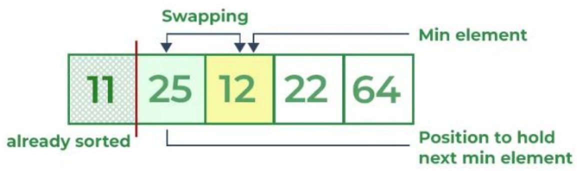
First pass:

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.
- Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.



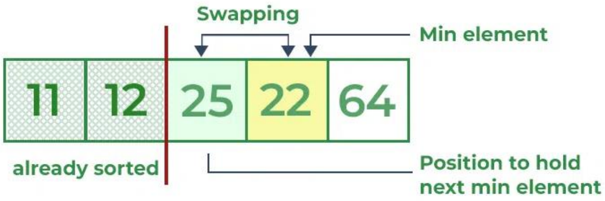
Second Pass:

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
- After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



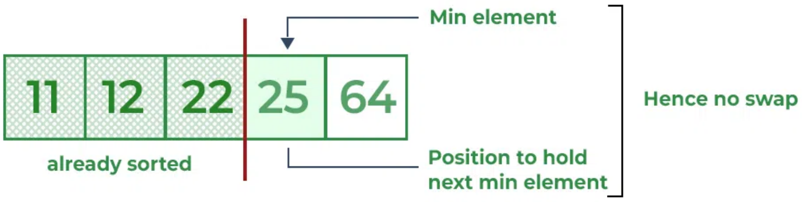
Third Pass:

- Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.
- While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.



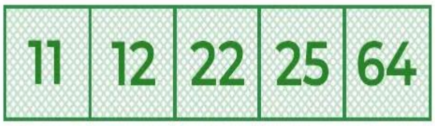
Fourth pass:

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As 25 is the 4th lowest value hence, it will place at the fourth position.



Fifth Pass:

- At last the largest value present in the array automatically get placed at the last position in the array
- The resulted array is the sorted array.



Sorted array

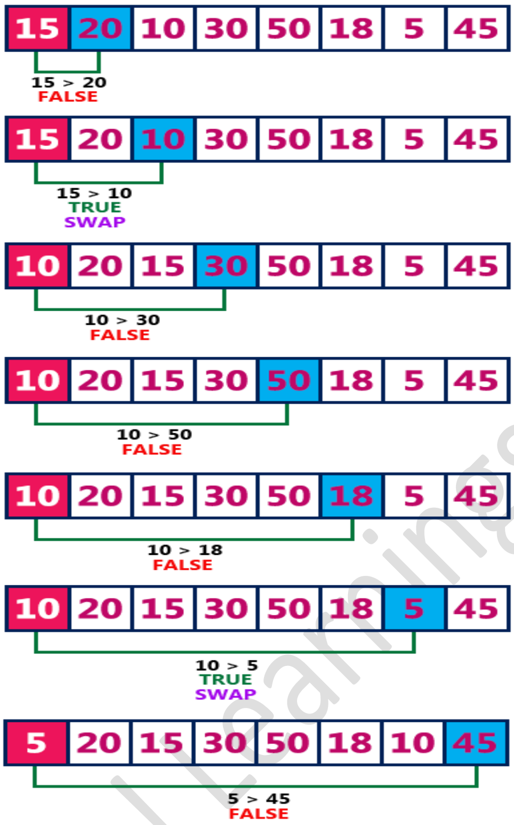
Example 2

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.



15	20	10	30	50	18	5	45

15 > 20
FALSE

15	20	10	30	50	18	5	45

15 > 10
TRUE
SWAP

10	20	15	30	50	18	5	45

10 > 30
FALSE

10	20	15	30	50	18	5	45

10 > 50
FALSE

10	20	15	30	50	18	5	45

10 > 18
FALSE

10	20	15	30	50	18	5	45

10 > 5
TRUE
SWAP

5	20	15	30	50	18	10	45

5 > 45
FALSE

List after 1st iteration

5	20	15	30	50	18	10	45

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

5	10	20	30	50	18	15	45

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

5	10	15	30	50	20	18	45

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

5	10	15	18	50	30	20	45

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

5	10	15	18	20	50	30	45

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

5	10	15	18	20	30	50	45

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration

5	10	15	18	20	30	45	50

Final sorted list

Implementation of Selection Sort

```
// C program for implementation of selection sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}
void selectionSort(int arr[], int n)
{
        int i, j, min_idx;
        // One by one move boundary of unsorted subarray
        for (i = 0; i < n-1; i++)
        {
                // Find the minimum element in unsorted array
                min_idx = i;
                for (j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                        min_idx = j;
                // Swap the found minimum element with the first element
                if(min_idx != i)
                        swap(&arr[min_idx], &arr[i]);
        }
}
/* Function to print an array */
void printArray(int arr[], int size)
{
        int i;
        for (i=0; i < size; i++)
                printf("%d ", arr[i]);
        printf("\n");
}
// Driver program to test above functions
int main()
{
        int arr[] = {64, 25, 12, 22, 11};
        int n = sizeof(arr)/sizeof(arr[0]);
        selectionSort(arr, n);
        printf("Sorted array: \n");
        printArray(arr, n);
        return 0;
}
```

Output

```
Sorted array:

11 12 22 25 64
```

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is O(N2) as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)

- Therefore overall complexity = O(N) * O(N) = O(N*N) = O(N2)

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Insertion Sort

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.

Insertion Sort Algorithm

To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Step by Step Process

Step 1 - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.

Step 2: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.

Step 3: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Insertion Sort Logic

```
  for i = 1 to size-1
{
    temp = list[i];
    j = i-1;
    while ((temp < list[j]) && (j > 0)) {
      list[j] = list[j-1];
      j = j - 1;
```

```
        }
        list[j] = temp;
      }
```

Working of Insertion Sort algorithm

Consider an example: arr[]: **{12, 11, 13, 5, 6}**

12	**11**	**13**	**5**	**6**

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12	**11**	**13**	**5**	**6**

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11	**12**	**13**	**5**	**6**

Second Pass:

- Now, move to the next two elements and compare them

11	**12**	**13**	**5**	**6**

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are **11** and **12**
- Moving forward to the next two elements which are 13 and 5

11	**12**	**13**	**5**	**6**

- Both 5 and 13 are not present at their correct place so swap them

11	**12**	**5**	**13**	**6**

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	**5**	**12**	**13**	**6**

- Here, again 11 and 5 are not sorted, hence swap again

5	**11**	12	13	6

- Here, 5 is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are **5, 11** and **12**
- Moving to the next two elements 13 and 6

5	11	12	**13**	**6**

- Clearly, they are not sorted, thus perform swap between both

5	11	12	**6**	**13**

- Now, 6 is smaller than 12, hence, swap again

5	11	**6**	**12**	13

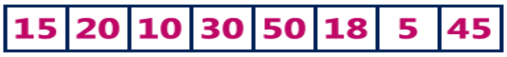
- Here, also swapping makes 11 and 6 unsorted hence, swap again

5	**6**	**11**	12	13

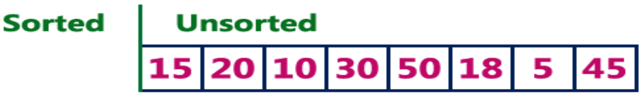
- Finally, the array is completely sorted.

Example

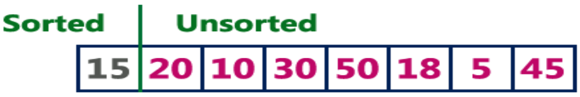
Consider the following unsorted list of elements...



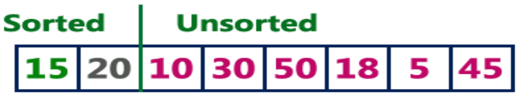
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



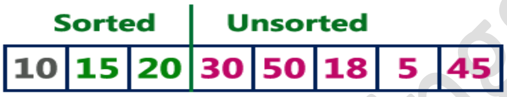
Move the first element 15 from unsorted portion to sorted portion of the list.



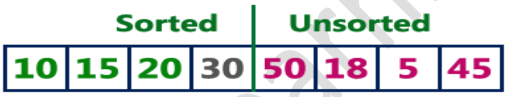
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position



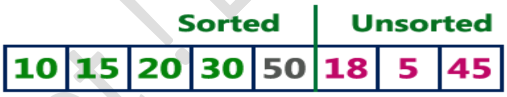
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



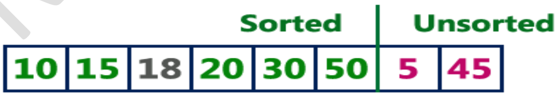
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



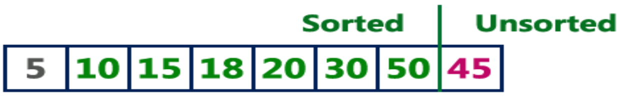
To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



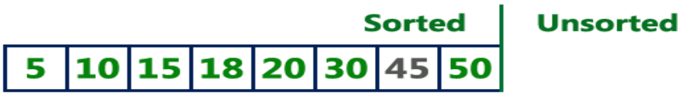
To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



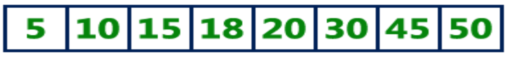
To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.



To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...



Implementation of Insertion Sort Algorithm

```
// C program for insertion sort
#include <math.h>
#include <stdio.h>
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
        int i, key, j;
        for (i = 1; i < n; i++)
        {
                key = arr[i];
                j = i - 1;
/* Move elements of arr[0..i-1], that are greater than key, to one
position ahead of their current position */
                while (j >= 0 && arr[j] > key)
                {
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
                arr[j + 1] = key;
        }
}
// A utility function to print an array of size n
void printArray(int arr[], int n)
{
        int i;
        for (i = 0; i < n; i++)
                printf("%d ", arr[i]);
        printf("\n");
}
/* Driver program to test insertion sort */
int main()
{
        int arr[] = { 12, 11, 13, 5, 6 };
        int n = sizeof(arr) / sizeof(arr[0]);
        insertionSort(arr, n);
        printArray(arr, n);
        return 0;
}
```

Output

```
5 6 11 12 13
```

Complexity Analysis of Insertion Sort:

Time Complexity: O(N^2)

Auxiliary Space: O(1)

Time Complexity of Insertion Sort

- The **worst-case** time complexity of the Insertion sort is **O(N^2)**
- The **average case** time complexity of the Insertion sort is **O(N^2)**
- The time complexity of the best case is **O(N).**

Space Complexity of Insertion Sort

The auxiliary space complexity of Insertion Sort is **O(1)**

Characteristics of Insertion Sort

- This algorithm is one of the simplest algorithms with a simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted.

UNIT-II

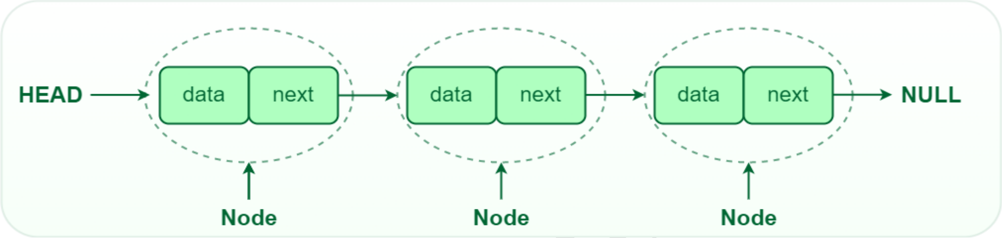
Linked Lists: Singly linked lists: representation and operations, doubly linked lists and circular linked lists, Comparing arrays and linked lists, Applications of linked lists.

LINKED LISTS

Linked List is a linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers. Linked List forms a series of connected nodes, where each node stores the data and the address of the next node

A linked list is ordered collection of finite Homogeneous data elements called **nodes** where the linear order is maintained by means of links or pointers.

A **linked list** is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.



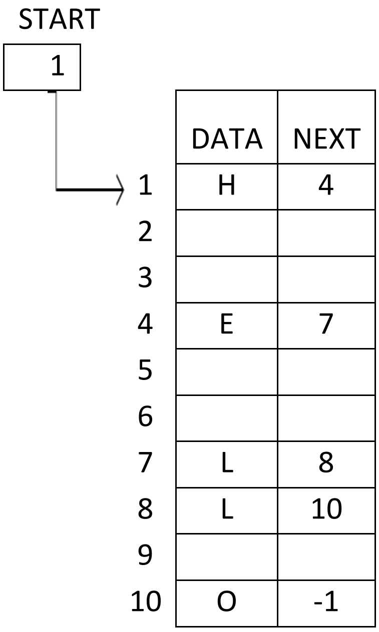
Node Structure: A node in a linked list typically consists of two components:

Data: It holds the actual value or data associated with the node.

Next Pointer: It stores the memory address (reference) of the next node in the sequence.

Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Memory Representation of Linked Lists



START

1

	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	-1

Let us see how a linked list is maintained in the memory, in order to form a linked list, we need a structure called node which has two fields, DATA and NEXT. Data will store the information part and next will store the address of the node in sequence. We see that the variable START is used to store the address of the first node.

Why linked list data structure needed?

- **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Example:

- In a system, if we maintain a sorted list of IDs in an array id[] = [1000, 1010, 1050, 2000, 2040].
- If we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
- Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved due to this so much work is being done which affects the efficiency of the code.

Types of Linked Lists

- Single linked list
- Double linked list
- Circular linked list

Representation of a Linked List

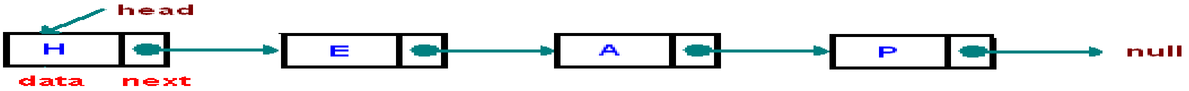
There are two ways to represent a linked list in memory:

1. Static representation using array
2. Dynamic representation using free pool of storage

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct definition that contains variables holding information about something and that has a pointer to a struct of its same type (it has to be a pointer--otherwise, every time an element was created, it would create a new element, infinitely). Each of these individuals structs or classes in the list is commonly known as a node or element of the list.

Head node in linked list

The entry point into a **linked list** is called the **head** of the **list**. It should be noted that **head** is not a separate **node**, but the reference to the first **node**. If the **list** is empty then the **head** is a null reference. Li**nked list** is a dynamic data structure.



Node is represented as

```
Struct node
{
int data;
struct node *next;
}
```

Linked List Operations

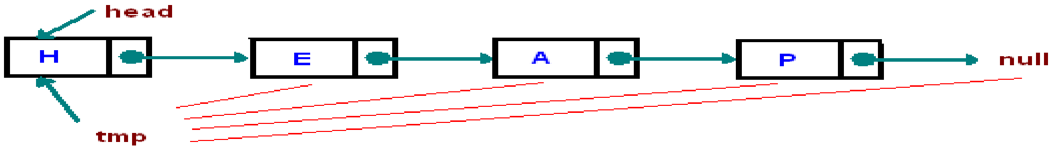
- **Display/ Traverse**– Displays the complete list.
- **Search** − Searches an element using the given key.
- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.

Traversing/Display

Start with the head and access each node until you reach null. Do not change the head reference

Algorithm for Traversing a Linked List

```
Step 1:    [Initialize] Set Ptr = Start
Step 2:    Repeat Steps 3 &4 While Ptr != Null
Step 3:    Apply Process to Ptr->Data
Step 4:    Set Ptr = Ptr->Next
        [End Of Loop]
Step 5:    Exit
```



```
#include<stdio.h>
#include<stdlib.h>
void createNode(int);
void traverse();
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("\n1.Insert Node\n2.Traverse\n3.Exit\n4.Enter your
choice:\n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the element to insert:\n");
         scanf("%d",&item);
         createNode(item);
         break;
         case 2:
         traverse();
         break;
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease enter a valid choice.\n");
      }
    }while(choice != 3);
}
void createNode(int item)
   {
      struct node *ptr = (struct node *)malloc(sizeof(struct node *));
      if(ptr == NULL)
      {
         printf("\nOVERFLOW\n");
      }
      else
      {
         ptr->data = item;
         ptr->next = head;
         head = ptr;
         printf("\nNode inserted successfully!!\n");
      }
    }
void traverse()
```

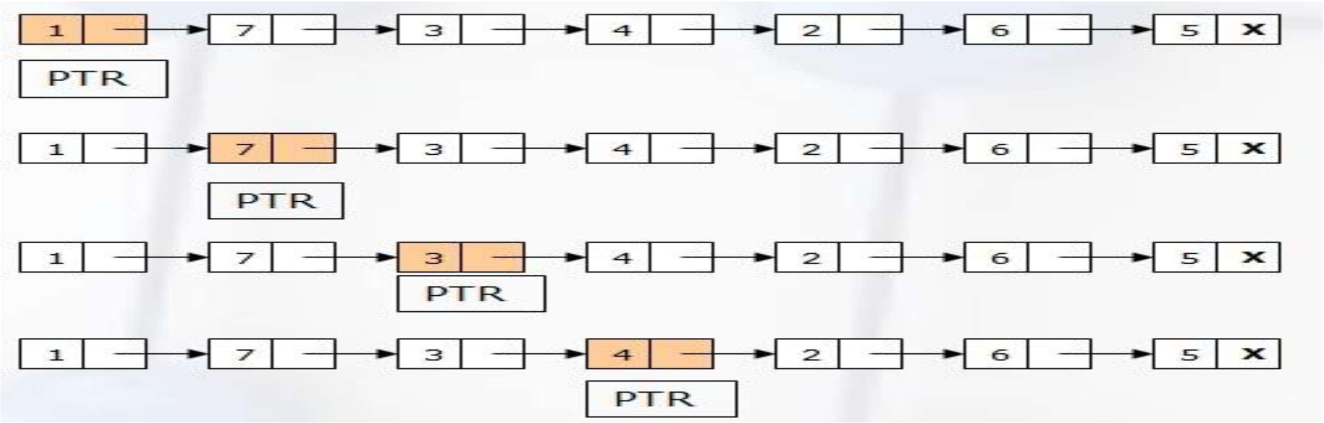
Output:

```
1.Insert Node
2.Traverse
3.Exit
4.Enter your choice:
1

Enter the element to insert:
2

Node inserted successfully!!

1.Insert Node
2.Traverse
3.Exit
4.Enter your choice:
1

Enter the element to insert:
4

Node inserted successfully!!

1.Insert Node
2.Traverse
3.Exit
4.Enter your choice:
1

Enter the element to insert:
6

Node inserted successfully!!

1.Insert Node
2.Traverse
3.Exit
4.Enter your choice:
2
printing values . . . . .

6
4
2
1.Insert Node
2.Traverse
3.Exit
4.Enter your choice:
1

Enter the element to insert:
  {
     struct node *ptr;
     ptr = head;
     if(ptr == NULL)
     {
        printf("List is Empty.");
     }
     else
     {
        printf("printing values . . . . .\n");
        while (ptr!=NULL)
        {
           printf("\n%d",ptr->data);
           ptr = ptr -> next;
        }
     }
   }
```

```
8

Node inserted successfully!!

1.Insert Node
2.Traverse
3.Exit
4.Enter your choice:
2
printing values . . . . .

8
6
4
2
1.Insert Node
2.Traverse
3.Exit
4.Enter your choice:
3
```

Algorithm to Search a Linked List

```
Step 1: [Initialize] Set Ptr = Start
Step 2: Repeat Step 3 While Ptr! = Null
Step 3:    If Val = Ptr->Data
           Set Pos = Ptr
           Go To Step 5
           Else
           Set Ptr = Ptr->Next
           [End of If]
[End of Loop]
Step 4: Set Pos = Null
Step 5: Exit
```



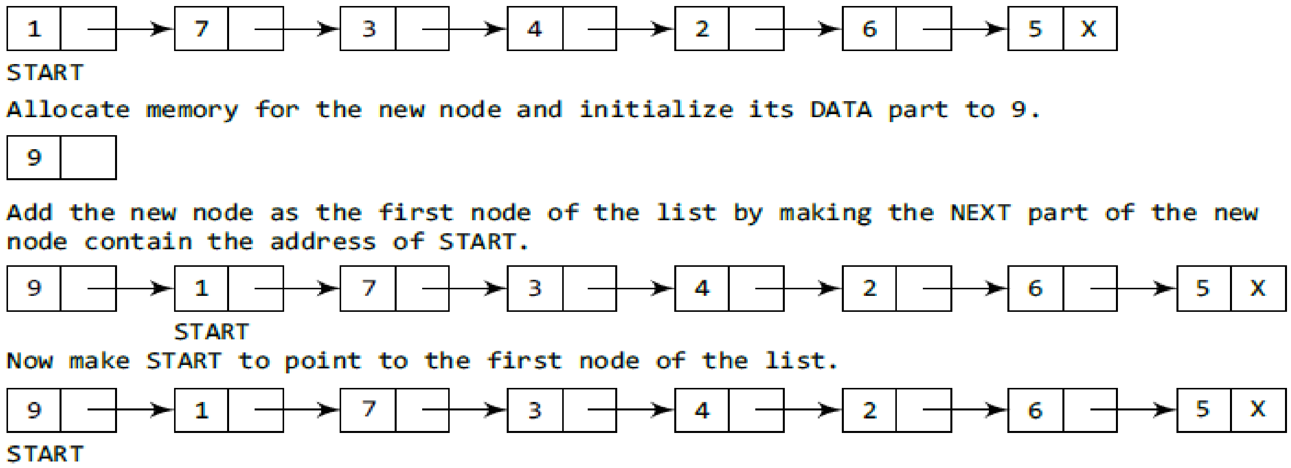
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
void addLast(struct node **head, int val)
{
    //create a new node
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    newNode->next     = NULL;
    //if head is NULL, it is an empty list
    if(*head == NULL)
         *head = newNode;
    //Otherwise, find the last node and add the newNode
    else
    {
        struct node *lastNode = *head;
        //last node's next address will be NULL.
        while(lastNode->next != NULL)
        {
            lastNode = lastNode->next;
        }
        //add the newNode at the end of the linked list
        lastNode->next = newNode;
    }
}
int searchNode(struct node *head,int key)
{
    struct node *temp = head;
    //iterate the entire linked list and print the data
    while(temp != NULL)
    {
         //key found return 1.
         if(temp->data == key)
             return 1;
         temp = temp->next;
    }
    //key not found
    return -1;
}
int main()
{
     struct node *head = NULL;
     addLast(&head,10);
     addLast(&head,20);
     addLast(&head,30);
     //change the key and try it yourself.
     if(searchNode(head,20) == 1)
         printf("Search Found\n");
    else
       printf("Search Not Found\n");
    return 0;
}
```

Inserting a Node at the Beginning

Suppose we want to add a new node with the data 9 and add it as the first node of the list. Then the changes will do in the linked list. Allocate the memory for the new node and initialize its DATA part to 9. Add the new node as the first node of the list by making the NEXT part of the new node contain the address of the START. Now make START to point to the first node of the list.

Algorithm to Insert a New Node in the Beginning

```
Step 1: if avail = null,
            Write overflow
            Go to step 7
[End of if]
Step 2: set new_node = avail
Step 3: set avail = avail->next
Step 4: set new_node->data = val
Step 5: set new_node->next = start
Step 6: set start = new_node
Step 7: exit
```



Allocate memory for the new node and initialize its DATA part to 9.

Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.

Now make START to point to the first node of the list.

```
// C program to show inserting a node at front of given Linked List
#include <stdio.h>
#include <stdlib.h>
// A linked list node
struct Node
{
            int data;
            struct Node* next;
};
```

```
// Given a reference (pointer to pointer) to the head of a list and an
int, inserts a new node on the front of the list.
void insertAtFront(struct Node** head_ref, int new_data)
{
        // 1. allocate node
        struct Node* new_node
                = (struct Node*)malloc(sizeof(struct Node));
        // 2. put in the data
        new_node->data = new_data;
        // 3. Make next of new node as head
        new_node->next = (*head_ref);
        // 4. move the head to point to the new node
        (*head_ref) = new_node;
}

// This function prints contents of linked list starting from head
void printList(struct Node* node)
{
        while (node != NULL)
        {
                printf(" %d", node->data);
                node = node->next;
        }
        printf("\n");
}
// Driver code
int main()
{
        // Start with the empty list
        struct Node* head = NULL;
        insertAtFront(&head, 1);
        insertAtFront(&head, 2);
        insertAtFront(&head, 3);
        insertAtFront(&head, 4);
        insertAtFront(&head, 5);
        insertAtFront(&head, 6);
        printf("After inserting nodes at their front: ");
        printList(head);
        return 0;
}
```

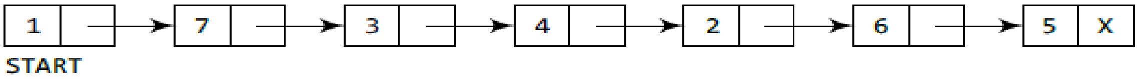
Time Complexity: O(1)
Auxiliary Space: O(1)

Inserting a Node at the End

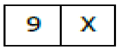
If we want to add a node with data 9 as the last node of the list,Then insert a new node at the end of a linked list. Take pointer variable initialize with START. That is the pointer now points to the first node of the linked list. With the help of loop will traverse through the linked list to reach the last node. Once reached the last node the next of the last node to store the address of the new node, then last node next will contains NULL.

Algorithm to Insert a New Node at the End of the Linked List

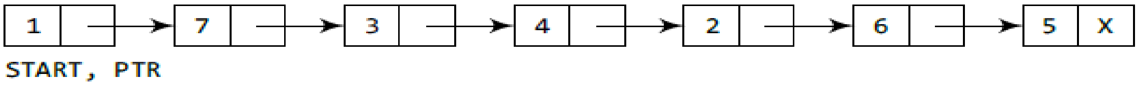
```
Step 1: if avail = null, then
            Write overflow
            go to step 10
            [end of if]
Step 2: set new_node = avail
Step 3: set avail = avail->next
Step 4: set new_node->data = val
Step 5: set new_node->next = null
Step 6: set ptr = start
Step 7: repeat step 8 while ptr->next! = null
Step 8:  set ptr = ptr ->next
 [End of loop]
Step 9: set ptr->next = new_node
Step 10: exit
```



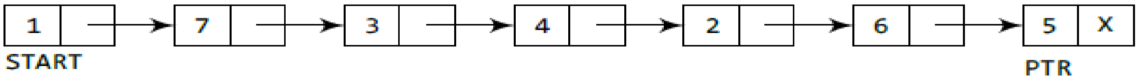
Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.



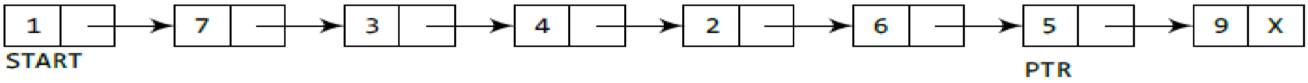
Take a pointer variable PTR which points to START.



Move PTR so that it points to the last node of the list.



Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.



```
#include <stdio.h>
#include <stdlib.h>
// A linked list node
struct Node
{
            int data;
            struct Node* next;
};
// Given a reference (pointer to pointer) to the head of a list and an int,
inserts a new node on the front of the list.
void push(struct Node** head_ref, int new_data)
{
            // Create a new node
            struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
            new_node->data = new_data;
```

Output

```
Created Linked list is:
2 3 4 5 6
After inserting 1 at the
end:  2 3 4 5 6 1
```

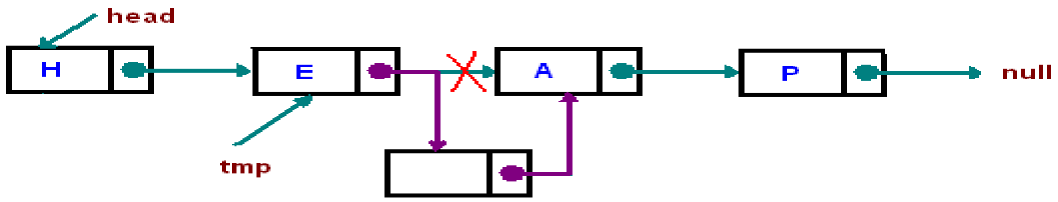
```
    // Make the new node point to the current head
    new_node->next = (*head_ref);
    // Update the head to point to the new node
    (*head_ref) = new_node;
}
// Given a reference (pointer to pointer) to the head of a list and an int, appends a new node at the end
void append(struct Node** head_ref, int new_data)
{
    // Create a new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    // Store the head reference in a temporary variable
    struct Node* last = *head_ref;
// Set the next pointer of the new node as NULL since it will be the last node
    new_node->next = NULL;
// If the Linked List is empty, make the new node as the head and return
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }
    // Else traverse till the last node
    while (last->next != NULL)
     {
        last = last->next;
    }
// Change the next pointer of the last node to point to  the new node
    last->next = new_node;
}
// This function prints the contents of the linked list starting from the head
void printList(struct Node* node)
{
    while (node != NULL) {
        printf(" %d", node->data);
        node = node->next;
    }
}
// Driver code
int main()
{
    // Start with an empty list
    struct Node* head = NULL;
    // Insert nodes at the beginning of the linked list
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    printf("Created Linked list is: ");
    printList(head);
```

```
        // Insert 1 at the end
        append(&head, 1);

        printf("\nAfter inserting 1 at the end: ");
        printList(head);

        return 0;
}
```

Inserting a Node after Node

ALGORITHM TO INSERT A NEW NODE AFTER A NODE

```
Step 1: if avail = null, then
            Write overflow
            Go to step 12
        [End of if]
Step 2: set new_node = avail
Step 3: set avail = avail->next
Step 4: set new_node->data = val
Step 5: set ptr = start
Step 6: set preptr = ptr
Step 7: repeat steps 8 and 9 while preptr->data != num
Step 8: set preptr = ptr
Step 9: set ptr = ptr->next
        [end of loop]
Step 10: preptr->next = new_node
Step 11: set new_node->next = ptr
Step 12: exit
```



```
// C program to show inserting a node after a given node in given Linked List
#include <stdio.h>
#include <stdlib.h>
 // A linked list node
struct Node {
    int data;
    struct Node* next;
};
 // Given a node prev_node, insert a new  node after the given prev_node
void insertAfter(struct Node* prev_node, int new_data)
{
```

Output

```
Created Linked list is:  2 3 4 5 6

After inserting 1 after 2:  2 1 3 4 5 6
```

```
    // 1. check if the given prev_node is NULL
    if (prev_node == NULL)
{
        printf("The given previous node cannot be NULL");
        return;
    }
    // 2. allocate new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    // 3. put in the data
    new_node->data = new_data;
    // 4. Make next of new node as next of prev_node
    new_node->next = prev_node->next;
     // 5. move the next of prev_node as new_node
    prev_node->next = new_node;
}
 // Function to insert element in LL
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// This function prints contents of linked list starting from head
void printList(struct Node* node)
{
    while (node != NULL)
            {
        printf(" %d", node->data);
        node = node->next;
    }
    printf("\n");
}
 // Driver code
int main()
{
    // Start with the empty list
    struct Node* head = NULL;
     push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);

    printf("Created Linked list is: ");
    printList(head);

    // Insert 1 at the beginning.
    insertAfter(head, 1);

    printf("After inserting 1 after 2: ");
```

```
   printList(head);

   return 0;
}
```

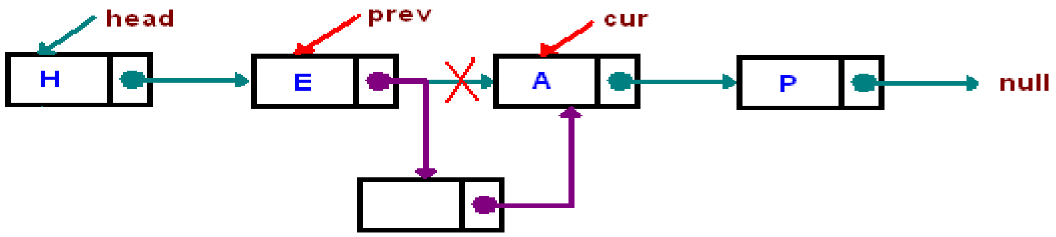
Time Complexity: O(1)
Auxiliary Space: O(1)

INSERTING "BEFORE"

ALGORITHM TO INSERT A NEW NODE BEFORE A NODE

```
Step 1: if avail = null, then
            Write overflow
            Go to step 12
        [End of if]
Step 2: set new_node = avail
Step 3: set avail = avail->next
Step 4: set new_node->data = val
Step 5: set ptr = start
Step 6: set preptr = ptr
Step 7: repeat steps 8 and 9 while ptr->data != num
Step 8: set preptr = ptr
Step 9: set ptr = ptr->next
        [end of loop]
Step 10: preptr->next = new_node
Step 11: set new_node->next = ptr
Step 12: exit
```

Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "a":



For the sake of convenience, we maintain two references prev and cur. When we move along the list we shift these two references, keeping prev one step before cur. We continue until cur reaches the node before which we need to make an insertion. If cur reaches null, we don't insert, otherwise we insert a new node between prev and cur.

```
// C program to show inserting a node at front of given Linked List
#include <stdio.h>
#include <stdlib.h>
// A linked list node
struct Node
```

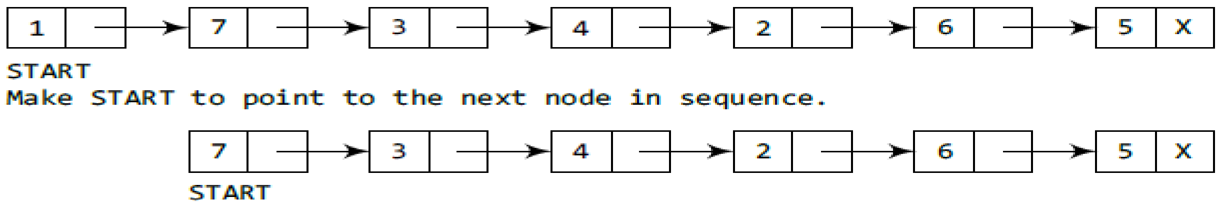
```
{
        int data;
        struct Node* next;
};

// Given a reference (pointer to pointer)to the head of a list and an int, inserts a new node on the front of the list.
void insertAtFront(struct Node** head_ref, int new_data)
{
        // 1. allocate node
        struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
        // 2. put in the data
        new_node->data = new_data;
        // 3. Make next of new node as head
        new_node->next = (*head_ref);
        // 4. move the head to point to the new node
        (*head_ref) = new_node;
}
// This function prints contents of linked list starting from head
void printList(struct Node* node)
{
        while (node != NULL)
        {
                printf(" %d", node->data);
                node = node->next;
        }
        printf("\n");
}
// Driver code
int main()
{
        // Start with the empty list
        struct Node* head = NULL;

        insertAtFront(&head, 1);
        insertAtFront(&head, 2);
        insertAtFront(&head, 3);
        insertAtFront(&head, 4);
        insertAtFront(&head, 5);
        insertAtFront(&head, 6);

        printf("After inserting nodes at their front: ");
        printList(head);

        return 0;
}
```

Deleting the First Node

Algorithm to Delete the First Node from the Linked List

```
Step 1: if start = null, then
          Write underflow
          Go to step 5
```

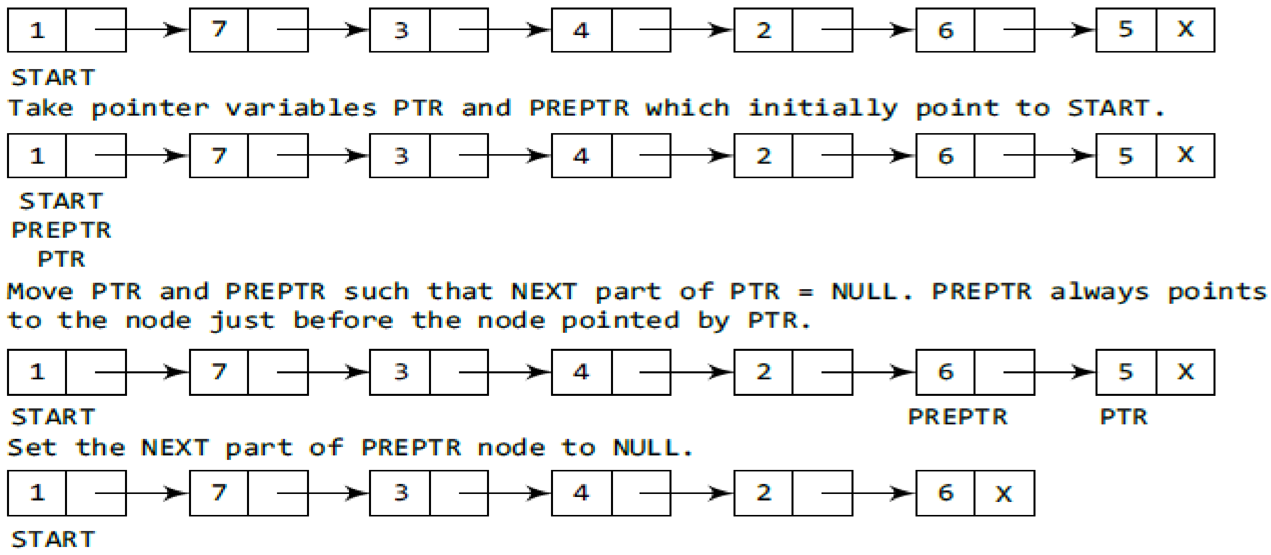
```
        [End of if]
Step 2: set ptr = start
Step 3: set start = start->next
Step 4: free ptr
Step 5: exit
```



Make START to point to the next node in sequence.

ALGORITHM TO DELETE THE LAST NODE

```
Step 1: if start = null, then
            Write underflow
            Go to step 8
        [End of if]
Step 2: set ptr = start
Step 3: repeat steps 4 and 5 while ptr->next != null
Step 4:     set preptr = ptr
Step 5:     set ptr = ptr->next
            [End of loop]
Step 6: set preptr->next = null
Step 7: free ptr
Step 8: exit
```



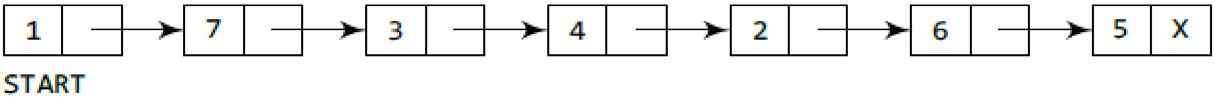
Take pointer variables PTR and PREPTR which initially point to START.

Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.

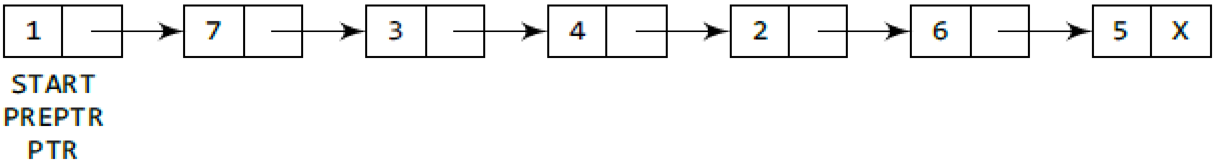
Set the NEXT part of PREPTR node to NULL.

ALGORITHM TO DELETE THE NODE AFTER A GIVEN NODE FROM THE LINKED LIST

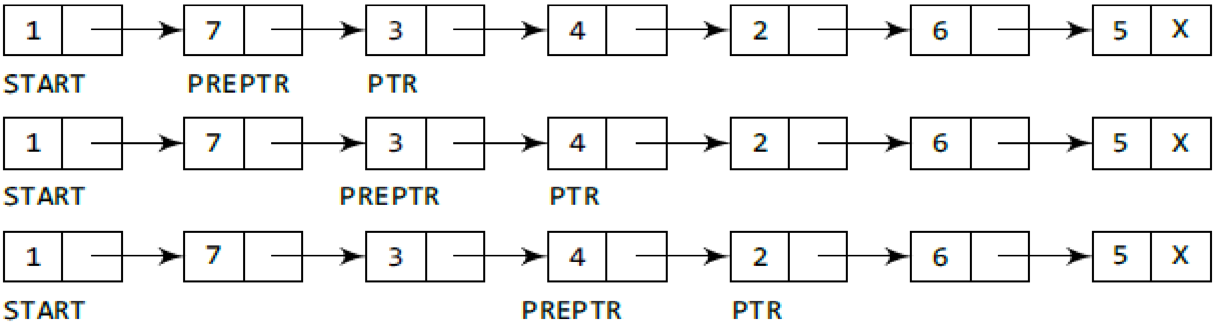
```
Step 1: if start = null, then
            Write underflow
            Go to step 10
            [end of if]
Step 2: set ptr = start
Step 3: set preptr = ptr
Step 4: repeat step 5 and 6 while pretr->data!=num
Step 5:     set preptr = ptr
Step 6:     set ptr = ptr->next
            [End of loop]
step7: set temp = ptr->next
Step 8: set preptr->next = temp->next
Step 9: free temp
Step 10: exit
```



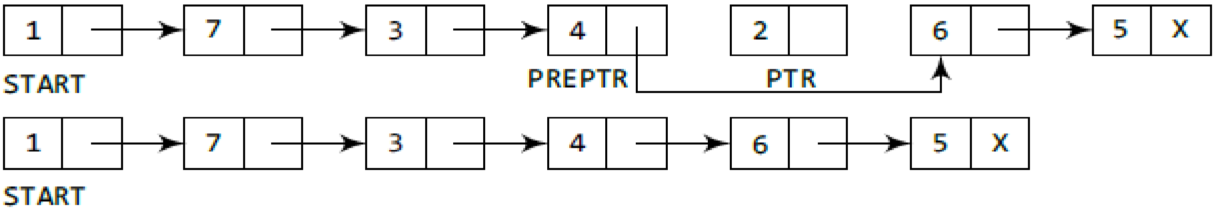
Take pointer variables PTR and PREPTR which initially point to START.



Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.



Set the NEXT part of PREPTR to the NEXT part of PTR.



```
/* Program: Deleting a node in the linked list Language: C */
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
void addLast(struct node **head, int val)
{
   //create a new node
   struct node *newNode = malloc(sizeof(struct node));
   newNode->data = val;
   newNode->next     = NULL;
   //if head is NULL, it is an empty list
   if(*head == NULL)
       *head = newNode;
   //Otherwise, find the last node and add the newNode
   else
   {
      struct node *lastNode = *head;
      //last node's next address will be NULL.
      while(lastNode->next != NULL)
      {
         lastNode = lastNode->next;
      }
      //add the newNode at the end of the linked list
      lastNode->next = newNode;
   }
void deleteNode(struct node **head, int key)
{
    //temp is used to freeing the memory
    struct node *temp;
//key found on the head node. //move to head node to the next and free the head.
    if((*head)->data == key)
    {
       temp = *head;    //backup to free the memory
       *head = (*head)->next;
       free(temp);
    }
    else
    {
       struct node *current  = *head;
       while(current->next != NULL)
       {
          //if yes, we need to delete the current->next node
          if(current->next->data == key)
          {
             temp = current->next;
             //node will be disconnected from the linked list.
             current->next = current->next->next;
```

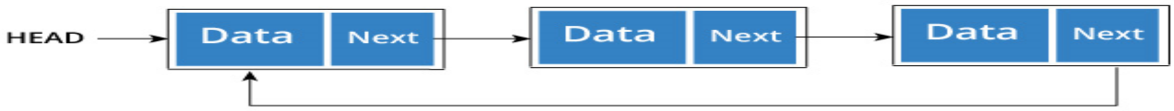
OutPut

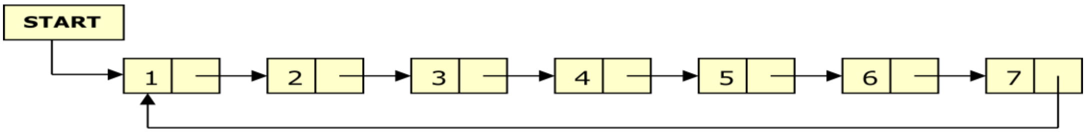
```
Linked List Elements:

10 ->20 ->30 ->NULL

Deleted 10. The New
Linked List:

20 ->30 ->NULL

Deleted 30. The New
Linked List:

20 ->NULL

Deleted 20. The New
Linked List:
NULL
```

```
                free(temp);
                break;
            }
            //Otherwise, move the current node and proceed
            else
                current = current->next;
        }
      }
}
void printList(struct node *head)
{
    struct node *temp = head;
    //iterate the entire linked list and print the data
    while(temp != NULL)
    {
         printf("%d ->", temp->data);
         temp = temp->next;
    }
    printf("NULL\n");
}
int main()
{
     struct node *head = NULL;

     addLast(&head,10);
     addLast(&head,20);
     addLast(&head,30);
     printf("Linked List Elements:\n");
     printList(head);

     //delete first node
     deleteNode(&head,10);
     printf("Deleted 10. The New Linked List:\n");
     printList(head);

     //delete last node
     deleteNode(&head,30);
     printf("Deleted 30. The New Linked List:\n");
     printList(head);

     //delete 20
     deleteNode(&head,20);
     printf("Deleted 20. The New Linked List:\n");
     printList(head);

     return 0;
}
```

Circular Linked List

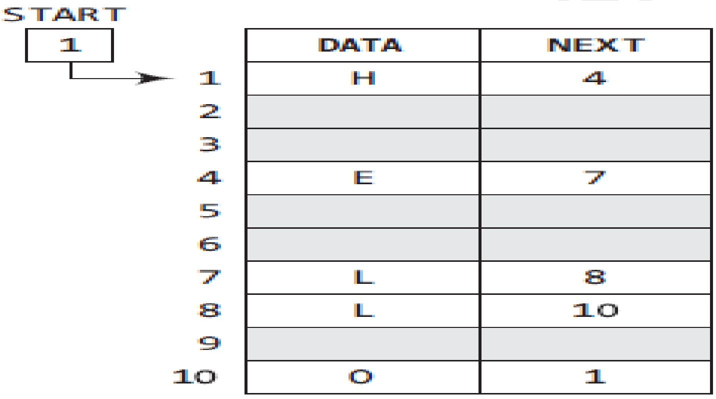
In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly listed list as well as circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward or backward until we reach the same node where we had started. Thus, a circular linked list has no beginning and no ending.





1. A circular linked list is a linked list in which the head element's previous pointer points to the tail element and the tail element's next pointer points to the head element.

2. A circularly linked list node looks exactly the same as a linear singly linked list.
A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.



	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

Memory representation
of a circular linked list

Circular linked lists are widely used in operating systems for task maintenance. We will now discuss an example where a circular linked list is used. When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited. How is this done? The answer is simple.

A circular linked list is used to maintain the sequence of the Web pages visited. Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons. Actually, this is done using either the circular stack or the circular queue.

Advantages:

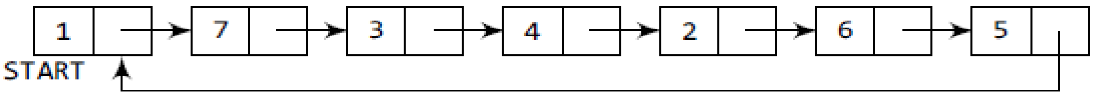
- ➢ Any node can be traversed starting from any other node in the list.
- ➢ There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.
- ➢ In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that item itself.

Insert a new node in the beginning of circular the linked list

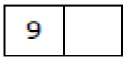
We will see how a new node is added into an already existing linked list. We will take two cases and then see how insertion is done in each case.
Case 1: The new node is inserted at the beginning of the circular linked list.
Case 2: The new node is inserted at the end of the circular linked list.

Algorithm to insert a new node in the beginning of circular the linked list

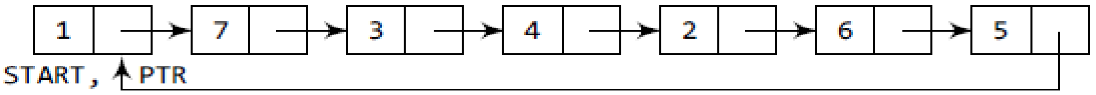
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
       [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```



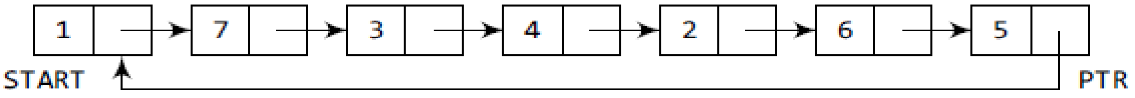
Allocate memory for the new node and initialize its DATA part to 9.



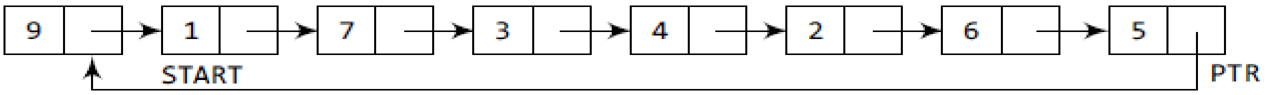
Take a pointer variable PTR that points to the START node of the list.



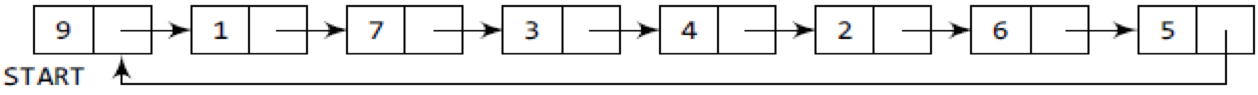
Move PTR so that it now points to the last node of the list.



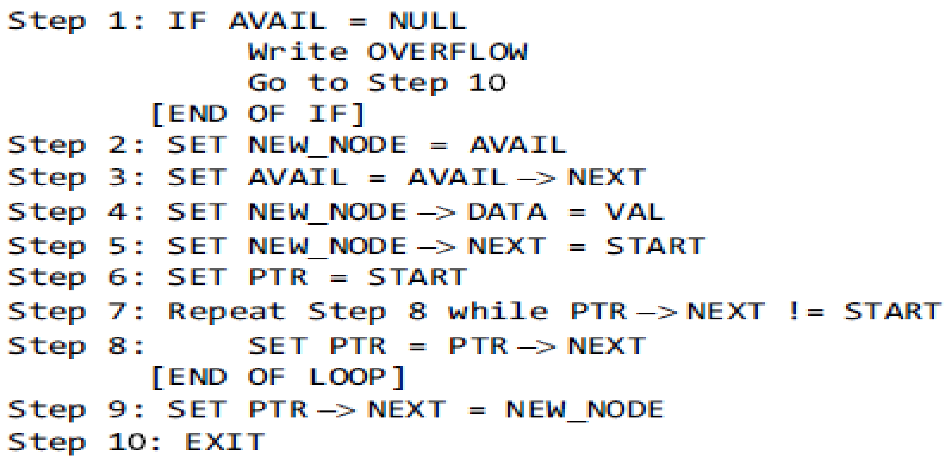
Add the new node in between PTR and START.



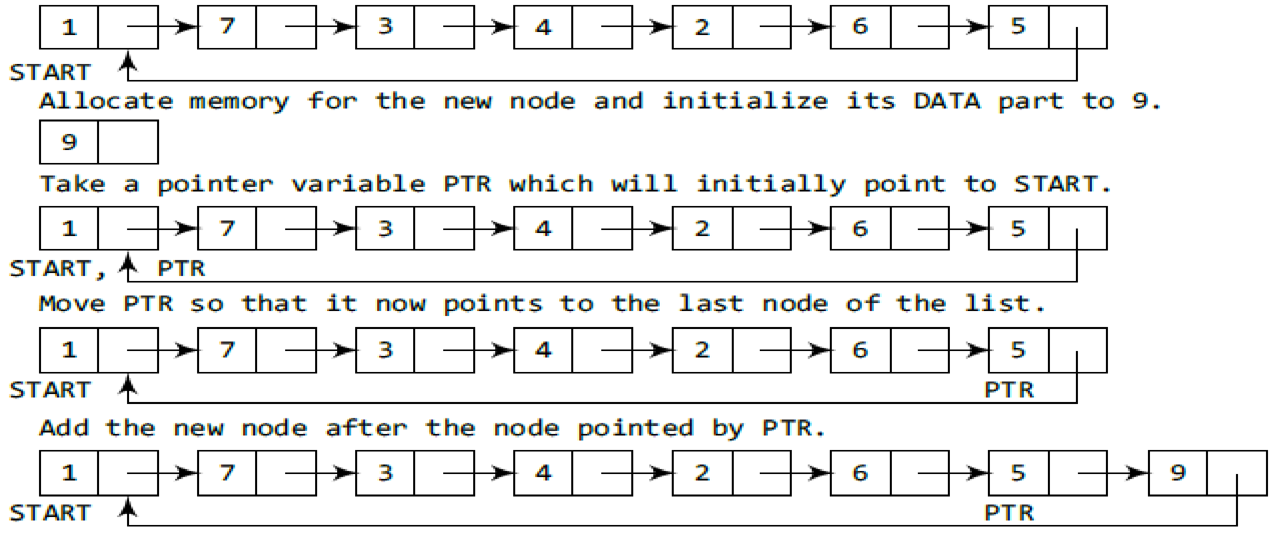
Make START point to the new node.



Algorithm to insert a new node at the end of the circular linked list



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```



Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR which will initially point to START.

Move PTR so that it now points to the last node of the list.

Add the new node after the node pointed by PTR.

Inserting a new node at the end of a circular linked list take pointer variable and initialize with start, the pointer points the first node of the circular linked list, with the help of the loop traverse through the linked list to reach the last node, once reached the last node change the pointer next of the last node to store the address of the first node which is denoted by START.

```
// C program for the above operation
#include <stdio.h>
#include <stdlib.h>
// Structure of a linked list node
struct node {
        int info;
        struct node* next;
};
// Pointer to last node in the list
struct node* last = NULL;
// Function to insert a node in the starting of the list
void insertAtFront(int data)
{
        // Initialize a new node
        struct node* temp;
        temp = (struct node*)malloc(sizeof(struct node));
        // If the new node is the only node in the list
        if (last == NULL)
```

OutPut

```
Data = 30
Data = 20
Data = 10
```

```
        {
                temp->info = data;
                temp->next = temp;
                last = temp;
        }
// Else last node contains the reference of the new node and new node
contains the reference of the previous first node
        else
        {
                temp->info = data;
                temp->next = last->next;
                // last node now has reference of the new node temp
                last->next = temp;
        }
}
// Function to print the list
void viewList()
{
        // If list is empty
        if (last == NULL)
                printf("\nList is empty\n");
        // Else print the list
        else {
                struct node* temp;
                temp = last->next;
// While first node is not reached again, print, since the list is circular
                do
                {
                        printf("\nData = %d", temp->info);
                        temp = temp->next;
                } while (temp != last->next);
        }
}
// Driver Code
int main()
{
        // Function Call
        insertAtFront(10);
        insertAtFront(20);
        insertAtFront(30);

        // Print list
        viewList();

        return 0;
}
```

Delete the node from the circular linked list

Deleting a node form the circular linked list will take two cases and then see how deletion is done in each case. Rest of the cases of deletion is same as that given for singly linked list lists.

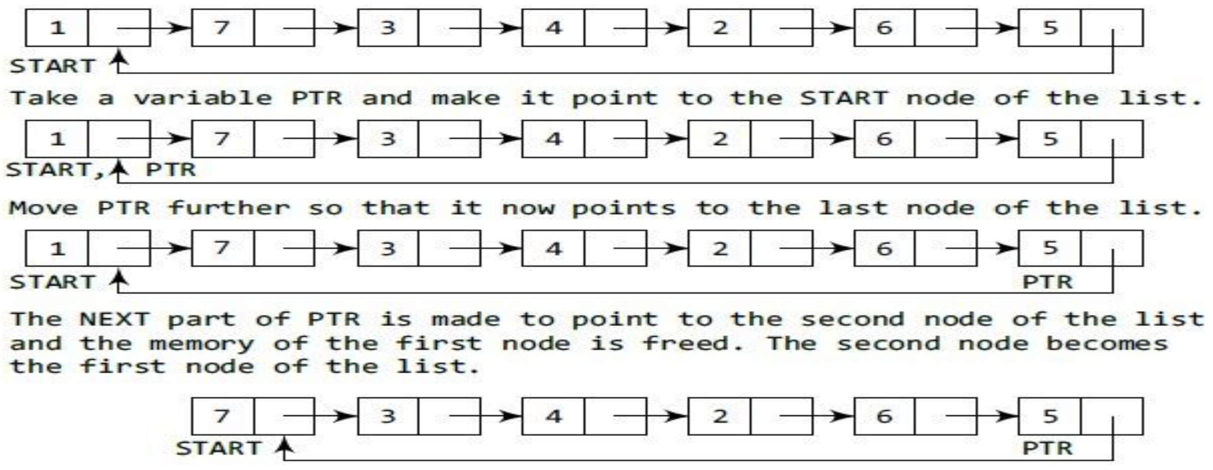
Case 1: the first node is deleted

Case 2: the last node is deleted

Algorithm to delete the first node from the circular linked list

```
Step 1: If Start = Null, Then
                Write Underflow
                Go To Step 8
        [End Of If]
Step 2: Set Ptr = Start
Step 3: Repeat Step 4 While Ptr->Next != Start
Step 4:         Set Ptr = Ptr->Next
        [End of If]
Step 5: Set Ptr->Next = Start->Next
Step 6: Free Start
Step 7: Set Start = Ptr->Next
Step 8: Exit
```

Consider the circular linked list when we want to delete a node from the beginning of the list, then the take a variable pointer and make it point to the START node of the list and move pointer further so that it now points to the last node of the list. The NEXT part of the pointer is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.



Take a variable PTR and make it point to the START node of the list.

Move PTR further so that it now points to the last node of the list.

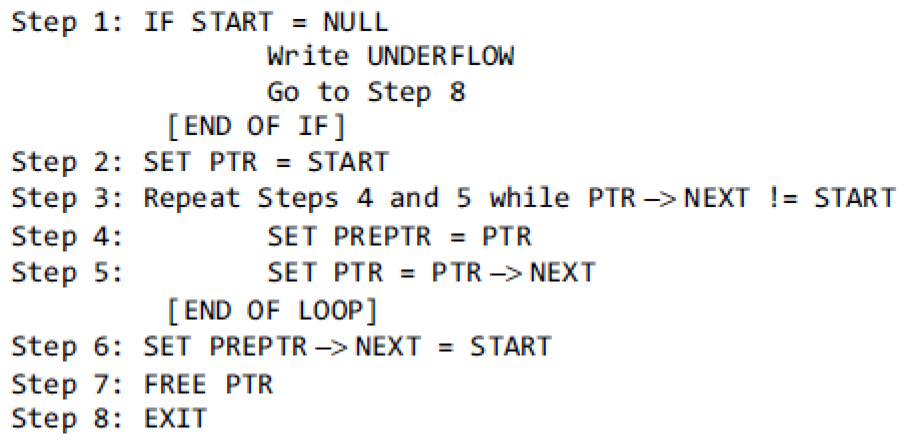
The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.

Deleting the first node from a circular linked list

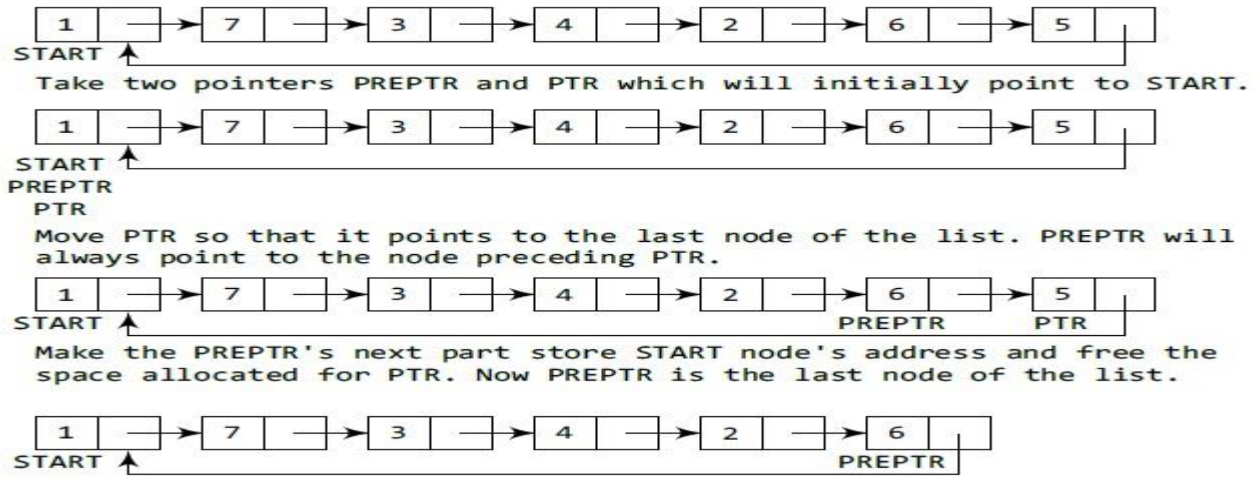
Deleting the Last Node from a Circular Linked List

Consider the circular linked list suppose we want to delete the last nodefrom the linked list, then the following changes will be done in the linked list.

Algorithm to delete the last node from the circular linked list



```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```



Take two pointers PREPTR and PTR which will initially point to START.

Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.

Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.

Deleting the last node from a circular linked list

```
/* Program: Deleting a node in the linked list Language: C */
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
void addLast(struct node **head, int val)
{
   //create a new node
   struct node *newNode = malloc(sizeof(struct node));
   newNode->data = val;
   newNode->next     = NULL;
   //if head is NULL, it is an empty list
   if(*head == NULL)
        *head = newNode;
   //Otherwise, find the last node and add the newNode
   else
   {
       struct node *lastNode = *head;
       //last node's next address will be NULL.
```

Output

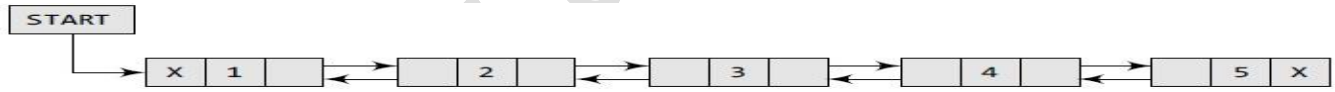
```
Linked List Elements:
10 ->20 ->30 ->NULL
Deleted 10. The New Linked
List:
20 ->30 ->NULL
Deleted 30. The New Linked
List:
20 ->NULL
Deleted 20. The New Linked
List:
NULL
```

```
        while(lastNode->next != NULL)
        {
           lastNode = lastNode->next;
        }
        //add the newNode at the end of the linked list
        lastNode->next = newNode;
    }
}
void deleteNode(struct node **head, int key)
{
     //temp is used to freeing the memory
     struct node *temp;
     //key found on the head node.
     //move to head node to the next and free the head.
     if((*head)->data == key)
     {
        temp = *head;    //backup to free the memory
        *head = (*head)->next;
        free(temp);
     }
     else
     {
        struct node *current  = *head;
        while(current->next != NULL)
        {
           //if yes, we need to delete the current->next node
           if(current->next->data == key)
           {
              temp = current->next;
              //node will be disconnected from the linked list.
              current->next = current->next->next;
              free(temp);
              break;
           }
           //Otherwise, move the current node and proceed
           else
              current = current->next;
        }
     }
}
void printList(struct node *head)
{
   struct node *temp = head;
   //iterate the entire linked list and print the data
   while(temp != NULL)
   {
       printf("%d ->", temp->data);
       temp = temp->next;
   }
   printf("NULL\n");
}
int main()
```

```
{
    struct node *head = NULL;
    addLast(&head,10);
    addLast(&head,20);
    addLast(&head,30);
    printf("Linked List Elements:\n");
    printList(head);
    //delete first node
    deleteNode(&head,10);
    printf("Deleted 10. The New Linked List:\n");
    printList(head);
    //delete last node
    deleteNode(&head,30);
    printf("Deleted 30. The New Linked List:\n");
    printList(head);
    //delete 20
    deleteNode(&head,20);
    printf("Deleted 20. The New Linked List:\n");
    printList(head);
    return 0;
}
```

DOUBLY LINKED LISTS

A doubly linked list or a two-way linked list is a more complex type of linked list which containsa pointer to the next as well as the previous node in the sequence. Therefore, it consists of threeparts—data, a pointer to the next node, and a pointer to the previous node.



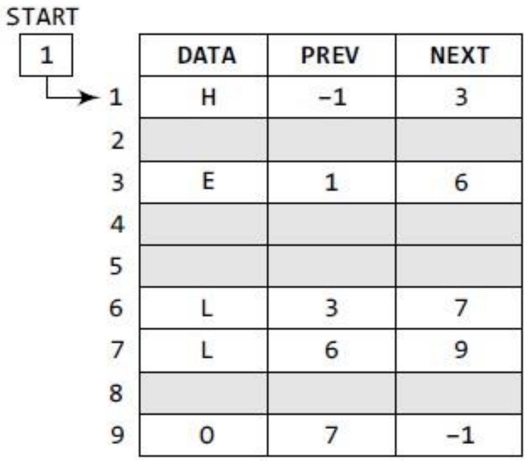
Doubly linked list

Doubly linked list can be representation as,

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

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV Field is used to store the address of the preceding node, which enables us to traverse the list in the Backward direction.

Thus, we see that a doubly linked list calls for more space per node and more expensive basic Operations. However, a doubly linked list provides the ease to manipulate the elements of the List as it maintains pointers to nodes in both the directions (forward and backward). The main Advantage of using a doubly linked list is that it makes searching twice as efficient. Let us view How a doubly linked list is maintained in the memory.



	DATA	PREV	NEXT
1	H	–1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	–1

Memory representation of a doubly linked list

We see that a variable START is used to store the address of the first node. In this
Example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field. We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list. When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Inserting a New Node in a Doubly Linked List

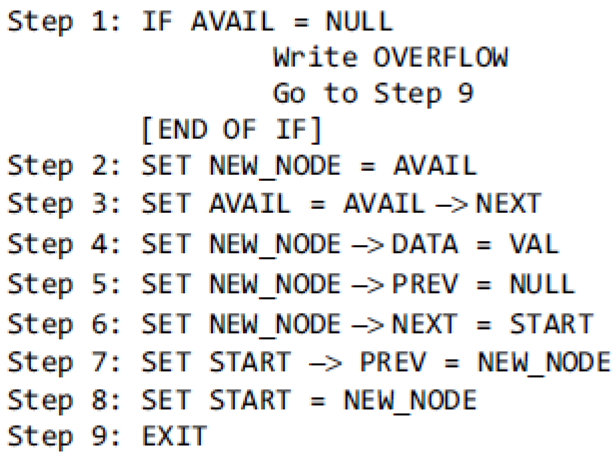
In this section, we will discuss how a new node is added into an already existing doubly linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.
Case 4: The new node is inserted before a given node.

Inserting a Node in a Doubly Linked List

To insert a new node at the beginning of a doubly linked list, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate Space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

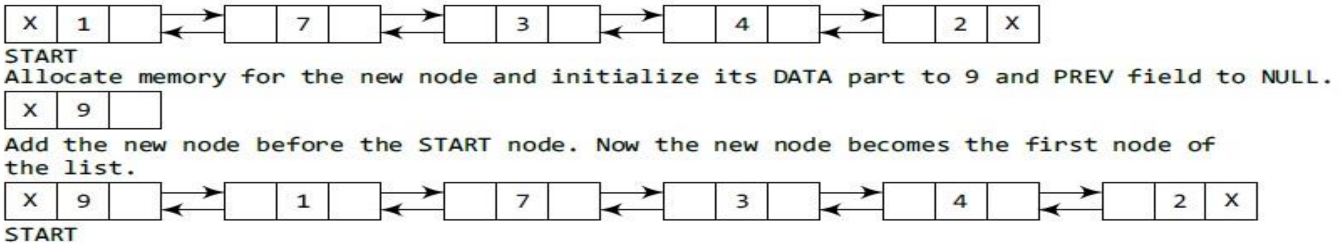
Algorithm to insert a new node in the beginning of the doubly linked list



```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Inserting a Node at the Beginning of a Doubly Linked List

Consider the doubly linked list, suppose we want to add a new node with data9 as the first node of the list. Then the following changes will be done in the linked list.



Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.

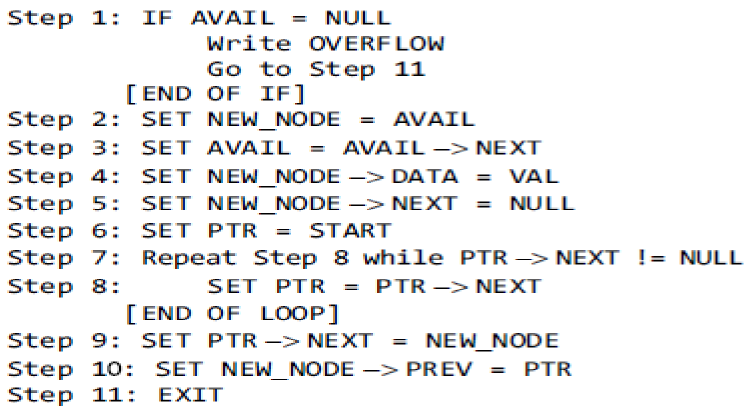
Add the new node before the START node. Now the new node becomes the first node of the list.

Inserting a new node at the beginning of a doubly linked list

Inserting a Node at the End of a Doubly Linked List

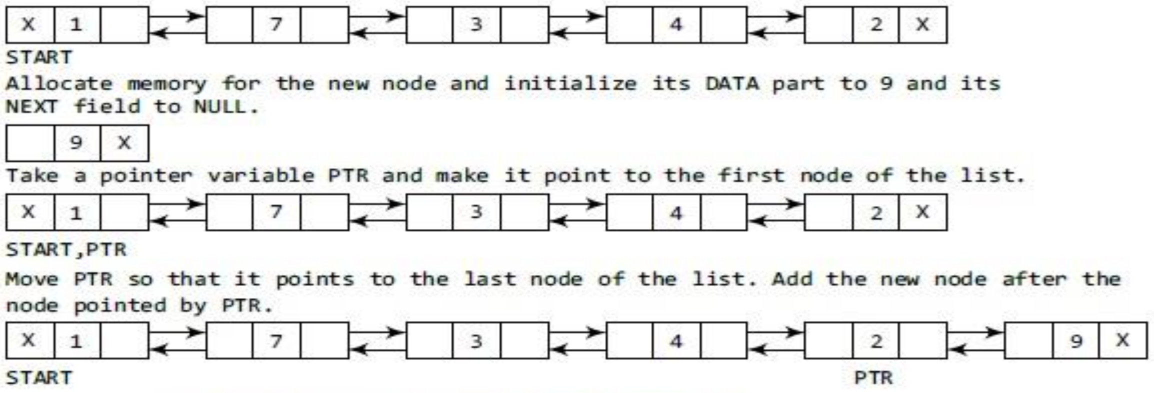
To insert a new node at the end of a doubly linked list, we take a pointer variable PTR and initialize it with START. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list. The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

Algorithm to insert a new node at the end of the doubly linked list



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Consider the doubly linked list and we want to add a new node with data at the last node of the list. Then



Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

Take a pointer variable PTR and make it point to the first node of the list.

Move PTR so that it points to the last node of the list. Add the new node after the node pointed by PTR.

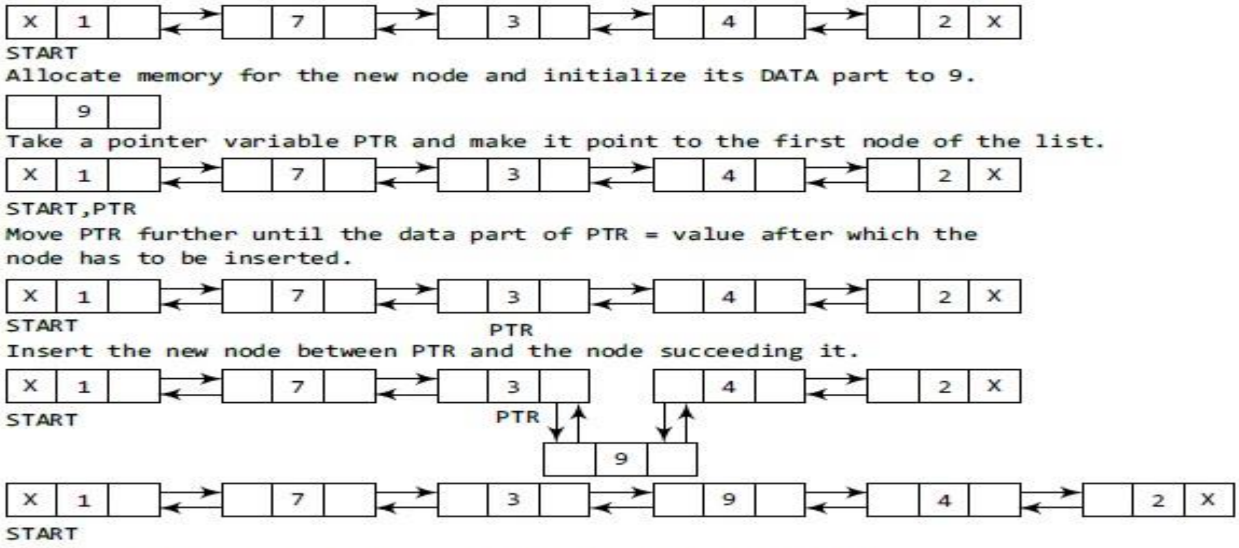
Inserting a new node at the end of a doubly linked list

Inserting a Node after a Given Node in a Doubly Linked List

To insert a new node after a given node in a doubly linked list, we take a pointer PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

Algorithm to insert a new node after a given node

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```



START

Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR and make it point to the first node of the list.

START,PTR

Move PTR further until the data part of PTR = value after which the node has to be inserted.

START

PTR

Insert the new node between PTR and the node succeeding it.

START

PTR

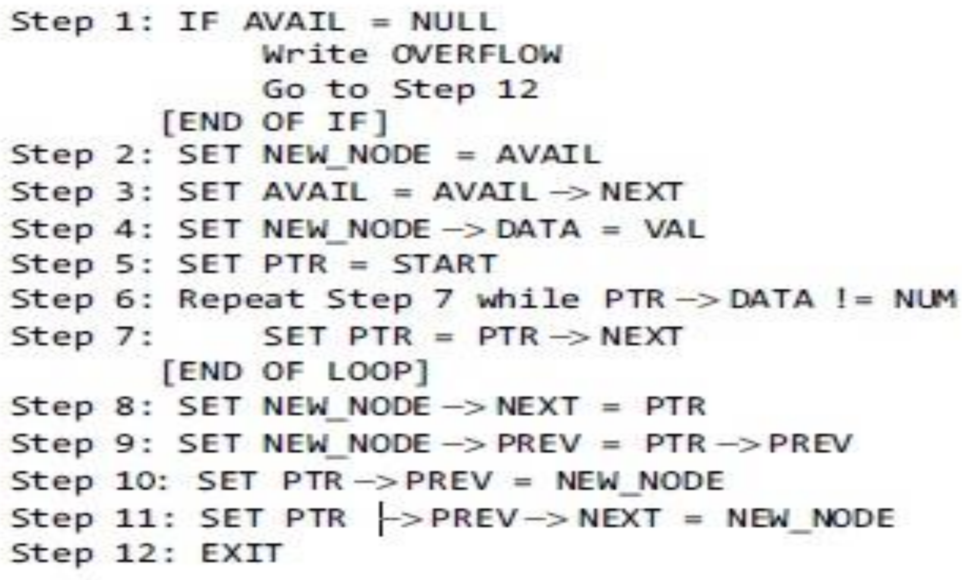
START

Inserting a new node after a given node in a doubly linked list

Inserting a Node before a Given Node in a Doubly Linked List

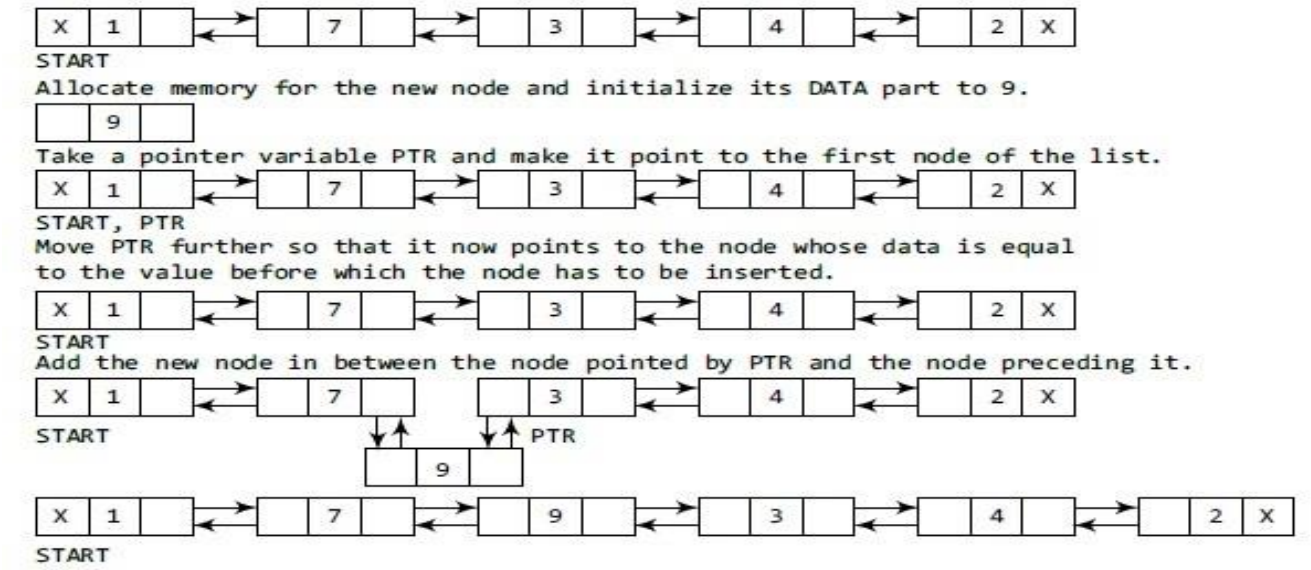
Consider the doubly linked list to add a new node with value before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm In Step 1, we first check whether memory is available for the new node. In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.

Algorithm to insert a new node before a node that has value NUM



```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR ├-> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

Algorithm to insert a new node before a given node



Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR and make it point to the first node of the list.

Move PTR further so that it now points to the node whose data is equal to the value before which the node has to be inserted.

Add the new node in between the node pointed by PTR and the node preceding it.

Inserting a new node before a given node in a doubly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
  int data;
  struct Node *next;
  struct Node *prev;
};
void insertStart (struct Node **head, int data)
{
  struct Node *newNode = (struct Node *) malloc (sizeof (struct Node));
  newNode->data = data;
  newNode->next = *head;
  newNode->prev = NULL;
  //If the linked list already had atleast 1 node
  if (*head != NULL) (*head)->prev = newNode;
  // *head->prev = newNode; would not work it has (*head) must be used
changing the new head to this freshly entered node
  *head = newNode;
}
// function to print the doubly linked list
void display (struct Node *node)
{
  struct Node *end;
  printf ("List in Forward direction: ");
  while (node != NULL)
    {
      printf (" %d ", node->data);
      end = node;
      node = node->next;
    }
  printf ("\nList in backward direction: ");
  while (end != NULL)
    {
```

Output

```
List in Forward
direction:  20  16
12
List in backward
direction:  12  16
20
```

```
     printf (" %d ", end->data);
     end = end->prev;
   }
}
int main ()
{
  struct Node *head = NULL;
  /*Need & i.e. address as we need to change head address only needs to
traverse and access items temporarily  */
  insertStart (&head, 12);
  insertStart (&head, 16);
  insertStart (&head, 20);
  /*No need for & i.e. address as we do not need to change head address */
  display (head);
  return 0;
}
```

Deleting a Node from a Doubly Linked List

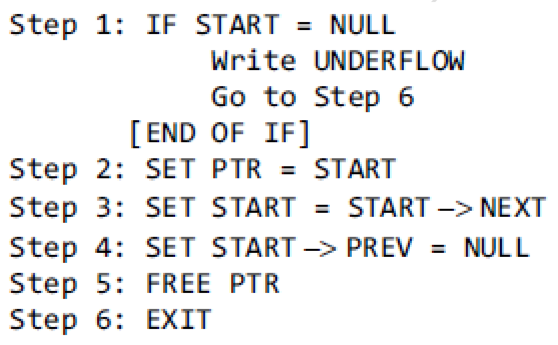
In this section, we will see how a node is deleted from an already existing doubly linked list. We Will take four cases and then see how deletion is done in each case.

Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.
Case 4: The node before a given node is deleted.

Deleting the First Node from a Doubly Linked List

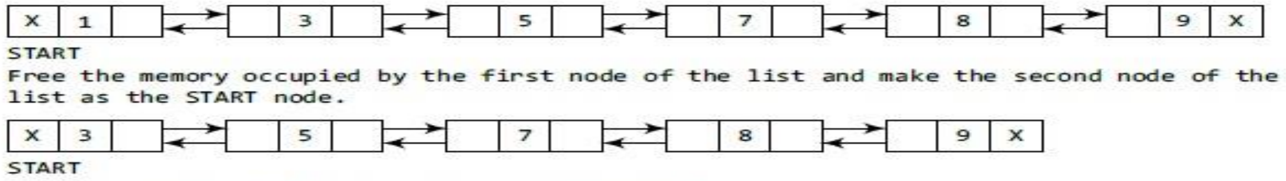
When we want to delete a node from thebeginning of the list, then the following changes will be done in the linked list.

Algorithm to delete the first node from the doubly linked list



```
Step 1: IF START = NULL
             Write UNDERFLOW
             Go to Step 6
       [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

To delete the first node of a doubly linked list, we check if the linked list exists or not. If START =NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.



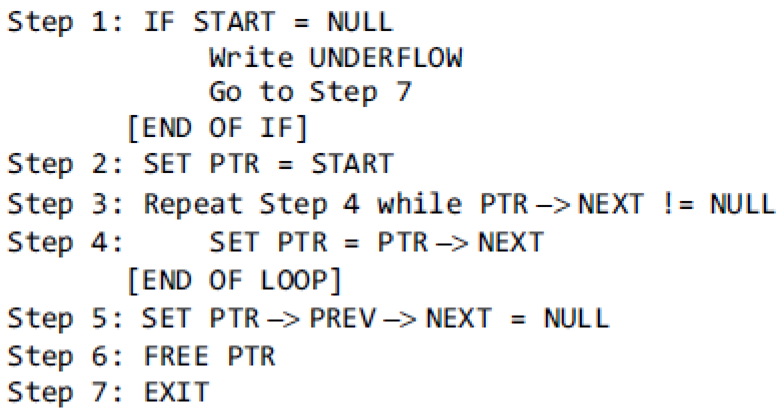
Free the memory occupied by the first node of the list and make the second node of the list as the START node.

Deleting the first node from a doubly linked list

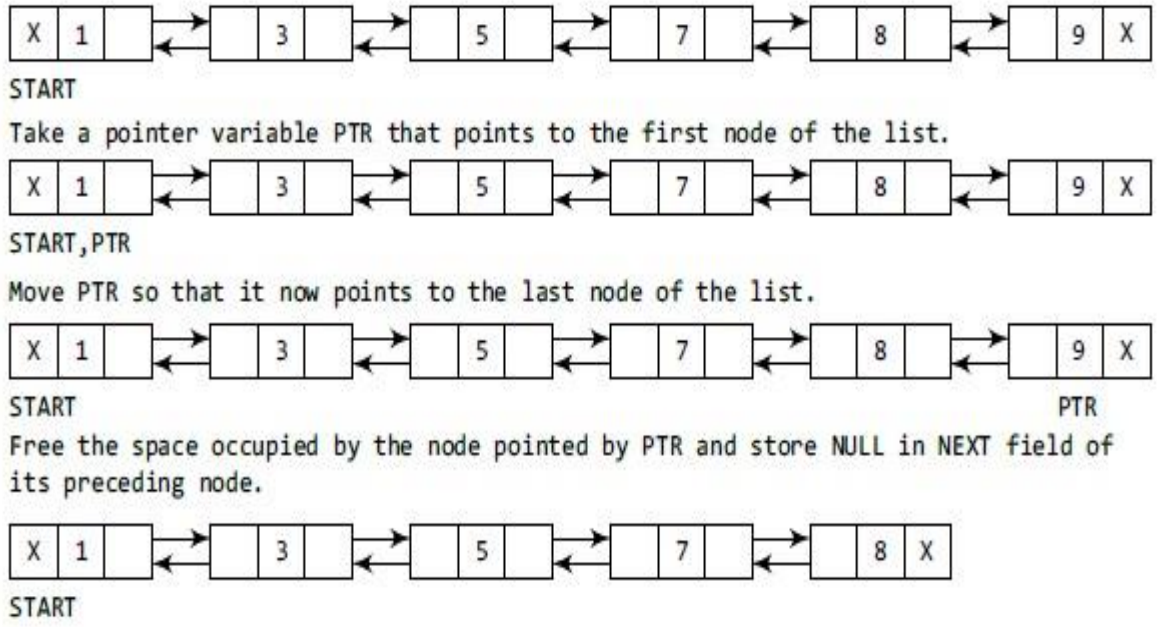
Deleting the Last Node from a Doubly Linked List

To delete the last node of a doubly linked list, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node, once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node. To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

Algorithm to delete the last node of the doubly linked list



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
       [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->NEXT != NULL
Step 4:     SET PTR = PTR->NEXT
       [END OF LOOP]
Step 5: SET PTR->PREV->NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```



Take a pointer variable PTR that points to the first node of the list.

Move PTR so that it now points to the last node of the list.

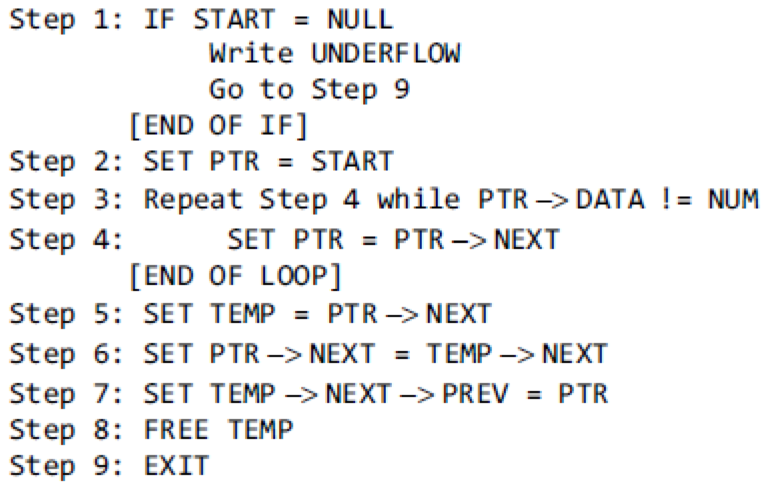
Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.

Deleting the last node from a doubly linked list

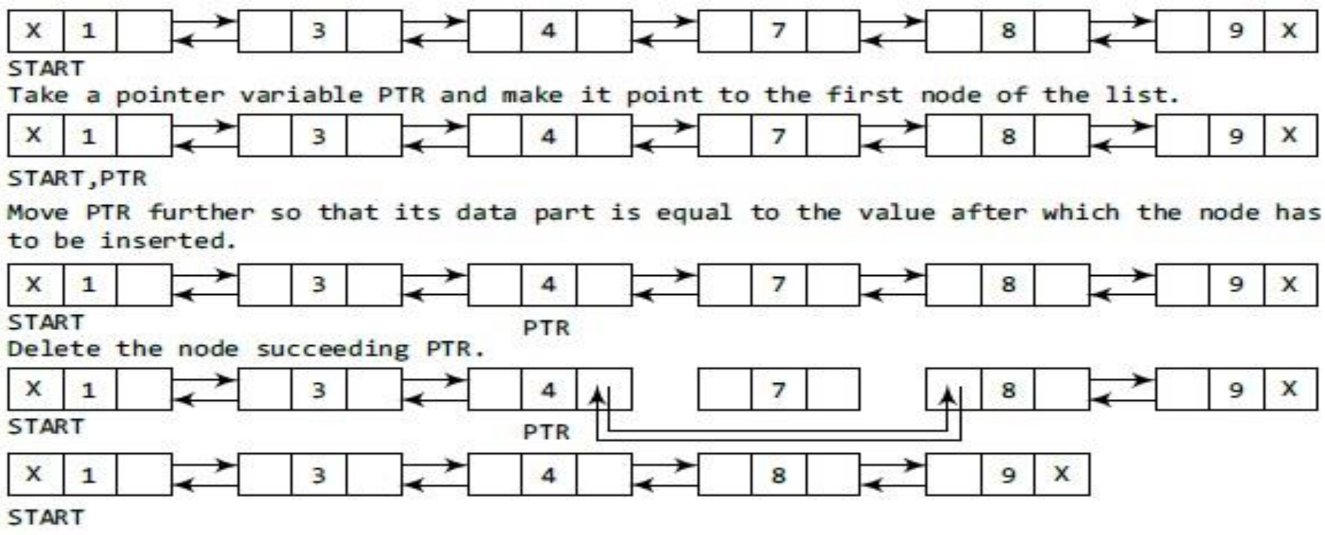
Deleting the Node after a Given Node in a Doubly Linked List

To delete a node after a given node of a doubly linked list, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node, Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

Algorithm to delete the node after a given node from the doubly linked list



```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 9
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->DATA != NUM
Step 4:       SET PTR = PTR->NEXT
          [END OF LOOP]
Step 5: SET TEMP = PTR->NEXT
Step 6: SET PTR->NEXT = TEMP->NEXT
Step 7: SET TEMP->NEXT->PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```



Take a pointer variable PTR and make it point to the first node of the list.

Move PTR further so that its data part is equal to the value after which the node has to be inserted.

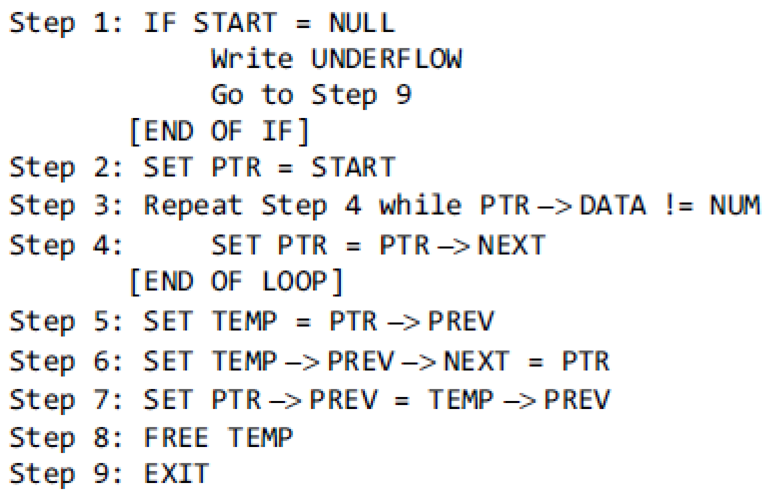
Delete the node succeeding PTR.

Deleting the node after a given node in a doubly linked list

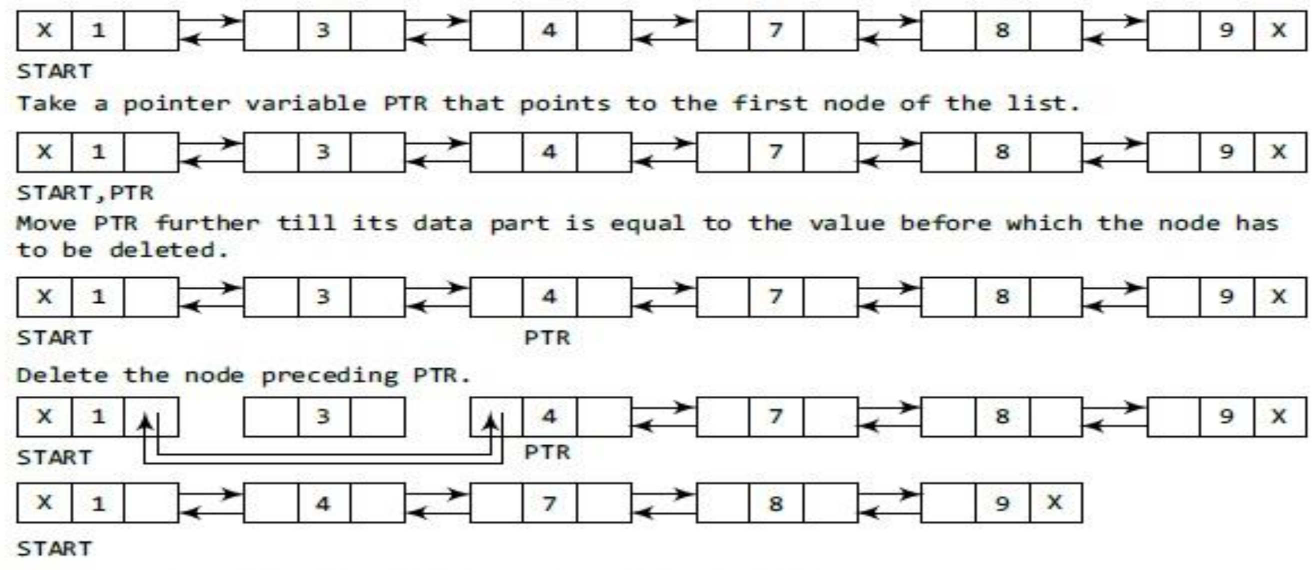
Deleting the Node before a Given Node in a Doubly Linked List

To delete a node before a given node of a doubly linked list, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the linked list to reach the desired node, Once we reach the node containing VAL, the PREV field of PTRis set to contain the address of the node preceding the node which comes before PTR. The memory of the node preceding PTR is freed and returned to the free pool.

Algorithm to delete the node before a given node from the doubly linked list



```
Step 1: IF START = NULL
             Write UNDERFLOW
             Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->DATA != NUM
Step 4:      SET PTR = PTR->NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR->PREV
Step 6: SET TEMP->PREV->NEXT = PTR
Step 7: SET PTR->PREV = TEMP->PREV
Step 8: FREE TEMP
Step 9: EXIT
```



Take a pointer variable PTR that points to the first node of the list.

Move PTR further till its data part is equal to the value before which the node has to be deleted.

Delete the node preceding PTR.

Deleting a node before a given node in a doubly linked list

Hence, we see that we can insert or delete a node in a constant number of operations given only that node’s address. Note that this is not possible in the case of a singly linked list which requires the previous node’s address also to perform the same operation

#include<stdio.h>	**OUTPUT**
#include<stdlib.h>	
struct Node	List in Forward direction:
{	12 11 10 9 8 7
int data;	List in backward direction:
struct Node *next;	7 8 9 10 11 12
struct Node *prev;	
};	12 deleted
int getLength (struct Node *node);	List in Forward direction:
void insert (struct Node **head, int data)	11 10 9 8 7
{	List in backward direction:
struct Node *freshNode = (struct Node *) malloc (sizeof (struct Node));	7 8 9 10 11
freshNode->data = data;	
freshNode->next = *head;	7 deleted

```
  freshNode->prev = NULL;
  // If the linked list already had atleast 1 node
  if (*head != NULL)
    (*head)->prev = freshNode;
  // freshNode will become head
  *head = freshNode;
}
void deleteFront (struct Node **head)
{
  struct Node *tempNode = *head;
  // if DLL is empty
  if (*head == NULL)
    {
      printf ("Linked List Empty, nothing to delete\n");
      return;
    }
  // if Linked List has only 1 node
  if (tempNode->next == NULL)
    {
      printf ("%d deleted\n", tempNode->data);
      *head = NULL;
      return;
    }
  // move head to next node
  *head = (*head)->next;
  // assign head node's previous to NULL
  (*head)->prev = NULL;
  printf ("%d deleted\n", tempNode->data);
  free (tempNode);
}
void deleteEnd (struct Node **head)
{
  struct Node *tempNode = *head;
  // if DLL is empty
  if (*head == NULL)
    {
      printf ("Linked List Empty, nothing to delete\n");
      return;
    }
  // if Linked List has only 1 node
  if (tempNode->next == NULL)
    {
      printf ("%d deleted\n", tempNode->data);
      *head = NULL;
      return;
    }
  // else traverse to the last node
  while (tempNode->next != NULL)
    tempNode = tempNode->next;
  struct Node *secondLast = tempNode->prev;
  // Curr assign 2nd last node's next to Null
  secondLast->next = NULL;
```

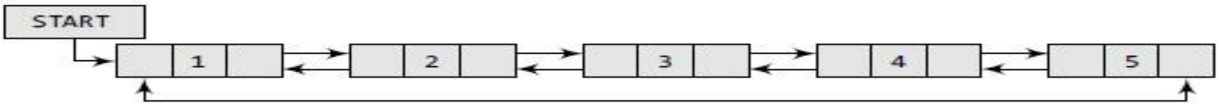
```
List in Forward direction:
11  10  9  8
List in backward direction:
8  9  10  11

9 deleted
List in Forward direction:
11  10  8
List in backward direction:
8  10  11

11 deleted
List in Forward direction:
10  8
List in backward direction:
8  10

8 deleted
List in Forward direction:
10
List in backward direction:
10
```

```
  printf ("%d deleted\n", tempNode->data);
  free (tempNode);
}
void deleteNthNode (struct Node **head, int n)
{
  //if the head node itself needs to be deleted
  int len = getLength (*head);
  // not valid
  if (n < 1 || n > len)
    {
      printf ("Enter valid position\n");
      return;
    }
  // delete the first node
  if (n == 1)
    {
      deleteFront (head);
      return;
    }
  if (n == len)
    {
      deleteEnd (head);
      return;
    }
  struct Node *tempNode = *head;
  // traverse to the nth node
  while (--n)
    {
      tempNode = tempNode->next;
    }
  struct Node *previousNode = tempNode->prev;      // (n-1)th node
  struct Node *nextNode = tempNode->next;// (n+1)th node
  // assigning (n-1)th node's next pointer to (n+1)th node
  previousNode->next = tempNode->next;
  // assigning (n+1)th node's previous pointer to (n-1)th node
  nextNode->prev = tempNode->prev;
  // deleting nth node
  printf ("%d deleted \n", tempNode->data);
  free (tempNode);
}
// required for deleteNthNode
int getLength (struct Node *node)
{
  int len = 0;
  while (node != NULL)
    {
      node = node->next;
      len++;
    }
  return len;
}
//function to print the doubly linked list
```

```
void display (struct Node *node)
{
  struct Node *end = NULL;
  printf ("List in Forward direction: ");
  while (node != NULL)
    {
      printf (" %d ", node->data);
      end = node;
      node = node->next;
    }
  printf ("\nList in backward direction:");
  while (end != NULL)
    {
      printf (" %d ", end->data);
      end = end->prev;
    }
  printf ("\n\n");
}
int main ()
{
  struct Node *head = NULL;
  insert (&head, 7);
  insert (&head, 8);
  insert (&head, 9);
  insert (&head, 10);
  insert (&head, 11);
  insert (&head, 12);
  display (head);
  deleteFront (&head);
  display (head);
  deleteEnd (&head);
  display (head);
  // delete 3rd node
  deleteNthNode (&head, 3);
  display (head);
  // delete 1st node
  deleteNthNode (&head, 1);
  display (head);
  // delete 1st node
  deleteEnd (&head);
  display (head);
  return 0;
}
```

CIRCULAR DOUBLY LINKED LISTS

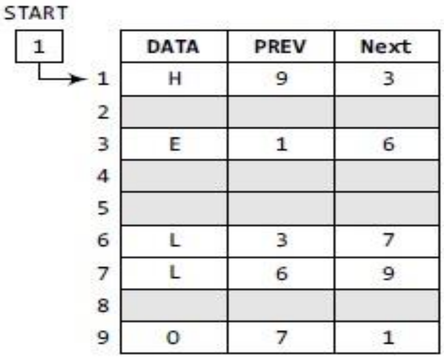
A circular doubly linked list or a circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. The difference between a doubly linked and a circular doubly linked list is same as that exists between a singly linked list and a circular linked list. The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node. Rather, the next field of the last node stores the address of the first node of the list, i.e., START. Similarly, the previous field of the first field stores the address of the last node.



Circular doubly linked list

Since a circular doubly linked list contains three parts in its structure, it calls for more space per node and more expensive basic operations. However, a circular doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a circular doubly linked list is that it makes search operation twice as efficient.

Let us view how a circular doubly linked list is maintained in the memory.



	DATA	PREV	Next
1	H	9	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	1

Memory representation of a circular doubly linked list

We see that a variable START is used to store the address of the first node. Here in this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it stores the address of the last node of the list in its previous field. The corresponding NEXT stores the address of the next node, which is 3. So, we will look at address3 to fetch the next data item. The previous field will contain the address of the first node. The second data element obtained from address 3 is E. We repeat this procedure until we reach a position where the NEXT entry stores the address of the first element of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.

Inserting a New Node in a Circular Doubly Linked List

We will see how a new node is added into an already existing circular doubly linked list. We will take two cases and then see how insertion is done in each case. Rests of the cases are similar to that given for doubly linked lists.

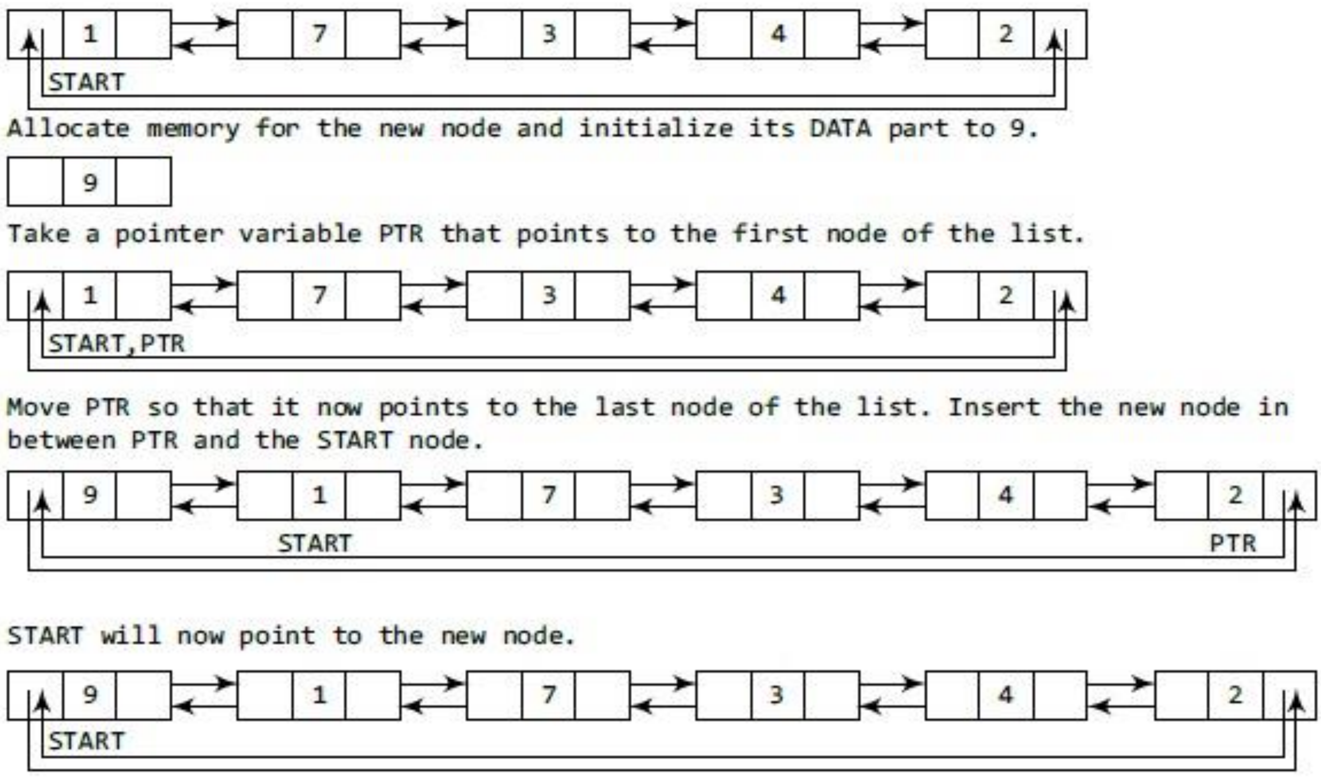
Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.

Inserting a Node at the Beginning of a Circular Doubly Linked List

To insert anew node at the beginning of a circular doubly linked list, we first check whether Memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, we allocate space for the new node. Set its data part with the given VAL and its next part is initialized with the address of the first node of the list, which is stored in START. Now since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE. Since it is a circular doubly linked list, the PREV field of the NEW_ NODE is set to contain the address of the last node.

Algorithm to insert a new node in the beginning of the circular doubly linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 13
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 8: SET PTR -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET NEW_NODE -> NEXT = START
Step 11: SET START -> PREV = NEW_NODE
Step 12: SET START = NEW_NODE
Step 13: EXIT
```



Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR that points to the first node of the list.

Move PTR so that it now points to the last node of the list. Insert the new node in between PTR and the START node.

START will now point to the new node.

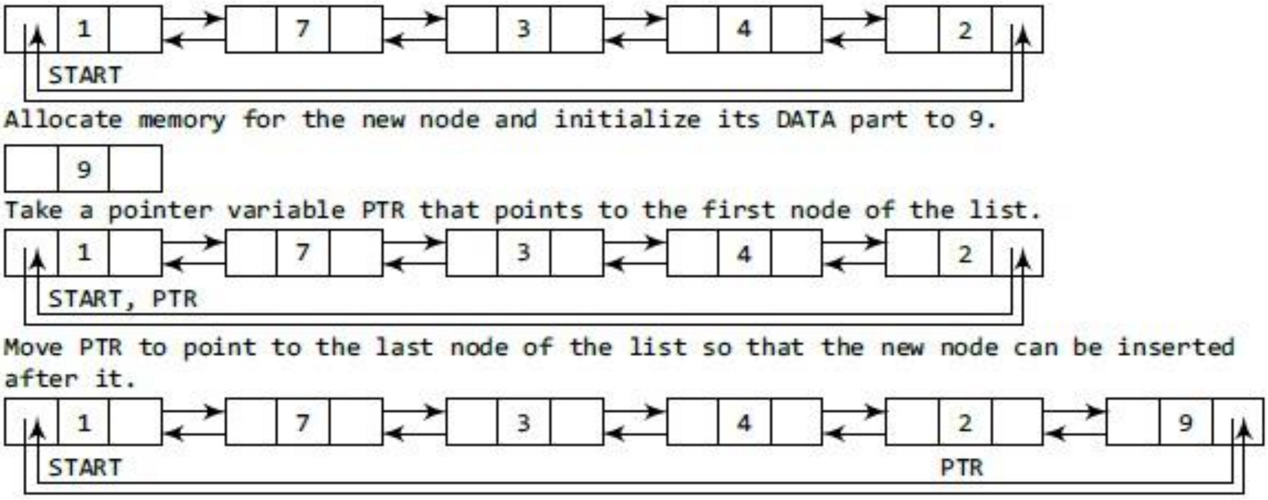
Inserting a new node at the beginning of a circular doubly linked list

Inserting a Node at the End of a Circular Doubly Linked List

To insert a new node at the end of a circular doubly linked list, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

Algorithm to insert a new node at the end of the circular doubly linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: SET START -> PREV = NEW_NODE
Step 12: EXIT
```



Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR that points to the first node of the list.

Move PTR to point to the last node of the list so that the new node can be inserted after it.

Inserting a new node at the end of a circular doubly linked list

#include<stdio.h>
#include<stdlib.h>
struct Node
{
int data;
struct Node *next;
};
void insertStart (struct Node **head, int data)
{
struct Node *newNode = (struct Node *) malloc (sizeof (struct Node));
newNode->data = data;
	Output
	Insert at beginning: 1 2
	Insert at End: 1 2 30 40
	Insert at Specific Position: 1 2 5 30 40

```
  // if its the first node being entered
  if (*head == NULL)
    {
      *head = newNode;
      (*head)->next = *head;
      return;
    }
  // if LL already as >=1 node
  struct Node *curr = *head;
  // traverse till last node in LL
  while (curr->next != *head)
    {
      curr = curr->next;
    }
  // assign LL's last node's next as this new node
  curr->next = newNode;
  // assign newNode's next as current head
  newNode->next = *head;
  // change head to this new node
  *head = newNode;
}
void insertLast (struct Node **head, int data)
{
  struct Node *newNode = (struct Node *) malloc (sizeof (struct Node));
  newNode->data = data;
  // if its the first node being entered
  if (*head == NULL)
    {
      *head = newNode;
      (*head)->next = *head;
      return;
    }
  // if LL already as >=1 node
  struct Node *curr = *head;
  // traverse till last node in LL
  while (curr->next != *head)
    {
      curr = curr->next;
    }
  // assign LL's current last node's next as this new node
  curr->next = newNode;
  // assign this new node's next as current head of LL
  newNode->next = *head;
}
void insertPosition (int data, int pos, struct Node **head)
//function to insert element at specific position
{
  struct Node *newnode, *curNode;
  int i;
  if (*head == NULL)
    {
      printf ("List is empty");
    }
  if (pos == 1)
    {
      insertStart (head, data);
      return;
    }
  else
    {
      newnode = (struct Node *) malloc (sizeof (struct Node));
      newnode->data = data;
      curNode = *head;
      while (--pos > 1)
            {
              curNode = curNode->next;
            }
      newnode->next = curNode->next;
      curNode->next = newnode;
    }
}
void display (struct Node *head)
{
  // if there are no node in LL
  if (head == NULL)
    return;
  struct Node *temp = head;
  //need to take care of circular structure of LL
  do
    {
      printf ("%d ", temp->data);
      temp = temp->next;
    }
  while (temp != head);
  printf ("\n");
}
int main ()
{
  struct Node *head = NULL;
  printf("Insert at beginning: ");
  insertStart (&head, 2);
  insertStart (&head, 1);
  display (head);
  printf("Insert at End: ");
  insertLast (&head, 30);
  insertLast (&head, 40);
  display (head);
  printf("Insert at Specific Position: ");
  insertPosition (5, 3, &head);
  display (head);
  return 0;
}
```

Deleting a Node from a Circular Doubly Linked List

We will see how a node is deleted from an already existing circular doubly linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases is same as that given for doubly linked lists.

Case 1: The first node is deleted.

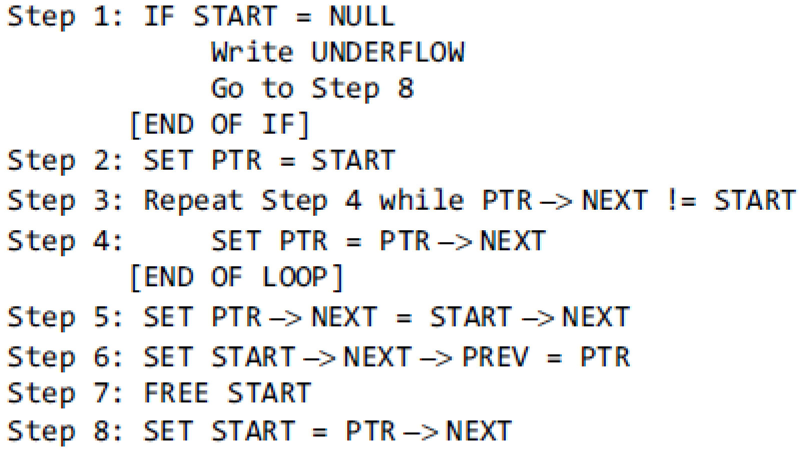
Case 2: The last node is deleted.

Deleting the First Node from a Circular Doubly Linked List

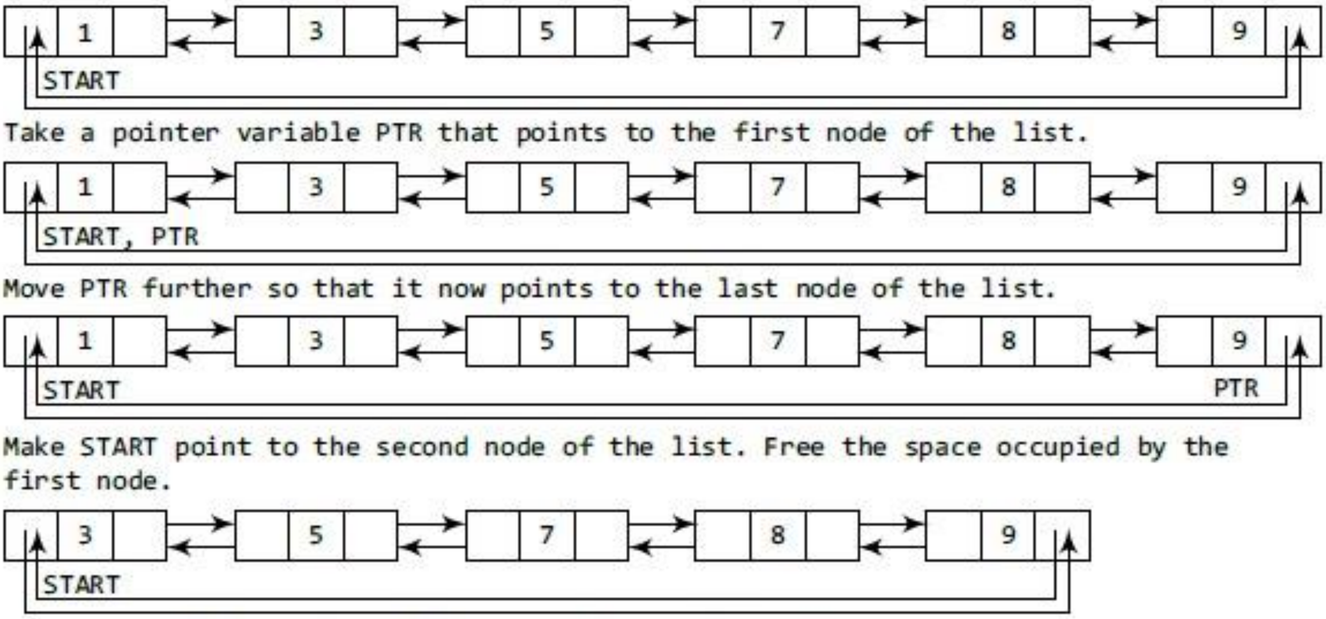
When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list, to delete the first node from a circular doubly linked list. we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.

However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. The while loop traverses through the list to reach the last node. Once we reach the last node, the NEXT pointer of PTR is set to contain the address of the node that succeeds START. Finally, START is made to point to the next node in the sequence and the memory occupied by the first node of the list is freed and returned to the free pool.

Algorithm to delete the first node from the circular doubly linked list



```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: SET START -> NEXT -> PREV = PTR
Step 7: FREE START
Step 8: SET START = PTR -> NEXT
```



Take a pointer variable PTR that points to the first node of the list.

Move PTR further so that it now points to the last node of the list.

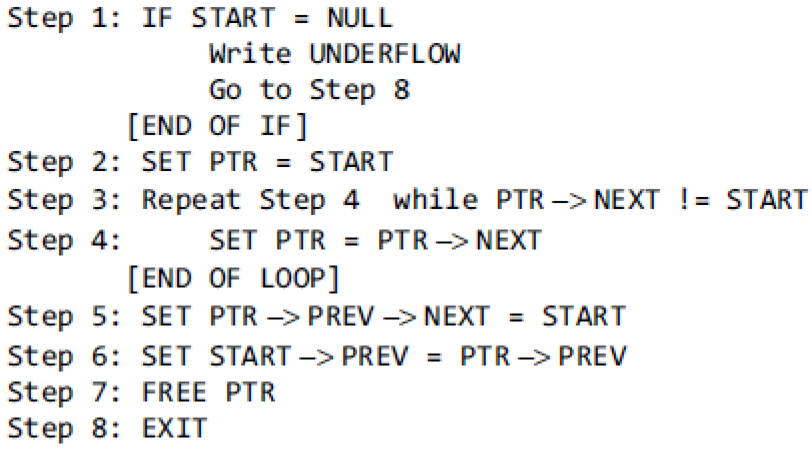
Make START point to the second node of the list. Free the space occupied by the first node.

Deleting the first node from a circular doubly linked list

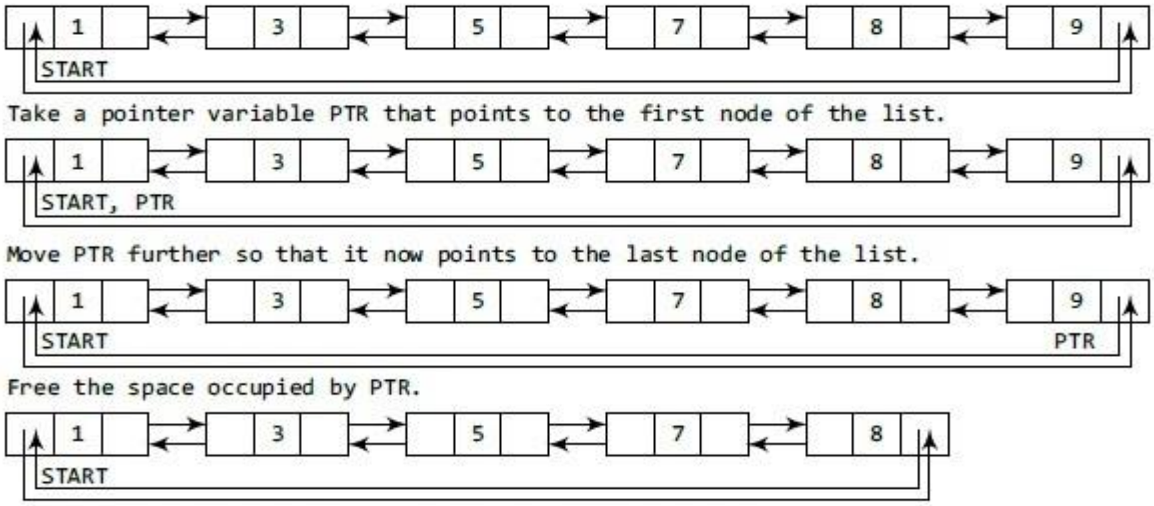
Deleting the Last Node from a Circular Doubly Linked List

To delete the last node from a circular doubly linked list, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop Traverses through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node. To delete the last node, we simply have to set the next field of the second last node to contain the address of START, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

Algorithm to delete the last node of the circular doubly linked list



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
       [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4  while PTR -> NEXT != START
Step 4:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = START
Step 6: SET START -> PREV = PTR -> PREV
Step 7: FREE PTR
Step 8: EXIT
```



Take a pointer variable PTR that points to the first node of the list.

Move PTR further so that it now points to the last node of the list.

Free the space occupied by PTR.

Deleting the last node from a circular doubly linked list

```
#include<stdio.h>
#include<stdlib.h>
// structure for Circular Linked List
struct Node
{
  int data;
  struct Node *next;
};
int calcSize (struct Node *head);
void deleteBegin(struct Node **head)
{
  struct Node *tempNode = *head;
  // if there are no nodes in Linked List can't delete
```

Output

```
16 15 14 13 12 11 10
15 14 13 12 11 10
15 14 13 12 11
15 14 12 11
```

```
  if (*head == NULL)
    {
      printf ("Linked List Empty, nothing to delete");
      return;
    }
  // if only 1 node in CLL
  if (tempNode->next == *head)
    {
      *head = NULL;
      return;
    }
  struct Node *curr = *head;
  // traverse till last node in CLL
  while (curr->next != *head)
    curr = curr->next;
  // assign last node's next to 2nd node in CLL
  curr->next = (*head)->next;
  // move head to next node
  *head = (*head)->next;
  free (tempNode);
}
void deleteEnd (struct Node **head)
{
  struct Node *tempNode = *head;
  struct Node *previous;
  // if there are no nodes in Linked List can't delete
  if (*head == NULL)
    {
      printf ("Linked List Empty, nothing to delete");
      return;
    }
  // if Linked List has only 1 node
  if (tempNode->next == *head)
    {
      *head = NULL;
      return;
    }
  // else traverse to the last node
  while (tempNode->next != *head)
    {
      // store previous link node as we need to change its next val
      previous = tempNode;
      tempNode = tempNode->next;
    }
  // Curr assign 2nd last node's next to head
  previous->next = *head;
  // delete the last node
  free (tempNode);
  // 2nd last now becomes the last node
}
void deletePos (struct Node **head, int n)
{
  int size = calcSize (*head);
  if (n < 1 || size < n)
    {
      printf ("Can't delete, %d is not a valid position\n", n);
    }
  else if (n == 1)
    deleteBegin(head);
  else if (n == size)
    deleteEnd (head);
  else
    {
      struct Node *tempNode = *head;
      struct Node *previous;
      // traverse to the nth node
      while (--n)
            {
 // store previous link node as we need to change its next val
              previous = tempNode;
              tempNode = tempNode->next;
            }
      // change previous node's next node to nth node's next node
      previous->next = tempNode->next;
      // delete this nth node
      free (tempNode);
    }
}
void insert (struct Node **head, int data)
{
 struct Node *newNode = (struct Node *) malloc (sizeof (struct
Node));
  newNode->data = data;
  if (*head == NULL)
    {
      *head = newNode;
      (*head)->next = *head;
      return;
    }
  struct Node *curr = *head;
  while (curr->next != *head)
    {
      curr = curr->next;
    }
  curr->next = newNode;
  newNode->next = *head;
  *head = newNode;
}
int calcSize (struct Node *head)
{
  int size = 0;
  struct Node *temp = head;

  if (temp == NULL)
```

```
    return size;
  do
    {
      size++;
      temp = temp->next;
    }
  while (temp != head);
  return size;
}
void display (struct Node *head)
{
  // if there are no node in CLL
  if (head == NULL)
    return;
  struct Node *temp = head;
  //need to take care of circular structure of CLL
  do
    {
      printf ("%d ", temp->data);
      temp = temp->next;
    }
  while (temp != head);
  printf ("\n");
}
int main ()
{
  // first node will be null at creation
  struct Node *head = NULL;
  insert (&head, 10);
  insert (&head, 11);
  insert (&head, 12);
  insert (&head, 13);
  insert (&head, 14);
  insert (&head, 15);
  insert (&head, 16);
  display (head);

  deleteBegin(&head);
  display (head);

  deleteEnd (&head);
  display (head);

  deletePos (&head, 3);
  display (head);

  return 0;
}
```

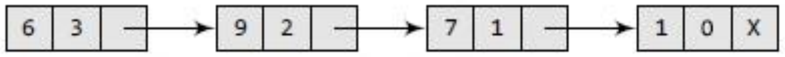
APPLICATIONS OF LINKED LISTS

Linked lists can be used to represent polynomials and the different operations that can be performed on them. We will see how polynomials are represented in the memory using linked lists.

Polynomial Representation

Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial 6x3 + 9x2 + 7x + 1. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.

Every term of a polynomial can be represented as a node of the linked list. Figure shows the linked representation of the terms of the above polynomial.



Linked representation of a polynomial

Linked representation of a polynomial Now that we know how polynomials are represented using nodes of a linked list, let us write a program to perform operations on polynomials.

****

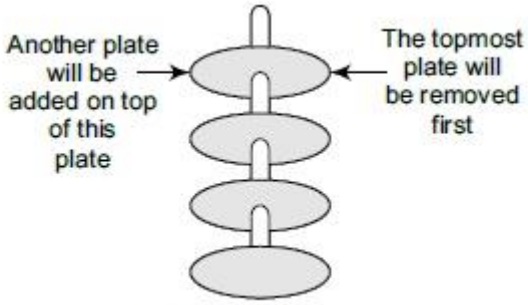
UNIT-III

Stacks: Introduction to stacks: properties and operations, implementing stacks using arrays and linked lists, Applications of stacks in expression evaluation, backtracking, reversing list etc.

INTRODUCTION

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.


Another plate will be added on top of this plate
The topmost plate will be removed first

Stack of plates

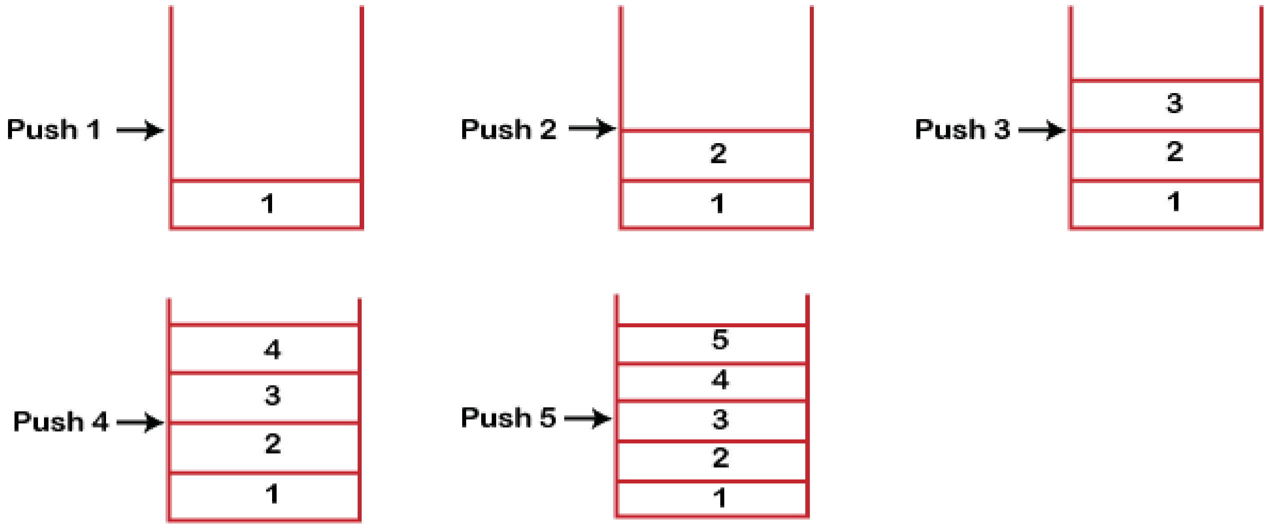
A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** or **FILO (First -In- Last -Out)** principle. Stack has one end, It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top 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

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.

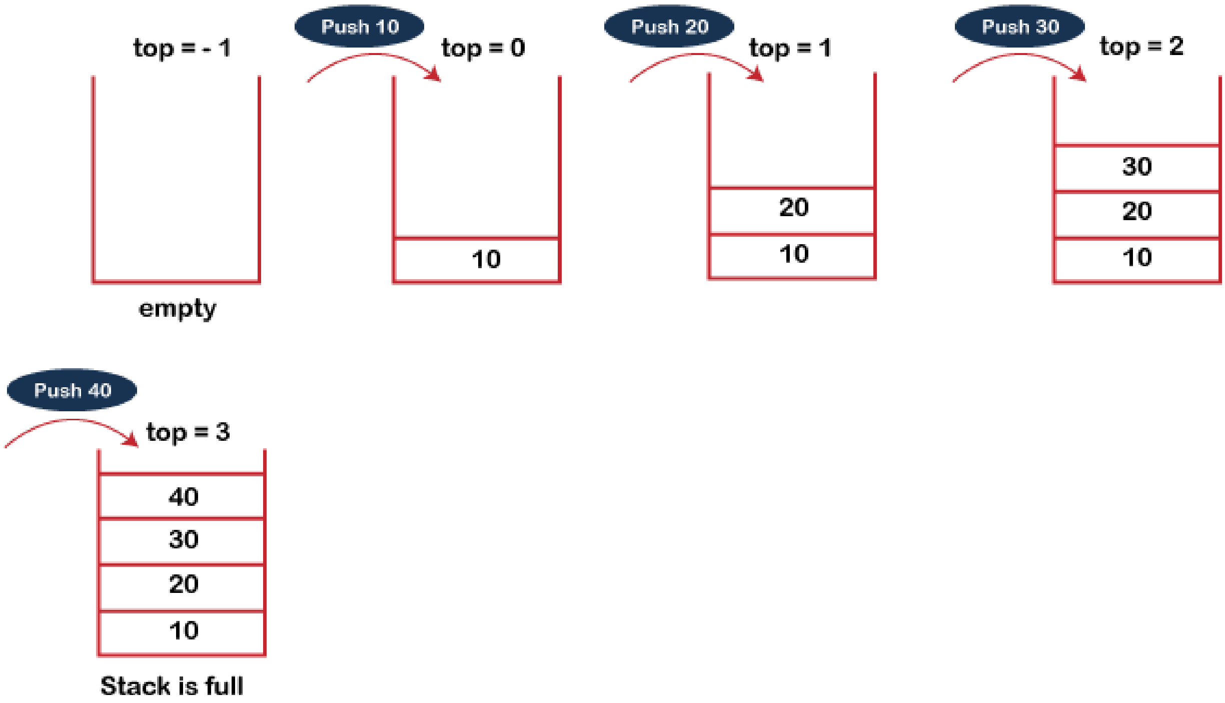
Display (): It prints all the elements available in the stack.

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.

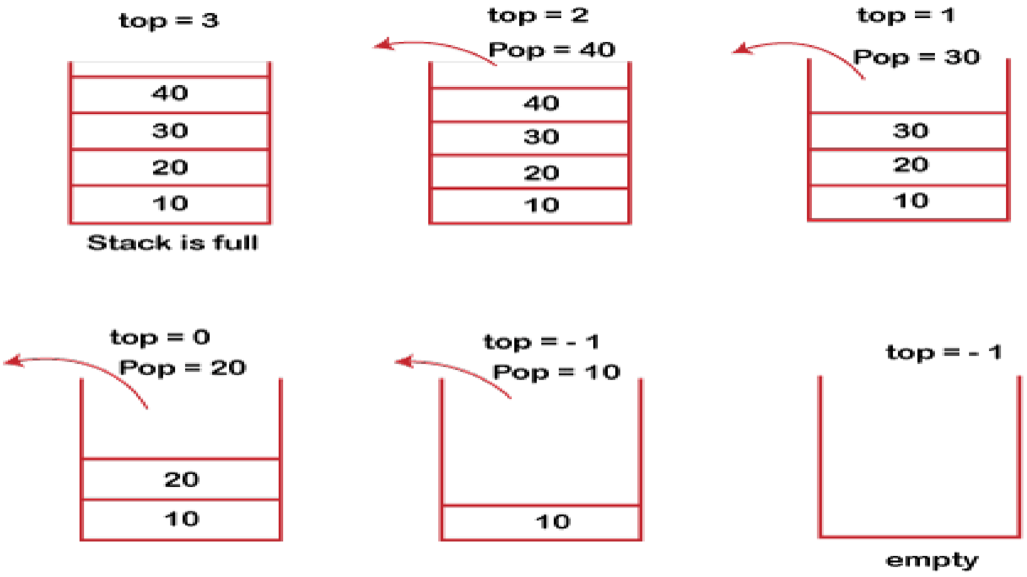
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.



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**.

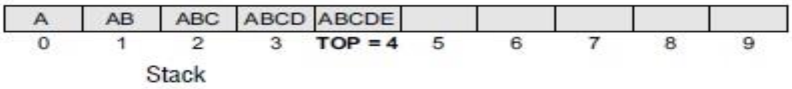


Stack Can be Representation in two ways

- Array Representation
- Linked List Representation

ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.(You must be wondering why we have written MAX–1. It is because array indices start from 0.)



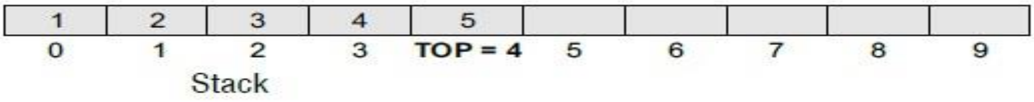
A	AB	ABC	ABCD	ABCDE					
0	1	2	3	TOP = 4	5	6	7	8	9

Stack

The stack in Fig. shows that TOP = 4, so insertions and deletions will be done at this position.In the above stack, five more elements can still be stored.

Push Operation

The push operation is used to insert an element into the stack. The new element is added at thetopmost position of the stack. However, before inserting the value, we must first check if TOP=MAX–1,because if that is the case, then the stack is full and no more insertions can be done. If an attemptis made to insert a value in a stack that is already full, an OVERFLOW message is printed.



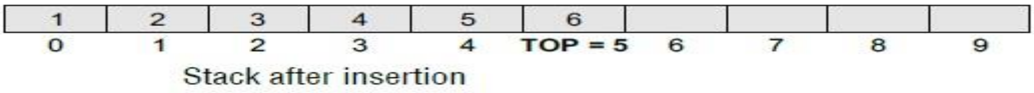
1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

Stack

To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack [TOP].

Algorithm to PUSH an element in a stack

```
Step 1: If Top = Max-1, Then
            Print "Overflow"
          Goto Step 4
      [End Of If]
Step 2: Set Top = Top + 1
Step 3: Set Stack [Top] = Value
Step 4: End
```



1	2	3	4	5	6				
0	1	2	3	4	TOP = 5	6	7	8	9

Stack after insertion

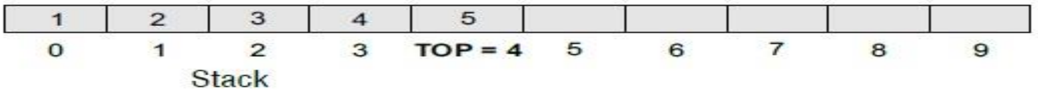
To insert an element in a stack, we first check for the OVERFLOW condition, then TOPis incremented so that it points to the next location in the array, the value is stored in the stack at the location pointed by TOP.

Pop Operation

The pop operation is used to delete the topmost element from thestack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is emptyand no more deletions can be done. If an attempt is made to delete a value from a stack that isalready empty, an UNDERFLOW message is printed.

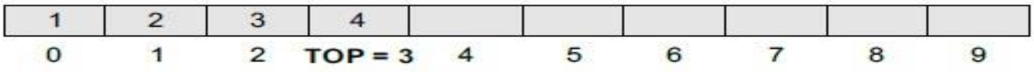
Algorithm to POP an element from a stack

```
Step 1: If Top = Null, Then
             Print "Underflow"
            Goto Step 4
        [End Of If]
Step 2: Set Val = Stack[Top]
Step 3: Set Top = Top - 1
Step 4: End
```



Stack

To delete the topmost element, we first check if TOP=NULL. If the condition is false, then wedecrement the value pointed by TOP.



Stack after deletion

To delete an element from a stack, we first check for the UNDERFLOW condition, the value of the location in the stack pointed by TOP is stored in VAL, TOP is decremented.

```
#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)
```

Output

```
Enter the number of elements in the
stack 6
*********Stack operations using
array*********
----------------------------------------------
Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
```

```
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
        };
    }
}

void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
       printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
       printf("Stack is empty");
    }
}
```

Peek/Peep Operation

Peek is an operation that returns the value of the top most element of the stack without deleting it from the stack. However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

Algorithm for Peek/Peep Operation

```
      Step 1: If Top =Null, Then
                 Print "Stack Is Empty"
                 Go To Step 3
   [End of If]
      Step 2: Return Stack [Top]
      Step 3: End
```

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

Stack

Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

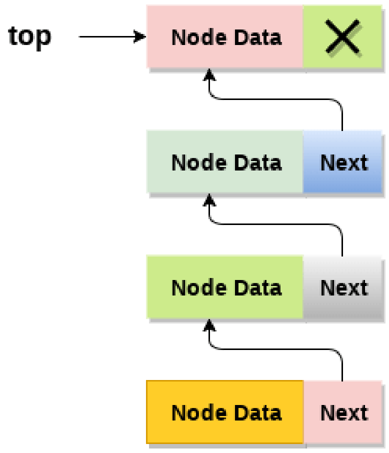
LINKED REPRESENTATION OF STACKs

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is **O(1).**In a linked stack, every node has two parts one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

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.



Stack

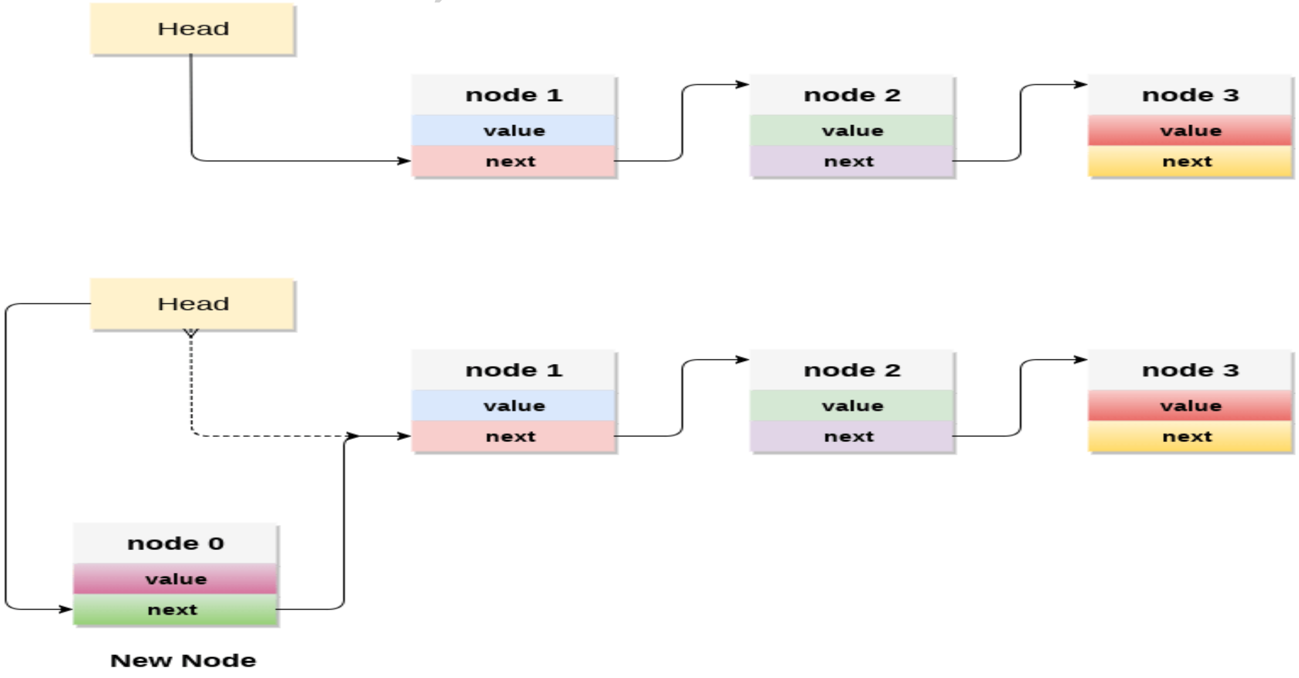
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

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.

- Create a node first and allocate memory to it.
- 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.
- 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)

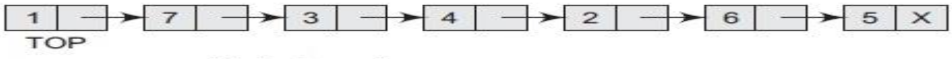


Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.

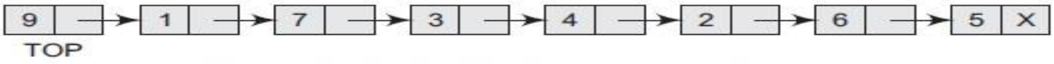
Algorithm to PUSH an element in a linked stack

```
Step 1: allocate memory for the new node and name it as new_node
Step 2: set new_node->data = val
Step 3: if top = null, then
                set new_node->next = null
                set top = new_node
    else
                set new_node->next = top
                set top = new_node
    [end of if]
Step 4: end
```



Linked stack

To insert an element with value, we first check if TOP=NULL, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP! =NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP.



Linked stack after inserting a new node

To push an element into a linked stack, memory is allocated for the new node, the DATA part of the new node is initialized with the value to be stored in the node, and we check if the new node is the first node of the linked list. This is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

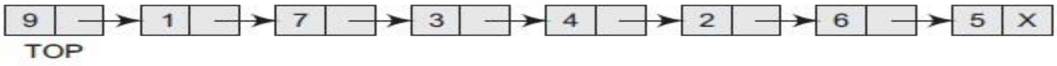
Deleting a node from the top of stack is referred to as pop operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation.

- **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.
- **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)

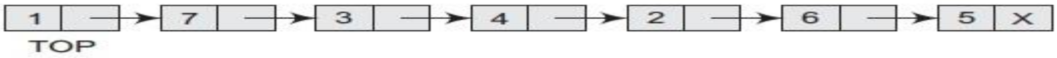
Algorithm to POP an element from a stack

```
Step 1: If Top = Null, Then
                Print "Underflow"
Goto Step 5
    [End Of If]
Step 2: Set Ptr = Top
Step 3: Set Top = Top ->Next
Step 4: Free Ptr
Step 5: End
```



Linked stack

In case TOP! =NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack.



Linked stack after deletion of the topmost element

To delete an element from a stack, we first check for the UNDERFLOW condition, we use a pointer PTR that points to TOP, TOP is made to point to the next node in sequence, the memory occupied by PTR is given back to the free pool.

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.

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

Time Complexity: o(n)

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

OUTPUT:

```
Linked List
30->20->10->NULL
Poped element = 30
After the pop, the new
linked list
20->10->NULL
```

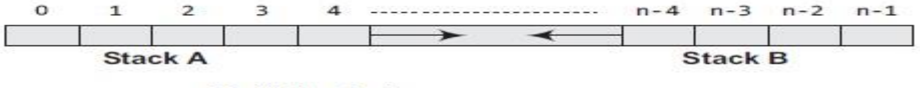
```
struct Node *head = NULL;
void push(int val)
{
   //create new node
   struct Node *newNode = malloc(sizeof(struct Node));
   newNode->data = val;
   //make the new node points to the head node
   newNode->next = head;
//make the new node as head node so that head will always point the
last inserted data
   head = newNode;
}
void pop()
{
   //temp is used to free the head node
   struct Node *temp;
   if(head == NULL)
      printf("Stack is Empty\n");
   else
   {
      printf("Poped element = %d\n", head->data);
   //backup the head node
      temp = head;
 //make the head node points to the next node.logically removing the
node
      head = head->next;
      //free the poped element's memory
      free(temp);
   }
}
//print the linked list
void display()
{
   struct Node *temp = head;
   //iterate the entire linked list and print the data
```

```
Poped element = 20
After the pop, the new
linked list
10->NULL
```

```
    while(temp != NULL)
    {
        printf("%d->", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main()
{
    push(10);
    push(20);
    push(30);
    printf("Linked List\n");
    display();
    pop();
    printf("After the pop, the new linked list\n");
    display();
    pop();
    printf("After the pop, the new linked list\n");
    display();
    return 0;
}
```

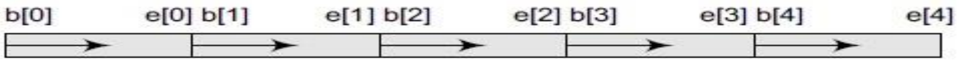
MULTIPLE STACKS

While implementing a stack using an array, we had seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent OVERFLOW conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array. In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory. Thus, there lies a trade-off between the frequency of overflows and the space allocated. So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size.



Multiple stacks

An array STACK[n] is used to represent two stacks, Stack A and Stack B. The value of n is such that the combined size of both the stacks will never exceed n. While operating on these stacks, it is important to note one thing Stack A will grow from left to right, whereas Stack B will grow from right to left at the same time. Extending this concept to multiple stacks, a stack can also be used to represent number of stacks in the same array. That is, if we have a STACK[n], then each stack I will be allocated an equal amount of space bounded by indices b[i] and e[i].



Multiple stacks

Implement two stacks in an array

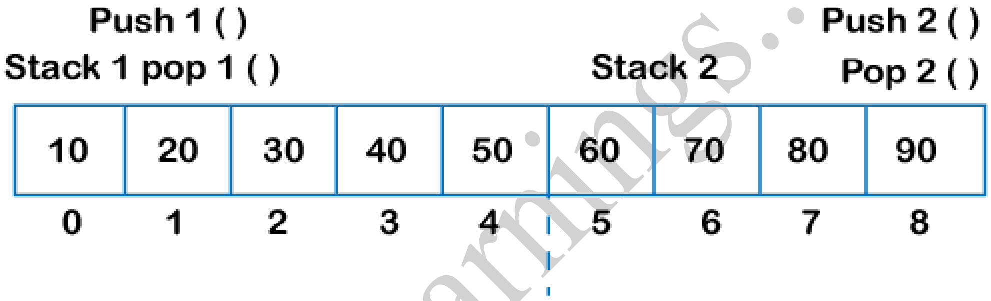
Here, we will create two stacks, and we will implement these two stacks using only one array, i.e., both the stacks would be using the same array for storing elements.

There are two approaches to implement two stacks using one array:

First Approach

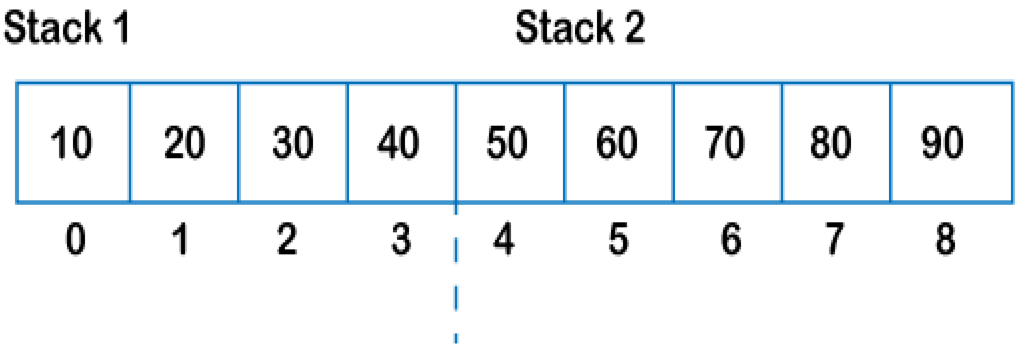
First, we will divide the array into two sub-arrays. The array will be divided into two equal parts. First, the sub-array would be considered stack1 and another sub array would be considered stack2.

For example, if we have an array of n equal to 8 elements. The array would be divided into two equal parts, i.e., 4 size each shown as below:



The first subarray would be stack 1 named as st1, and the second subarray would be stack 2 named as st2. On st1, we would perform push1() and pop1() operations, while in st2, we would perform push2() and pop2() operations. The stack1 would be from 0 to n/2, and stack2 would be from n/2 to n-1.

If the size of the array is odd. For example, the size of an array is 9 then the left subarray would be of 4 size, and the right subarray would be of 5 size shown as below:



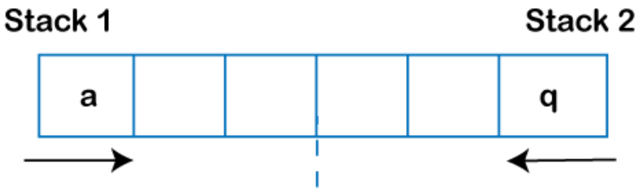
Disadvantage of using this approach

Stack overflow condition occurs even if there is a space in the array. In the above example, if we are performing push1() operation on the stack1. Once the element is inserted at the 3rd index and if we try to insert more elements, then it leads to the overflow error even there is a space left in the array.

Second Approach

In this approach, we are having a single array named as 'a'. In this case, stack1 starts from 0 while stack2 starts from n-1. Both the stacks start from the extreme corners, i.e., Stack1 starts from the leftmost corner (at index 0), and Stack2 starts from the rightmost corner (at index n-1). Stack1 extends in the right direction, and stack2 extends in the left direction,

If we push 'a' into stack1 and 'q' into stack2



Therefore, we can say that this approach overcomes the problem of the first approach. In this case, the stack overflow condition occurs only when **top1 + 1 = top2.** This approach provides a space-efficient implementation means that when the array is full, then only it will show the overflow error. In contrast, the first approach shows the overflow error even if the array is not full.

SOURCE CODE

```
#include <stdio.h>
#define SIZE 20
 int array[SIZE];  // declaration of array type variable.
int top1 = -1;
int top2 = SIZE;
//Function to push data into stack1
void push1 (int data)
{
// checking the overflow condition
  if (top1 < top2 - 1)
  {
      top1++;
    array[top1] = data;
  }
  else
  {
    printf ("Stack is full");
  }
}
// Function to push data into stack2
void push2 (int data)
{
// checking overflow condition
if (top1 < top2 - 1)
  {
    top2--;
    array[top2] = data;
  }
  else
  {
    printf ("Stack is full..\n");
  }
}
```

OUT PUT

```
We can push a total of 20 values
Value Pushed in Stack 1 is 1
Value Pushed in Stack 1 is 2
Value Pushed in Stack 1 is 3
Value Pushed in Stack 1 is 4
Value Pushed in Stack 1 is 5
Value Pushed in Stack 1 is 6
Value Pushed in Stack 1 is 7
Value Pushed in Stack 1 is 8
Value Pushed in Stack 1 is 9
Value Pushed in Stack 1 is 10
Value Pushed in Stack 2 is 11
Value Pushed in Stack 2 is 12
Value Pushed in Stack 2 is 13
Value Pushed in Stack 2 is 14
Value Pushed in Stack 2 is 15
Value Pushed in Stack 2 is 16
Value Pushed in Stack 2 is 17
Value Pushed in Stack 2 is 18
Value Pushed in Stack 2 is 19
Value Pushed in Stack 2 is 20
10 9 8 7 6 5 4 3 2 1
20 19 18 17 16 15 14 13 12 11
Pushing Value in Stack 1 is 11
Stack is full10 is being popped
from Stack 1
9 is being popped from Stack 1
8 is being popped from Stack 1
7 is being popped from Stack 1
6 is being popped from Stack 1
5 is being popped from Stack 1
```

```
//Function to pop data from the Stack1
void pop1 ()
{
// Checking the underflow condition
 if (top1 >= 0)
  {
    int popped_element = array[top1];
    top1--;
    printf ("%d is being popped from Stack 1\n", popped_element);
  }
  else
  {
    printf ("Stack is Empty \n");
  }
}
// Function to remove the element from the Stack2
void pop2 ()
{
// Checking underflow condition
if (top2 < SIZE)
  {
      int popped_element = array[top2];
    top2--;
    printf ("%d is being popped from Stack 1\n", popped_element);
  }
  else
  {
    printf ("Stack is Empty!\n");
  }
}
//Functions to Print the values of Stack1
void display_stack1 ()
{
  int i;
  for (i = top1; i >= 0; --i)
  {
    printf ("%d ", array[i]);
  }
  printf ("\n");
}
// Function to print the values of Stack2
void display_stack2 ()
{
  int i;
  for (i = top2; i < SIZE; ++i)
  {
    printf ("%d ", array[i]);
  }
  printf ("\n");
}
int main()
{
  int ar[SIZE];
  int i;
  int num_of_ele;
  printf ("We can push a total of 20 values\n");
  //Number of elements pushed in stack 1 is 10
  //Number of elements pushed in stack 2 is 10
// loop to insert the elements into Stack1
for (i = 1; i <= 10; ++i)
  {
    push1(i);
    printf ("Value Pushed in Stack 1 is %d\n", i);
  }
// loop to insert the elements into Stack2.
for (i = 11; i <= 20; ++i)
  {
    push2(i);
    printf ("Value Pushed in Stack 2 is %d\n", i);
  }
  //Print Both Stacks
  display_stack1 ();
 display_stack2 ();
  //Pushing on Stack Full
  printf ("Pushing Value in Stack 1 is %d\n", 11);
  push1 (11);
  //Popping All Elements from Stack 1
  num_of_ele = top1 + 1;
  while (num_of_ele)
  {
    pop1 ();
    --num_of_ele;
  }
  // Trying to Pop the element From the Empty Stack
  pop1 ();
  return 0;
}
```

APPLICATIONS OF STACKS

In this section we will discuss typical problems where stacks can be easily applied for a simpleand efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker/ Delimiter Checking
- Expression evaluation:
  - ✓ Conversion of an infix expression into a postfix expression
  - ✓ Evaluation of a postfix expression
  - ✓ Conversion of an infix expression into a prefix expression
  - ✓ Evaluation of a prefix expression
- Recursion
- Tower of Hanoi
- Function calls
- Backtracking
- Syntax parsing

Reversing a List

A list of numbers can be reversed by reading each number from an array starting from the firstindex and pushing it on a stack. Once all the numbers have been read, the numbers can be poppedone at a time and then stored in the array starting from the first index.

Write a program to reverse a list of given numbers.

```
#include <stdio.h>
#include <stdlib.h>
struct linked_list
{
  int data;
  struct linked_list *next;
};
int stack[30], top = -1;
struct linked_list* head = NULL;
int printfromstack(int stack[])
{
  printf("Stack:");
  while(top>=0)
  {
    printf("%d ", stack[top--]);
  }
}
int push(struct linked_list** head, int n)
{
  struct linked_list* newnode = (struct
linked_list*)malloc(sizeof(struct linked_list));
  newnode->data = n;
  newnode->next = (*head);
  (*head) = newnode;
}
int intostack(struct linked_list* head)
 {
  printf("Linked list:");
  while(head!=NULL)
  {
    printf("%d ", head->data);
    stack[++top] = head->data;
    head = head->next;
  }
}
int main(int argc, char const *argv[])
{
  push(&head, 10);
  push(&head, 20);
  push(&head, 30);
  push(&head, 40);
  intostack(head);
  printfromstack(stack);
  return 0;
}
```

Output

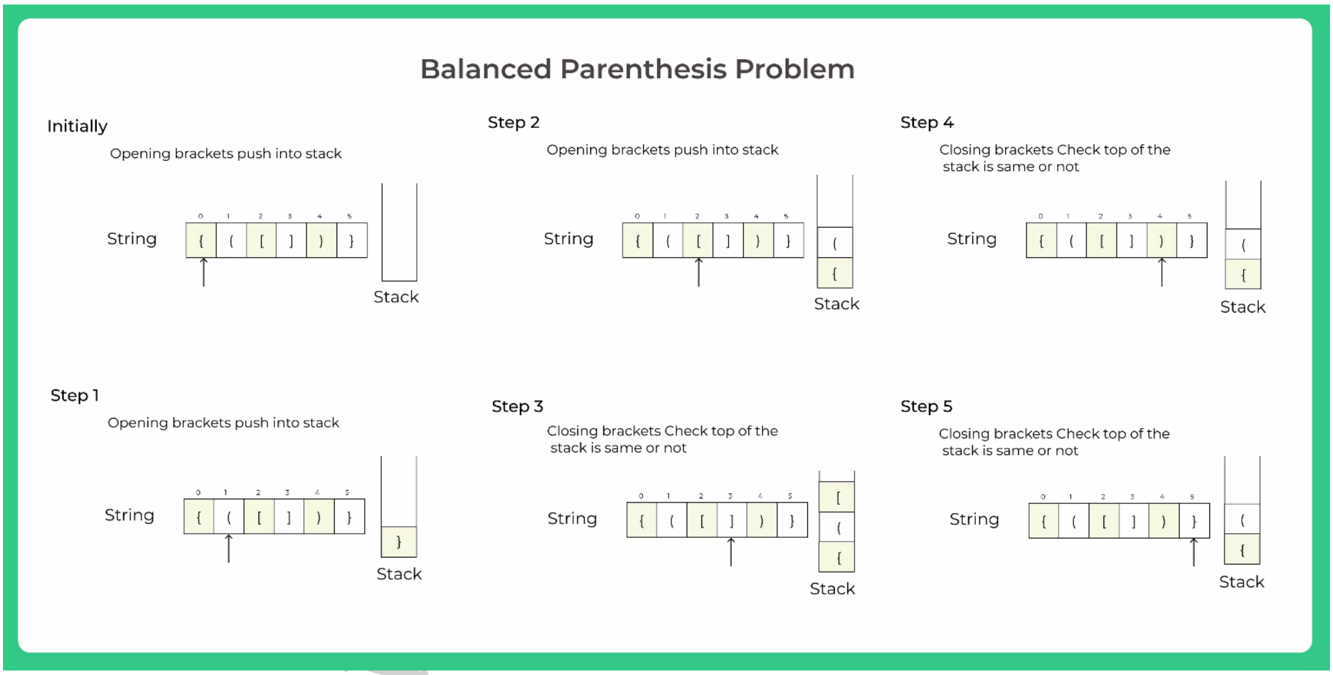
```
Linked list:40 30 20 10
Stack:10 20 30 40
```

Implementing Parentheses Checker

Stacks can be used to check the validity of parentheses in any algebraic expression. For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket. For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid. Look at the program below which traverses an algebraic expression to check for its validity.

Algorithm :

1. Declare a structure for character stack.
2. Now traverse the expression string exp.
3. If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.
4. If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.
5. After complete traversal, if there is some starting bracket left in stack then “NOT BALANCED”


Balanced Parenthesis Problem
Initially
Opening brackets push into stack
String
Stack
Step 1
Opening brackets push into stack
String
Stack
Step 2
Opening brackets push into stack
String
Stack
Step 3
Closing brackets Check top of the stack is same or not
String
Stack
Step 4
Closing brackets Check top of the stack is same or not
String
Stack
Step 5
Closing brackets Check top of the stack is same or not
String
Stack

Write a program to check nesting of parentheses using a stack.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
struct stack
{
  char stck[MAX];
  int top;
}s;
void push(char item)
{
  if (s.top == (MAX - 1))
    printf("Stack is Full\n");
  else
  {
    s.top = s.top + 1;
    s.stck[s.top] = item;
  }
```

```
}
void pop()
{
  if (s.top == -1)
    printf("Stack is Empty\n");

  else
    s.top = s.top - 1;
}
int checkPair(char val1,char val2){
    return (( val1=='(' && val2==')' )||( val1=='[' && val2==']' )||( val1=='{' && val2=='}' ));
}
int checkBalanced(char expr[], int len)
{
      int i;
    for (i = 0; i < len; i++)
    {
        if (expr[i] == '(' || expr[i] == '[' || expr[i] == '{')
        {
          push(expr[i]);
        }
        else
        {
            // exp = {{}}}
            // if you look closely above {{}} will be matched with pair, Thus, stack "Empty"
            //but an extra closing parenthesis like '}' will never be matched
            //so there is no point looking forward
        if (s.top == -1)
            return 0;
        else if(checkPair(s.stck[s.top],expr[i]))
        {
            pop();
            continue;
        }
        // will only come here if stack is not empty
        // pair wasn't found and it's some closing parenthesis
        //Example : {{}}(]
        return 0;
        }
    }
    return 1;
}
int main()
{
  char exp[MAX] = "({})[]{}";
  int i = 0;
  s.top = -1;
  int len = strlen(exp);
  checkBalanced(exp, len)?printf("Balanced"): printf("Not Balanced");
  return 0;
}
```

Evaluation of Arithmetic Expressions

Polish Notations

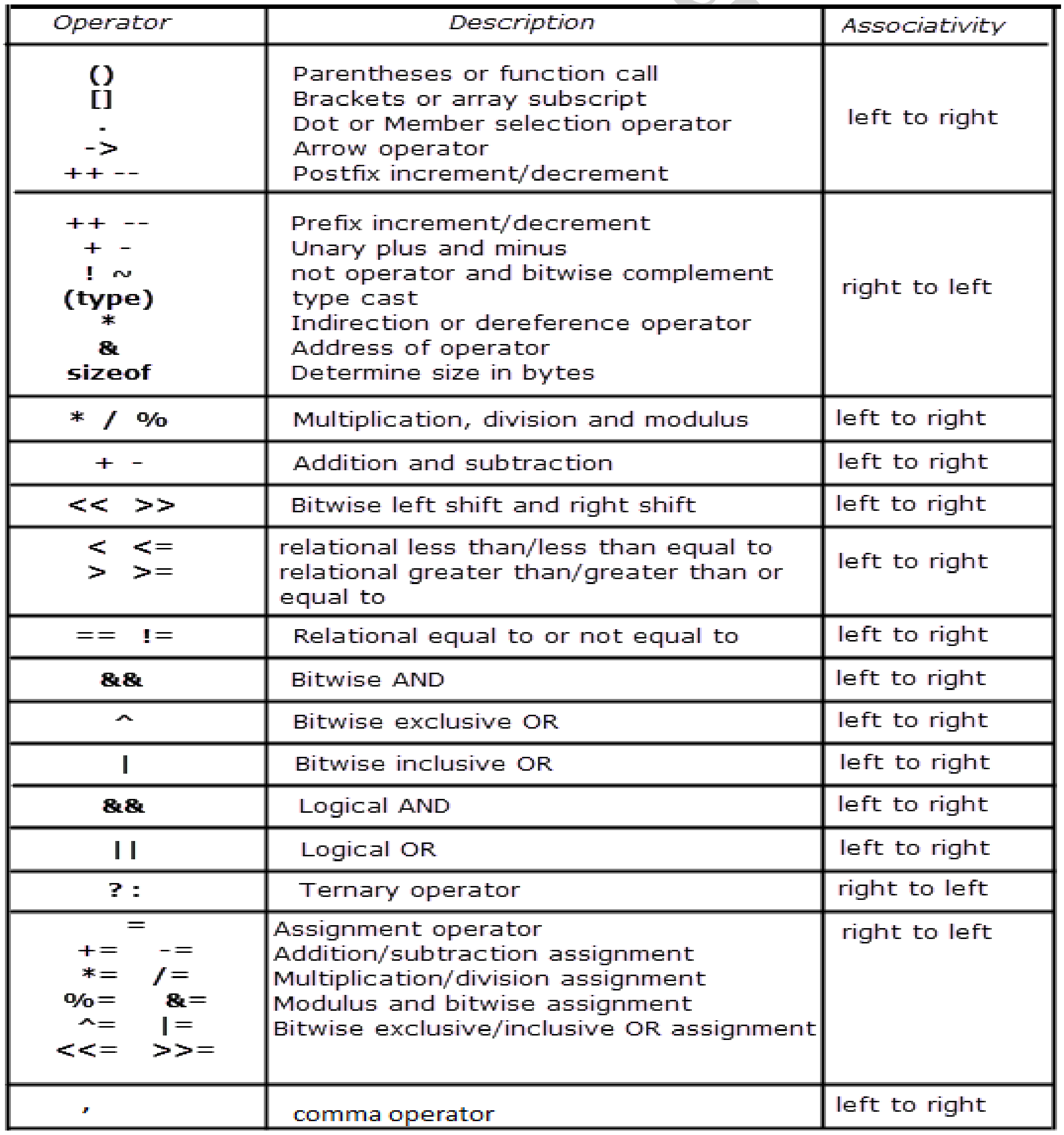
Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is. We all are familiar with the infix notation of writing algebraic expressions.

While writing a arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B; here, plus operator are placed between the two operands A and B. Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence and associability rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix notation was developed by **Jan Łukasiewicz** who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.

In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as A+B in infix notation, the same expression can be written as AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.

Operator Precedence and Associativity in C



Operator	Description	Associativity
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

Evaluation of Arithmetic Expression requires two steps:

- ✓ First, convert the given expression into special notation.
- ✓ Evaluate the expression in this new notation.

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- ➢ Infix Notation
- ➢ Prefix Notation
- ➢ Postfix Notation

Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or un-parenthesized depending upon the problem requirement.

Example: A + B, (C - D) etc.

All these expressions are in infix notation because the operator comes between the operands.

Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example: + A B, -CD etc.

All these expressions are in prefix notation because the operator comes before the operands.

Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

Example: AB +, CD+, etc.

All these expressions are in postfix notation because the operator comes after the operands.

Conversion of an Infix Expression into a Postfix Expression

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, %operators. The precedence of these operators can be given as follows:

Higher priority *, /, %
Lower priority +, –

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression A + B * C, then first B * C will be done and the result will be added to A. But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

The algorithm given below transforms an infix expression into postfix expression, as shown in Fig. The algorithm accepts an infix expression that may contain operators, operands, and parentheses. For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (—) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

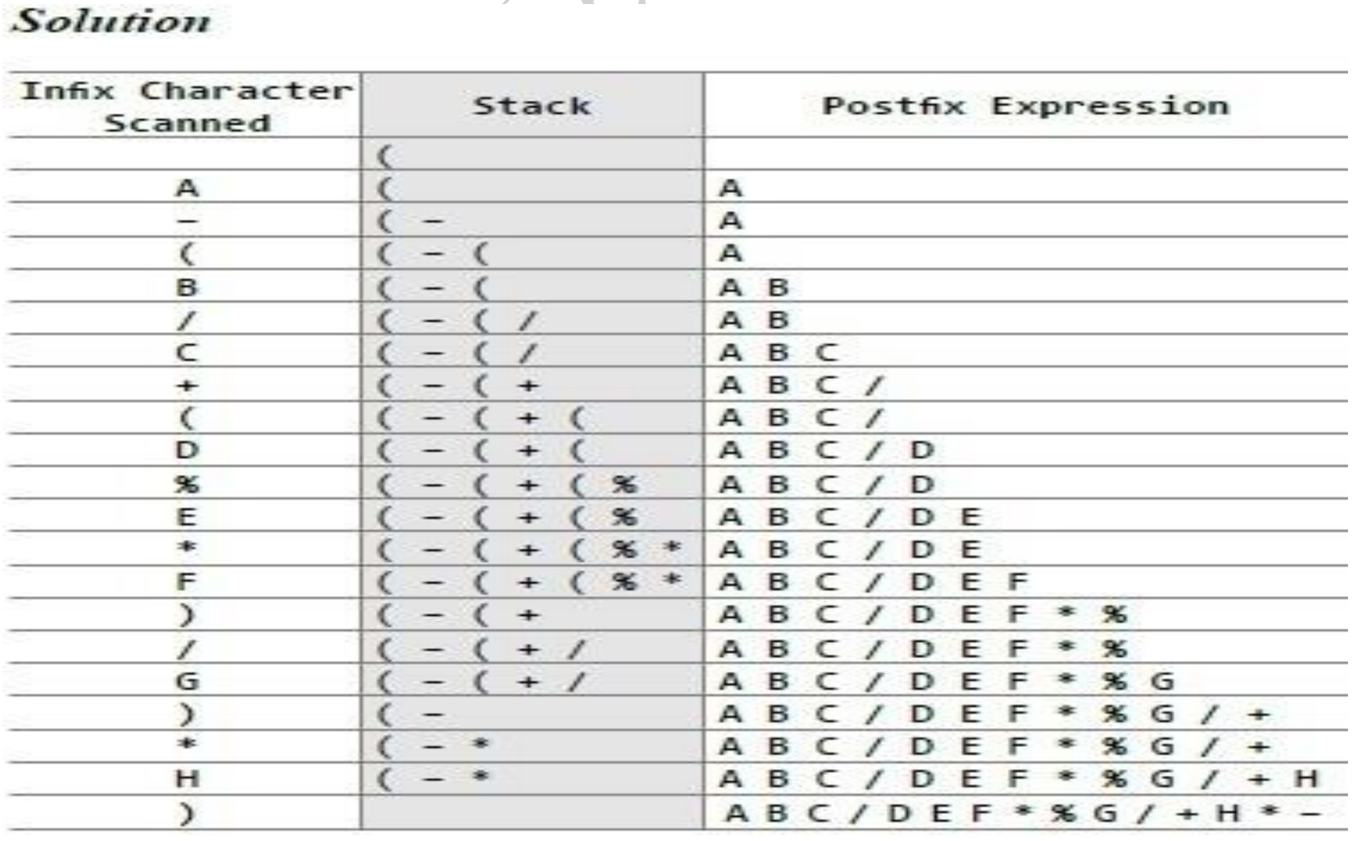
Algorithm to convert an Infix notation into postfix notation

1. Add the open and close parentheses ( ) at starting and ending of the expression
2. Scan each and every individual character of the expression
3. If it symbols or operator encountered push it into the stack
4. IF an operand (whether a digit or an alphabet) is encountered, Add it to the postfix expression.
5. IF a “)” is encountered, then;
   A. Repeatedly pop from stack and add it to the postfix expression until a “(” is encountered.
   B. Discard the “(“. That is, remove the “(“from stack and do not add it to the postfix expression
6. In stack if entered operator is less than precedence of existing operator than compare both the operators, which is having high precedence that operator is added to expression.
7. In Stack if entered operator is same precedence or greater than precedence existing operator than no need to change remain the same.
8. Repeatedly pop from the stack and add it to the postfix expression until the stack is empty.
9. EXIT

Convert the following infix expression into postfix expression using the algorithm

A – (B / C + (D % E * F) / G)* H

A – (B / C + (D % E * F) / G)* H)



Solution

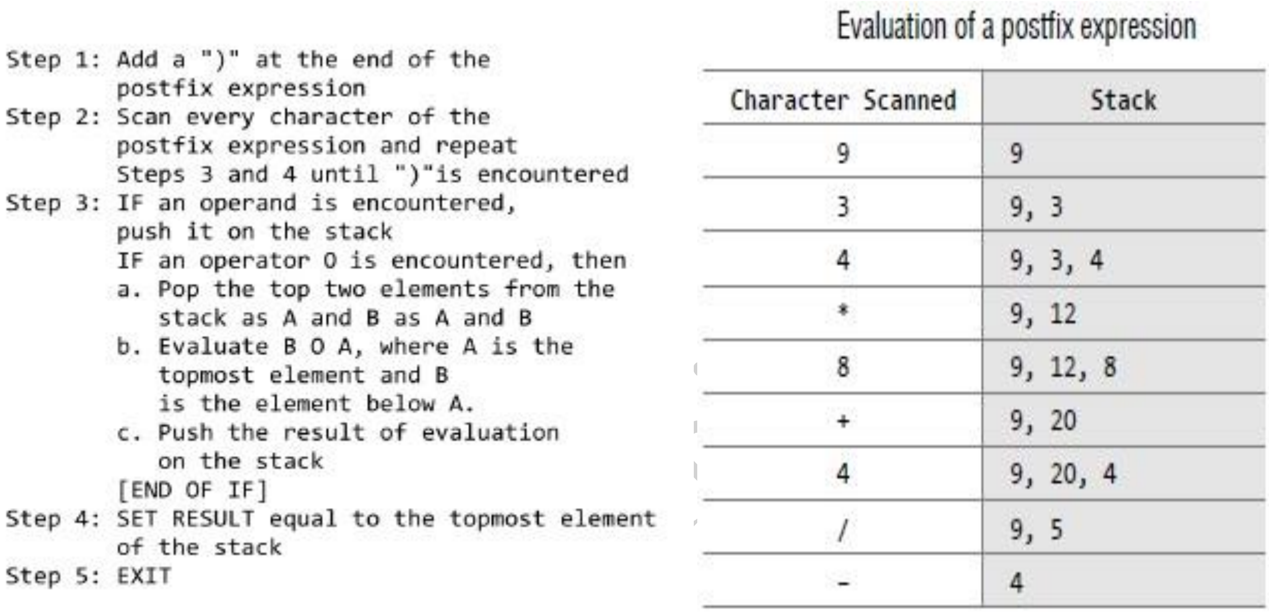
Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

Evaluation of a Postfix Expression

The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.

Both these tasks converting the infix notation into postfix notation and evaluating the postfix expression make extensive use of stacks as the primary tool.

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped fromthe stack and the operator is applied on these values. The result is then pushed on to the stack.



```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Evaluation of a postfix expression

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Let us now take an example that makes use of this algorithm. Consider the infix expressiongiven as 9 – ((3 * 4) + 8) / 4. Evaluate the expression.

The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation. Look at Table 7.1, which shows the procedure.

Recursion

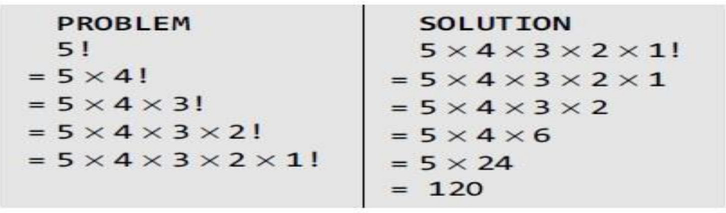
Recursion which is an implicit application of the STACK ADT, A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases.

- **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

To understand recursive functions, let us take an example of calculating factorial of a number. To calculate n!, we multiply the number with factorial of the number that is 1 less than that number. In other words,

n! = n x (n–1)!



PROBLEM	SOLUTION
5!	5 × 4 × 3 × 2 × 1!
= 5 × 4!	= 5 × 4 × 3 × 2 × 1
= 5 × 4 × 3!	= 5 × 4 × 3 × 2
= 5 × 4 × 3 × 2!	= 5 × 4 × 6
= 5 × 4 × 3 × 2 × 1!	= 5 × 24
	= 120

Recursive factorial function

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. Every recursive function must have a base case and a recursive case. For the factorial function,

- Base case is when n = 1, because if n = 1, the result will be 1 as1! = 1.
- Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as factorial(n) = n × factorial (n–1)Look at the following program which calculates the factorial of a number recursively.

NOTE: Every recursive function must have at least one base case. Otherwise, the recursive function will generate an infinite sequence of calls, thereby resulting in an error condition known as an infinite stack**.**

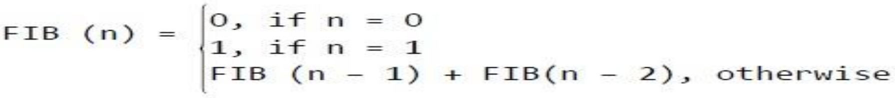
The Fibonacci Series

The Fibonacci series can be given as

0 1 1 2 3 5 8 13 21 34 55 ……

That is, the third term of the series is the sum of the first and second terms. Similarly, fourth term is the sum of second and third terms, and so on. Now we will design a recursive solution to find the nth term of the Fibonacci series. The general formula to do so can be given as.

As per the formula, FIB(0) =0 and FIB(1) = 1. So we have two base cases. This is necessary because every problem is divided into two smaller problems.



$$\text{FIB (n)} = \begin{cases} 0, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ \text{FIB}(n - 1) + \text{FIB}(n - 2), \text{ otherwise} \end{cases}$$

Write a program to print the Fibonacci series using recursion.

```
#include <stdio.h>
int Fibonacci(int);
int main()
{
int n, i = 0, res;
printf("Enter the number of terms\n");
scanf("%d",&n);
printf("Fibonacci series\n");
for(i = 0; i < n; i++ )
{
res = Fibonacci(i);
printf("%d\t",res);
}
```

OUTPUT

```
Enter the number of terms
Fibonacci series
0 1 1 2 3
```

```
return 0;
}
int Fibonacci(int n)
{
if ( n == 0 )
return 0;
else if ( n == 1 )
return 1;
else
return ( Fibonacci(n–1) + Fibonacci(n–2) );
}
```

Tower of Hanoi

The tower of Hanoi is one of the main applications of recursion. It says, ‘if you can solve n–1 cases, then you can easily solve the nth case’.

Three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk. We will be doing this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings (n–1 rings) from the source pole to the spare pole. We move the first two rings from pole A to B .

Now that n–1 rings have been removed from pole A, the nth ring can be easily moved from the source pole (A) to the destination pole (C).

The final step is to move the n–1 rings from the spare pole (B) to the destination pole (C).

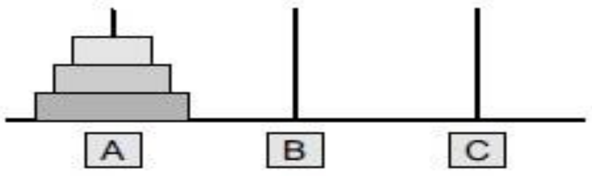
To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

Base case: if n=1

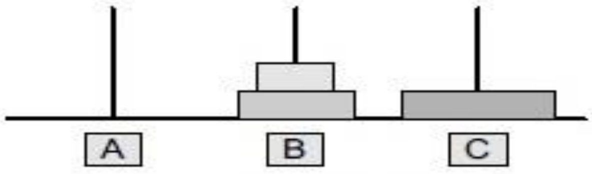
- Move the ring from A to C using B as spare

Recursive case:

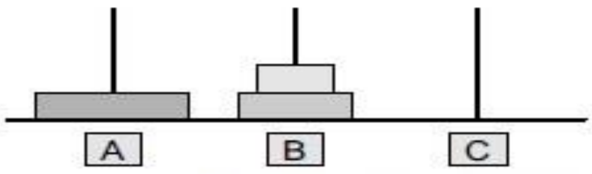
- Move n – 1 rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move n – 1 rings from B to C using A as spare



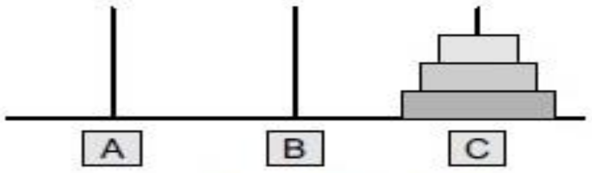
Tower of Hanoi



Move ring from A to C



Move rings from A to B



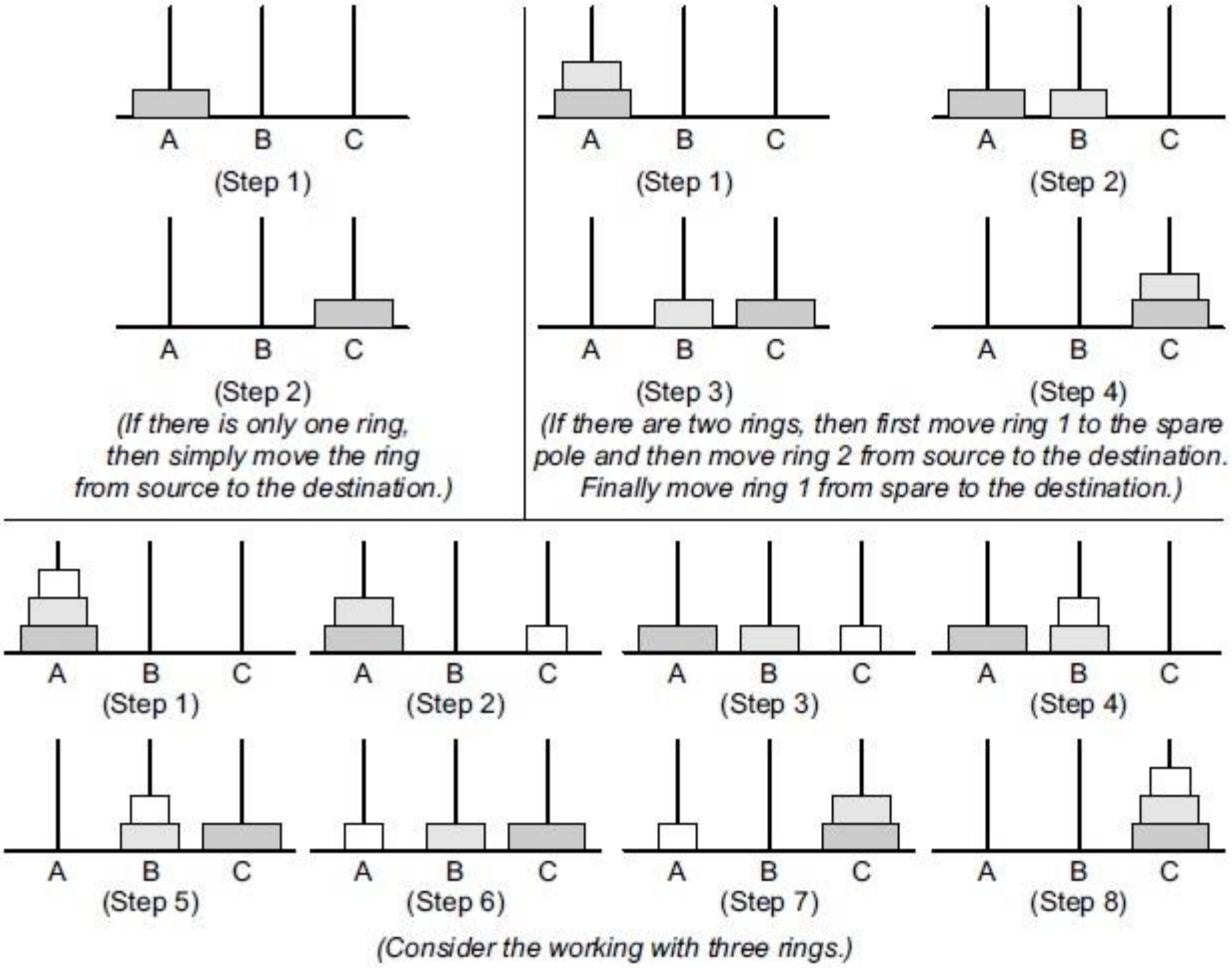
Move ring from B to C

Recursion versus Iteration

Recursion is more of a top-down approach to problem solving in which the original problem is divided into smaller sub-problems. On the contrary, iteration follows a bottom-up approach that begins with what is known and then constructing the solution step by step.

Recursion is an excellent way of solving complex problems especially when the problem can be defined in recursive terms. For such problems, a recursive code can be written and modified in a much simpler and clearer manner.

However, recursive solutions are not always the best solutions. In some cases, recursive programs may require substantial amount of run-time overhead. Therefore, when implementing a recursive solution, there is a trade-off involved between the time spent in constructing and maintaining the program and the cost incurred in running-time and memory space required for the execution of the program.


A B C
(Step 1)
A B C
(Step 2)
(If there is only one ring, then simply move the ring from source to the destination.)
A B C
(Step 1)
A B C
(Step 2)
A B C
(Step 3)
A B C
(Step 4)
(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)
A B C
(Step 1)
A B C
(Step 2)
A B C
(Step 3)
A B C
(Step 4)
A B C
(Step 5)
A B C
(Step 6)
A B C
(Step 7)
A B C
(Step 8)
(Consider the working with three rings.)

Working of Tower of Hanoi with one, two, and three rings

The advantages of using a recursive program include the following:

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Recursion follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

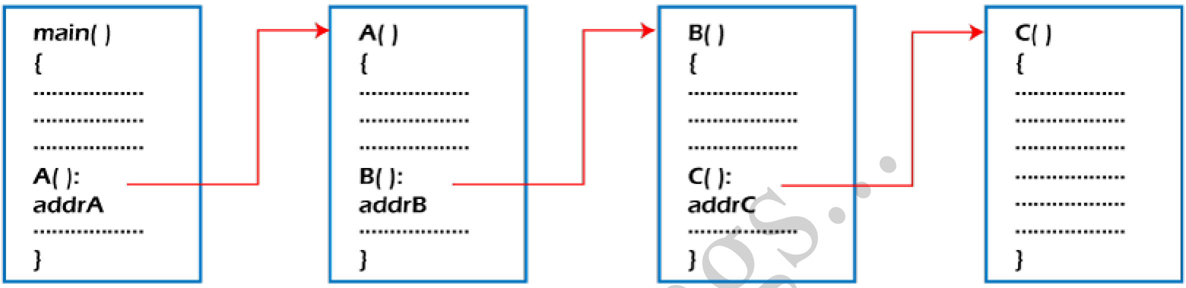
The drawbacks/disadvantages of using a recursive program include the following:

- For some programmers and readers, recursion is a difficult concept.

- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly while using global variables.
- The advantages of recursion pay off for the extra overhead involved in terms of time and space required.

Processing Function Calls:

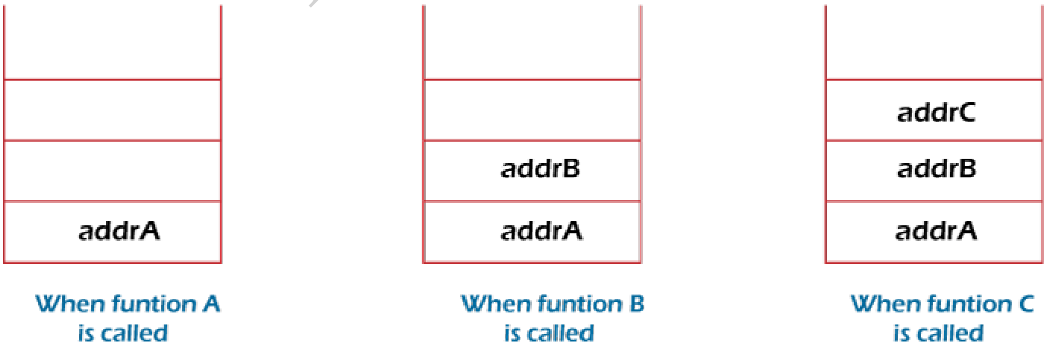
Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Function call

When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.



Different states of stack

The above figure shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack. Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

Backtracking

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options** and **undoing** them if they lead to a **dead end**.

It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like **Sudoku**. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

How Does a Backtracking Algorithm Work?

A **backtracking algorithm** works by recursively exploring all possible solutions to a problem. It starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If an extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension.

The following is a general outline of how a backtracking algorithm works:

1. Choose an initial solution.
2. Explore all possible extensions of the current solution.
3. If an extension leads to a solution, return that solution.
4. If an extension does not lead to a solution, backtrack to the previous solution and try a different extension.
5. Repeat steps 2-4 until all possible solutions have been explored.

When to use a Backtracking algorithm?

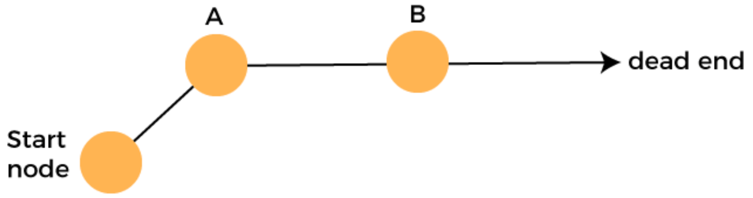
When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

- ✓ A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
- ✓ Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

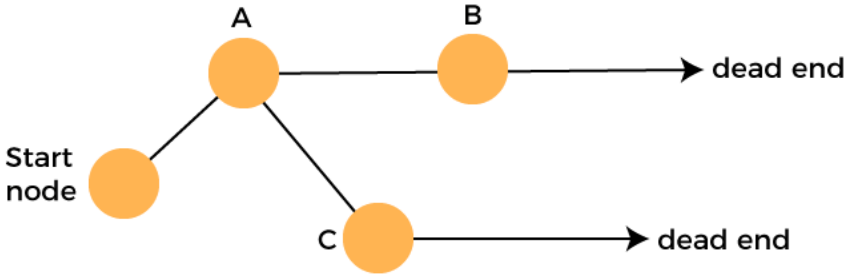
How does Backtracking work?

Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

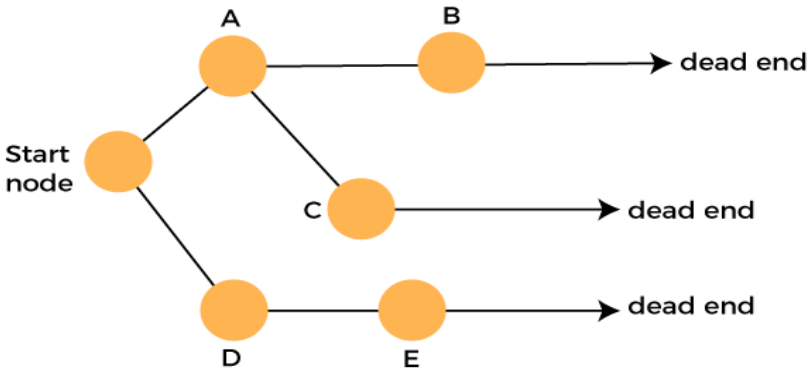
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



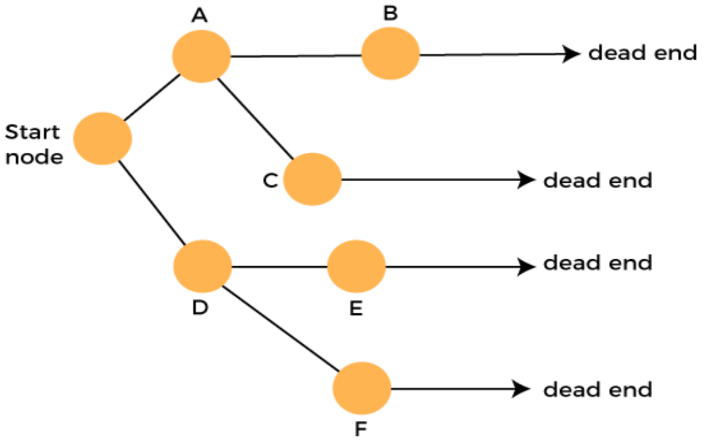
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node



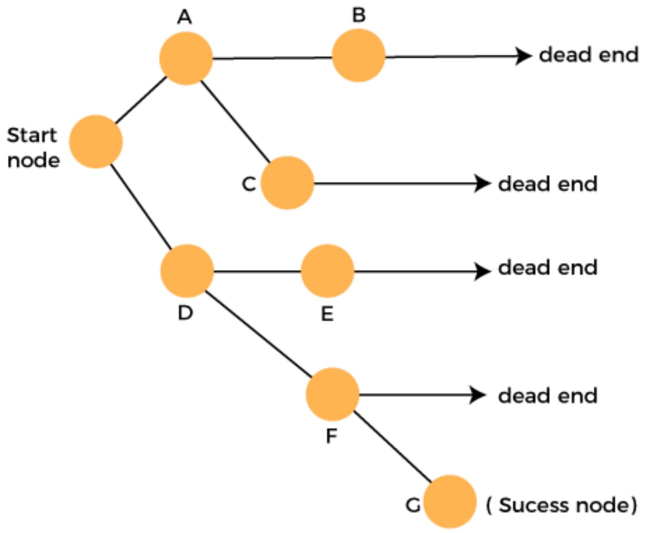
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



The terms related to the backtracking are:

- **Live node:** The nodes that can be further generated are known as live nodes.
- **E node:** The nodes whose children are being generated and become a success node.
- **Success node:** The node is said to be a success node if it provides a feasible solution.
- **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

- **Implicit constraint:** It is a rule in which how each element in a tuple is related.
- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Applications of Backtracking

- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle
- Solving puzzles (e.g., Sudoku, crossword puzzles)
- Finding the shortest path through a maze
- Scheduling problems
- Resource allocation problems
- Network optimization problems

Example 2 of Backtracking Algorithm

Example: Finding the shortest path through a maze

Input: A maze represented as a 2D array, where **0** represents an open space and **1** represents a wall.

Algorithm:

1. Start at the starting point.

2. For each of the four possible directions (up, down, left, right), try moving in that direction.
3. If moving in that direction leads to the ending point, return the path taken.
4. If moving in that direction does not lead to the ending point, backtrack to the previous position and try a different direction.
5. Repeat steps 2-4 until the ending point is reached or all possible paths have been explored.

When to Use a Backtracking Algorithm?

Backtracking algorithms are best used to solve problems that have the following characteristics:

- There are multiple possible solutions to the problem.
- The problem can be broken down into smaller subproblems.
- The subproblems can be solved independently.

Difference between the Backtracking and Recursion

Recursion is a technique that calls the same function again and again until you reach the base case. Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.

Advantages of Stacks:

- ✓ **Simplicity:** Stacks are a simple and easy-to-understand data structure, making them suitable for a wide range of applications.
- ✓ **Efficiency:** Push and pop operations on a stack can be performed in constant time **(O(1))**, providing efficient access to data.
- ✓ **Last-in, First-out (LIFO):** Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed. This behavior is useful in many scenarios, such as function calls and expression evaluation.
- ✓ **Limited memory usage:** Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

Disadvantages of Stacks:

- ✓ **Limited access:** Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
- ✓ **Potential for overflow:** If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
- ✓ **Not suitable for random access:** Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
- ✓ **Limited capacity:** Stacks have a fixed capacity, which can be a limitation if the number of elements that need to be stored is unknown or highly variable.

****

UNIT-IV

Queues: Introduction to queues: properties and operations, implementing queues using arrays and linked lists, Applications of queues in breadth-first search, scheduling, etc.

Deques: Introduction to deques (double-ended queues), Operations on deques and their applications.

A queue is an important data structure which is extensively used in computer applications. In this we will study the operations that can be performed on a queue. Will also discuss the implementation of a queue by using both arrays as well as linked lists, Will illustrate different types of queues like multiple queues, double ended queues, circular queues, and priority queues. And also lists some real-world applications of queues.

INTRODUCTION

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

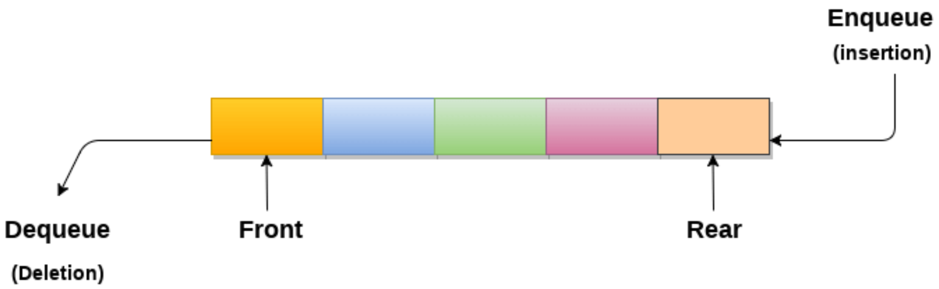
LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

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

Let us explain the concept of queues using the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the **REAR** and removed from the other end called the **FRONT**. Queues can be implemented by using either arrays or linked lists.



Basic Operations of Queue

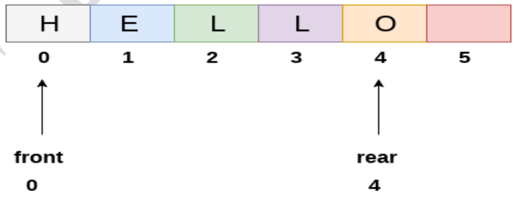
- **Enqueue (Insert)**: Adds an element to the rear of the queue.
- **Dequeue (Delete)**: Removes and returns the element from the front of the queue.
- **Peek**: Returns the element at the front of the queue without removing it.
- **isEmpty**: Checks if the queue is empty.
- **isFull**: Checks if the queue is full.

REPRESENTATION OF QUEUES

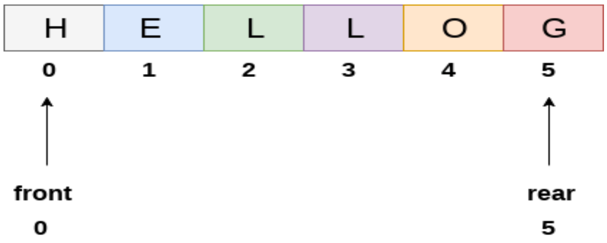
- **Array**
- **Linked List**

ARRAY REPRESENTATION OF QUEUES

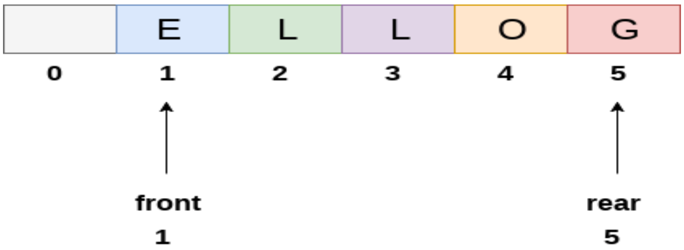
Queues can be easily represented using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1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



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. The value of rear will become 5 while the value of front remains same.



After deleting an element, the value of front will increase from -1 to 0.



ALGORITHM TO INSERT AN ELEMENT IN A QUEUE

```
Step 1: If Rear = Max-1
        Write Overflow
        Goto Step 4
        [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
```

The algorithm to insert an element in a queue, In Step 1, we first check for the overflow condition. In Step 2, we check if the queue is empty. In case the queue is empty, then both FRONT and REAR are set to zero, so that the new value can be stored at the 0th location. Otherwise, if the queue already has some values, then REAR is incremented so that it points to the next location in the array. In Step 3, the value is stored in the queue at the location pointed by REAR.

Similarly, before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If FRONT = –1 and REAR = –1, it means there is no element in the queue.

QUEUE INSERTION LOGIC

```
void insert (int queue[], int max, int front, int rear, int item)
{
   if (rear + 1 == max)
   {
      printf("overflow");
   }
   else
   {
      if(front == -1 && rear == -1)
      {
         front = 0;
```

```
            rear = 0;
        }
        else
        {
            rear = rear + 1;
        }
        queue[rear]=item;
    }
}
```

ALGORITHM TO DELETE AN ELEMENT FROM A QUEUE

```
Step 1: If Front = -1 or Front > Rear
            Write Underflow
            Else
            Set Front = Front + 1
            [End of If]
Step 2: Exit
```

The algorithm to delete an element from a queue, In Step 1, we check for underflow condition. An underflow occurs if FRONT = –1 or FRONT > REAR. However, if queue has some values, then FRONT is incremented so that it now points to the next value in the queue.

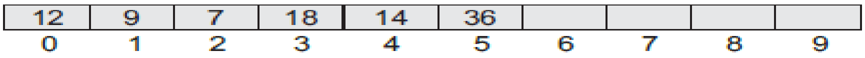
Note: The process of inserting an element in the queue is called en-queue, and the process of deleting an element from the queue is called de-queue.

QUEUE DELECTION LOGIC

```
int delete (int queue[], int max, int front, int rear)
{
    int y;
    if (front == -1 || front > rear)

    {
        printf("underflow");
    }
    else
    {
        y = queue[front];
        if(front == rear)
        {
            front = rear = -1;
            else
            front = front + 1;

        }
        return y;
    }
}
```

EXAMPLE: -

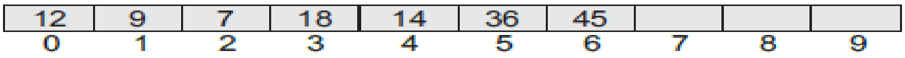
Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.



12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Queue

In above Fig FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.

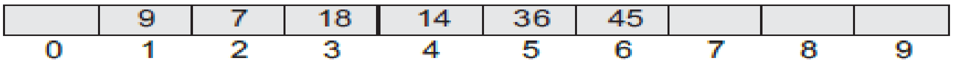


12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Queue after insertion of a new element

Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.

If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue. Here, FRONT = 1 and REAR = 6.



	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

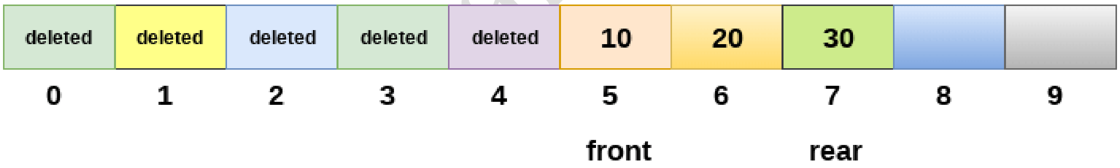
Queue after deletion of an element

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
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");
        }
    }
}
void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");
}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
            front = front + 1;
        }
        printf("\nvalue deleted ");
    }
}
void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {
printf("\nprinting values .....\n");
        for(i=front;i<=rear;i++)
        {
            printf("\n%d\n",queue[i]);
        }
    }
}
```

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.



It shows how the memory space is wasted in the array representation of queue. a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we cannot re-insert the values in the place of already deleted element before the position of front. That much space of the array is wasted and cannot 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.

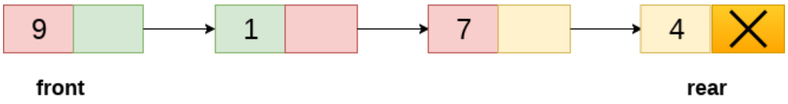
LINKED REPRESENTATION OF QUEUES

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of over head and consuming a lot of time.

In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used. The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).

In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end. If

FRONT = REAR = NULL, then it indicates that the queue is empty.



Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues.

- **Insertion**
- **Deletion.**

The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start of the queue. Apart from this, there is another operation peek which returns the value of the first element of the queue.

Insert Operation

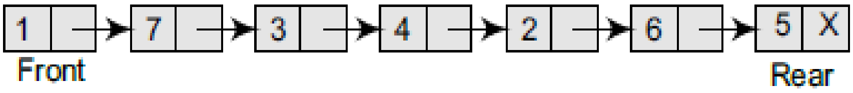
The insert operation is used to insert an element into a queue. The new element is added as thelast element of the queue. To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory fora new node, store the value in its data part and NULL in its next part. The new node will then becalled both FRONT and rear. However, if FRONT!= NULL, then we will insert the new node at the rear end of the linked queue and name this new_node as rear.

Firstly, allocate the memory for the new node ptr

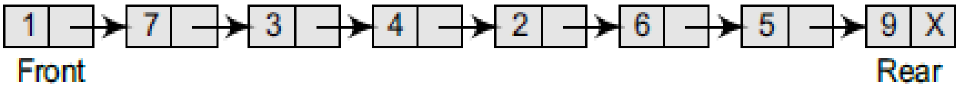
Ptr = (struct node *) malloc (sizeof(struct node));

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.



Linked queue



Linked queue after inserting a new node

ALGORITHM TO INSERT AN ELEMENT IN A LINKED QUEUE

```
➢ 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
```

The algorithm to insert an element in a linked queue, In Step 1, the memory is allocated for the new node. In Step2, the DATA part of the new node is initialized with the value to be stored in the node. In Step3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the new node is tagged as FRONT as well as REAR. Also NULL is stored in the NEXT part of the node (which is also the FRONT and the REAR node). However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or the last node of the queue).

LINKED QUEUE INSERTION LOGIC

```
void insert(struct node *ptr, int item; )
{
   ptr = (struct node *) malloc (sizeof(struct node));
```

```
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

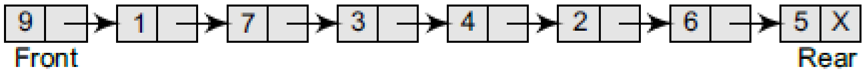
Delete Operation

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed.

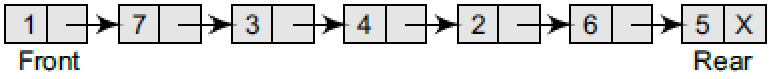
To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue.

ALGORITHM TO DELETE AN ELEMENT IN A LINKED QUEUE

- **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



Linked queue



Linked queue after deletion of an element

The algorithm to delete an element from a linked queue, In Step 1, we first check for the underflow condition. If the condition is true, then an appropriate message is displayed, otherwise in Step 2, we use a pointer PTR that points to FRONT. In Step 3, FRONT is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

LINKED QUEUE DELETION LOGIC

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

Program implementing all the operations on Linked Queue

```
#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");
      }
   }
}
void insert()
{
   struct node *ptr;
   int item;

   ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
      return;
   }
```

```
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
```

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

APPLICATIONS OF QUEUES IN BREADTH- FIRST

Some common applications of Queue data structure:

- **Task Scheduling:** Queues can be used to schedule tasks based on priority or the order in which they were received.
- **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
- **Batch Processing:** Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
- **Message Buffering:** Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
- **Event Handling:** Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
- **Traffic Management:** Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.
- **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.
- **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
- **Printer queues:** In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
- **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
- **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

Useful Applications of Queue

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, etc.

Applications of Queue in Operating systems:

- Semaphores
- FCFS (first come first serve) scheduling, example: FIFO queue
- Spooling in printers
- Buffer for devices like keyboard
- CPU Scheduling
- Memory management

Applications of Queue in Networks:

- Queues in routers/ switches
- Mail Queues
- **Variations:** (Deque, Priority Queue, Doubly Ended Priority Queue)

Some other applications of Queue:

- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add a song at the end or to play from the front.
- Applied on WhatsApp when we send messages to our friends and they don't have an internet connection then these messages are queued on the server of WhatsApp.
- Traffic software (Each light gets on one by one after every time of interval of time.)

BFS algorithm

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories –

- visited
- non-visited.

It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

APPLICATIONS OF BFS ALGORITHM

- BFS can be used to find the neighboring locations from a given source location.
- In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
- BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
- BFS is used to determine the shortest path and minimum spanning tree.
- BFS is also used in Cheney's technique to duplicate the garbage collection.
- It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

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 neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2

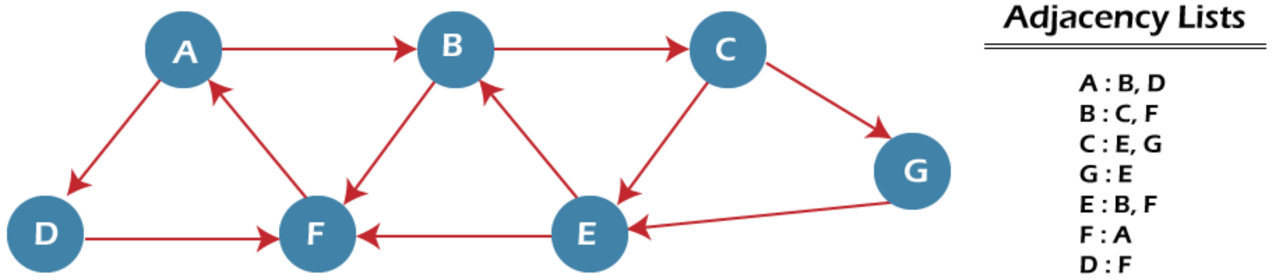
(**waiting state)**

[END OF LOOP]

Step 6: **EXIT**

EXAMPLE OF BFS ALGORITHM

let's understand the working of BFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node 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.

Step 1 - First, add A to queue1 and NULL to queue2.

QUEUE1 = {A}

QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

QUEUE1 = {B, D}

QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

QUEUE1 = {D, C, F}

QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

QUEUE1 = {C, F}

QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

QUEUE1 = {F, E, G}

QUEUE2 = {A, B, D, C}

Step 6 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

QUEUE1 = {E, G}

QUEUE2 = {A, B, D, C, F}

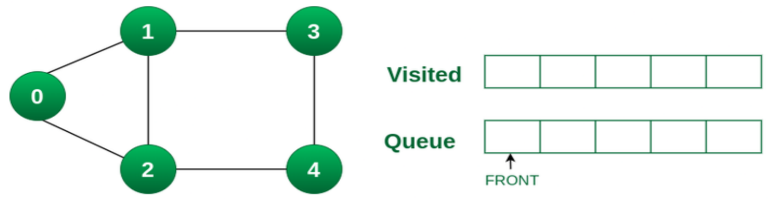
Step 7 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

QUEUE1 = {G}

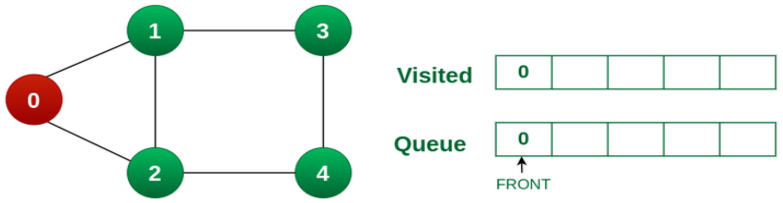
QUEUE2 = {A, B, D, C, F, E}

EXAMPLE 2

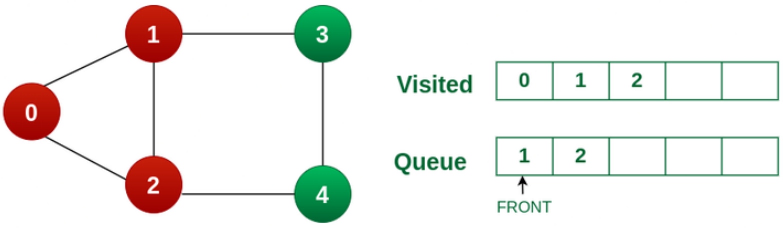
Step1: Initially queue and visited arrays are empty.



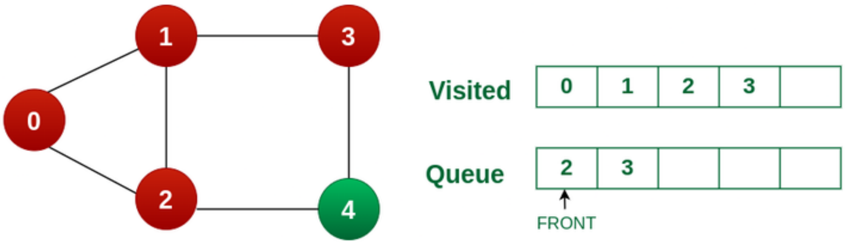
Step2: Push node 0 into queue and mark it visited.



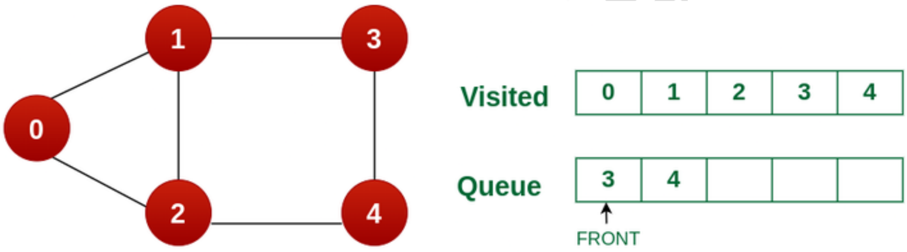
Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.

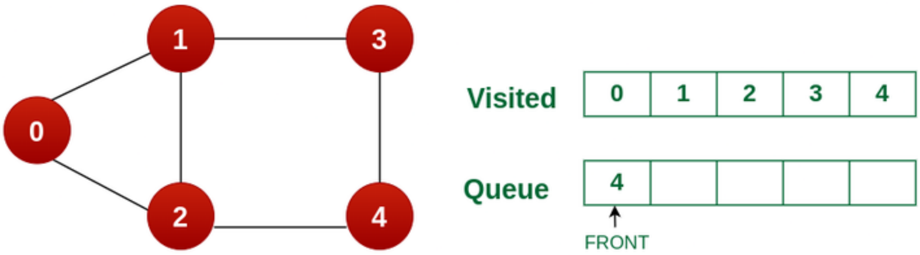


Step 5: Remove node 2 from the front of queue and visit the unvisited neighbors and push them into queue.



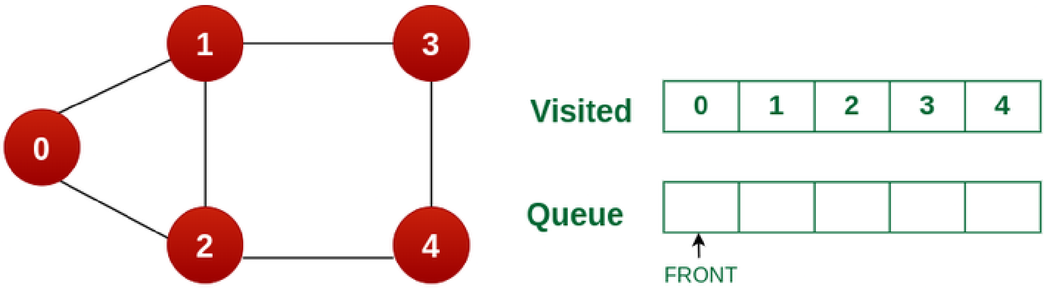
Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbor of node 3 is visited, so move to the next node that are in the front of the queue.



Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbors and push them into queue.

As we can see that every neighbor of node 4 are visited, so move to the next node that is in the front of the queue.



Now, Queue becomes empty, So, terminate this process of iteration.

IMPLEMENTATION OF BFS ALGORITHM

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue
{
  int items[SIZE];
  int front;
  int rear;
};
struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);
struct node
{
  int vertex;
  struct node* next;
};
struct node* createNode(int);
struct Graph
{
  int numVertices;
  struct node** adjLists;
  int* visited;
};
```

OUTPUT

```
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```

```
// BFS algorithm
void bfs(struct Graph* graph, int startVertex)
{
  struct queue* q = createQueue();
  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);
  while (!isEmpty(q))
  {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);
    struct node* temp = graph->adjLists[currentVertex];
    while (temp)
            {
      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0)
              {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }
  }
}
// Creating a node
struct node* createNode(int v)
{
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}
// Creating a graph
struct Graph* createGraph(int vertices)
```

```
{
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;
  graph->adjLists = malloc(vertices * sizeof(struct node*));
  graph->visited = malloc(vertices * sizeof(int));
  int i;
  for (i = 0; i < vertices; i++)
  {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}
// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;
  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
// Create a queue
struct queue* createQueue()
{
  struct queue* q = malloc(sizeof(struct queue));
  q->front = -1;
  q->rear = -1;
  return q;
}
// Check if the queue is empty
int isEmpty(struct queue* q)
{
  if (q->rear == -1)
    return 1;
  else
    return 0;
}
// Adding elements into queue
void enqueue(struct queue* q, int value)
{
  if (q->rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}
// Removing elements from queue
int dequeue(struct queue* q)
 {
  int item;
  if (isEmpty(q))
 {
    printf("Queue is empty");
    item = -1;
  }
else
{
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear)
{
      printf("Resetting queue ");
      q->front = q->rear = -1;
    }
  }
  return item;
}
// Print the queue
void printQueue(struct queue* q)
{
  int i = q->front;
  if (isEmpty(q))
  {
    printf("Queue is empty");
  }
else
{
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++)
            {
      printf("%d ", q->items[i]);
    }
  }
}
int main()
{
  struct Graph* graph = createGraph(6);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 4);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 4);
  bfs(graph, 0);
  return 0;
}
```

COMPLEXITY OF BFS ALGORITHM

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is O(V+E), since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

The space complexity of BFS can be expressed as O(V), where V is the number of vertices.

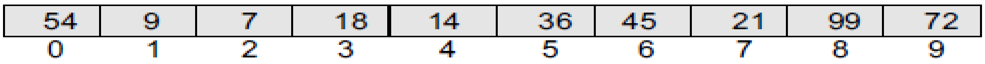
TYPES OF QUEUES

A queue data structure can be classified into the following types:

1. Circular Queue
2. Deque
3. Priority Queue
4. Multiple Queues

Circular Queues

In linear queues, we have discussed so far that insertions can be done only at one end called theREAR and deletions are always done from the other end called the FRONT.

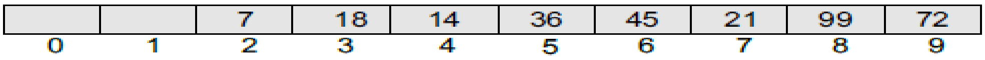


54	9	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Linear queue

Here, FRONT = 0 and REAR = 9.

Now, if you want to insert another value, it will not be possible because the queue is completelyfull. There is no empty space where the value can be inserted. Consider a scenario in which twosuccessive deletions are made.



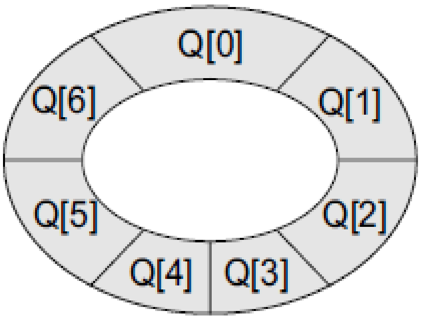
		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Queue after two successive deletions

Here, front = 2 and REAR = 9.

Even though thereis space available, the overflow condition still exists because the condition rear = MAX – 1 still holdstrue. This is a major drawback of a linear queue.

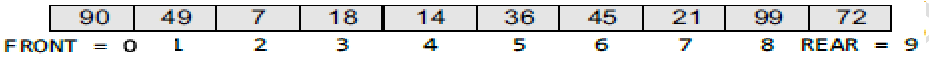
To resolve this problem, we have two solutions. First, shift the elements tothe left so that the vacant space can be occupied and utilized efficiently. Butthis can be very time-consuming, especially when the queue is quite large.The second option is to use a circular queue. In the circular queue, thefirst index comes right after the last index.



Circular queue

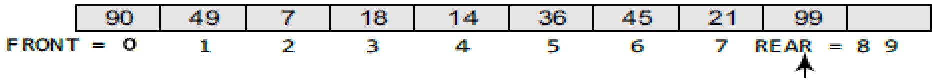
The circular queue will be full only when front = 0 and rear = Max – 1. Acircular queue is implemented in the same manner as a linear queue isimplemented. The only difference will be in thecode that performs insertion and deletionoperations. For insertion, we now have to checkfor the following three conditions:

- If front = 0 and rear = MAX – 1, then the circular queue is full.



Full queue

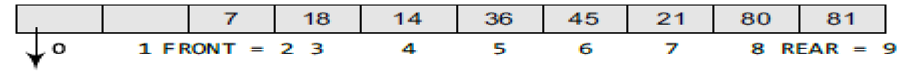
- Ifrear! = MAX – 1, then rear will be incremented and the value will be inserted.



Increment rear so that it points to location 9 and insert the value here

- If front! = 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there

Queue with vacant locations



Set REAR = 0 and insert the value here

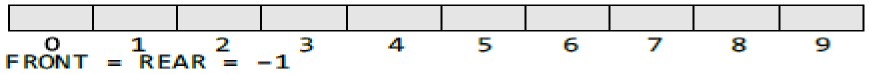
ALGORITHM TO INSERT AN ELEMENT IN A CIRCULAR QUEUE

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
             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
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

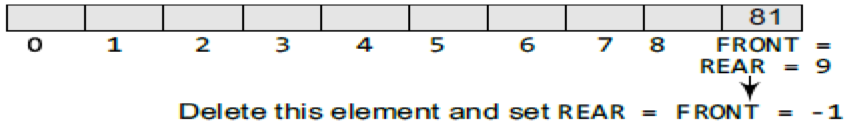
The algorithm to insert an element in a circular queue, In Step 1, we check for the overflow condition. In Step 2, we make two checks. First to see if the queue is empty, and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end. In Step 3, the value is stored in the queue at the location pointed by REAR.

After seeing how a new element is added in a circular queue, let us now discuss how deletions are performed in this case. To delete an element, again we check for three conditions.

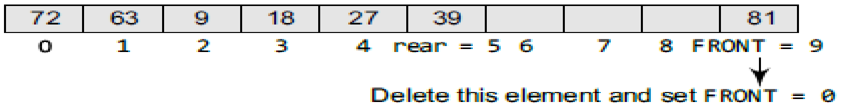
- If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and front = rear, then after deleting the element at the frontthe queue becomes empty and so front and rear are set to –1.
- If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0.



Empty queue



Queue with a single element


72 63 9 18 27 39 81
0 1 2 3 4 rear = 5 6 7 8 FRONT = 9
Delete this element and set FRONT = 0

Queue where FRONT = MAX–1 before deletion

ALGORITHM TO DELETE AN ELEMENT FROM A 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
```

Which shows the algorithm to delete an element from a circular queue, In Step 1, we check for the underflow condition. In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL. In Step 3,we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.

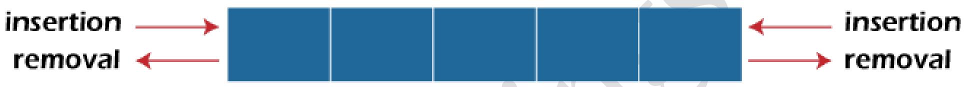
DEQUES

A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Insertion in the queue is done from one end known as the **rear end** or the **tail,** whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the person enters last in the queue gets the ticket at last.

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule.



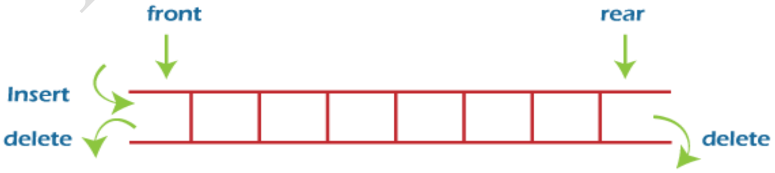
Representation of deque

TYPES OF DEQUE

- Input restricted queue
- Output restricted queue

INPUT RESTRICTED QUEUE

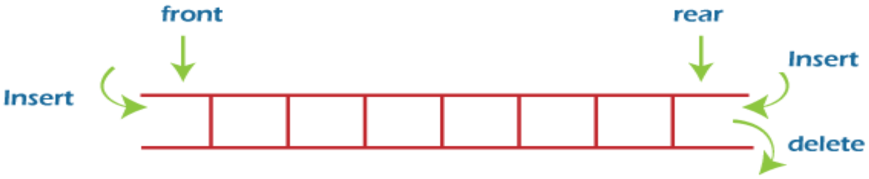
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

OUTPUT RESTRICTED QUEUE

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

OPERATIONS PERFORMED ON DEQUE

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

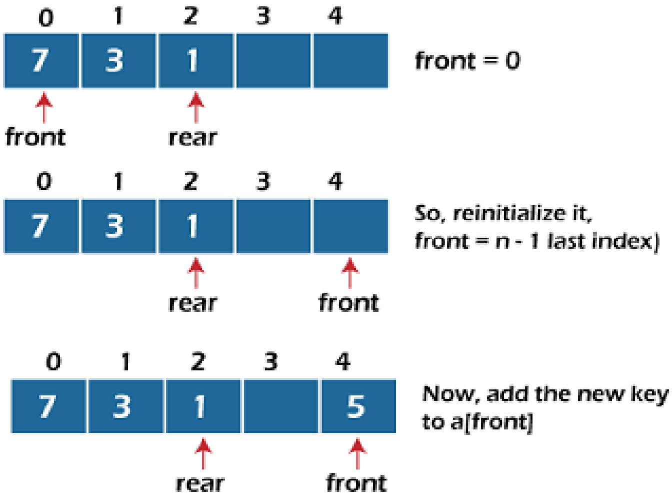
We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque -

- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

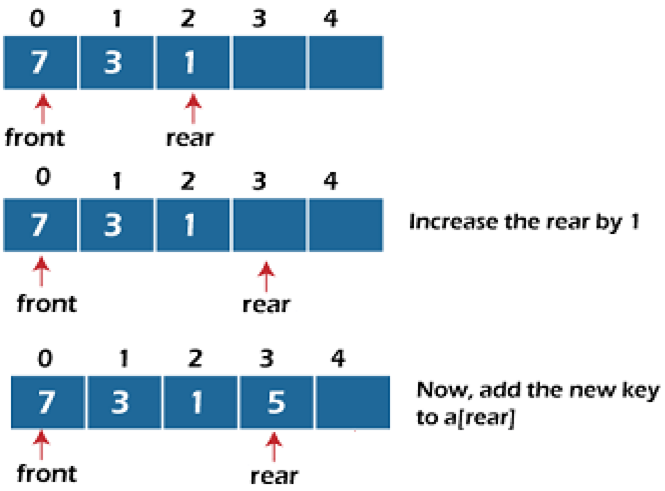
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



INSERTION AT THE REAR END

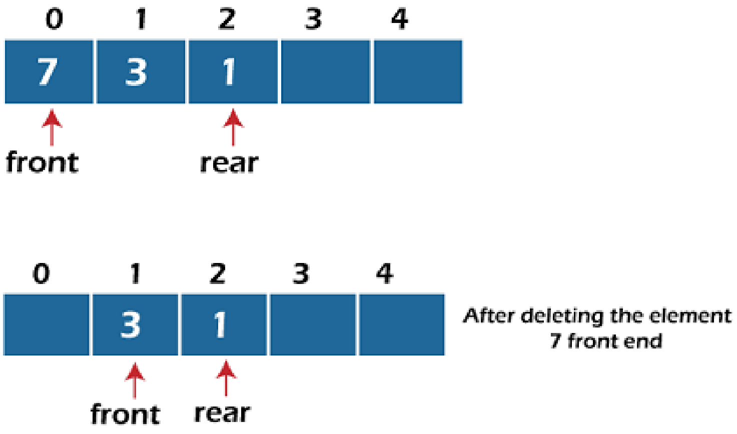
In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end.

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



DELETION AT THE FRONT END

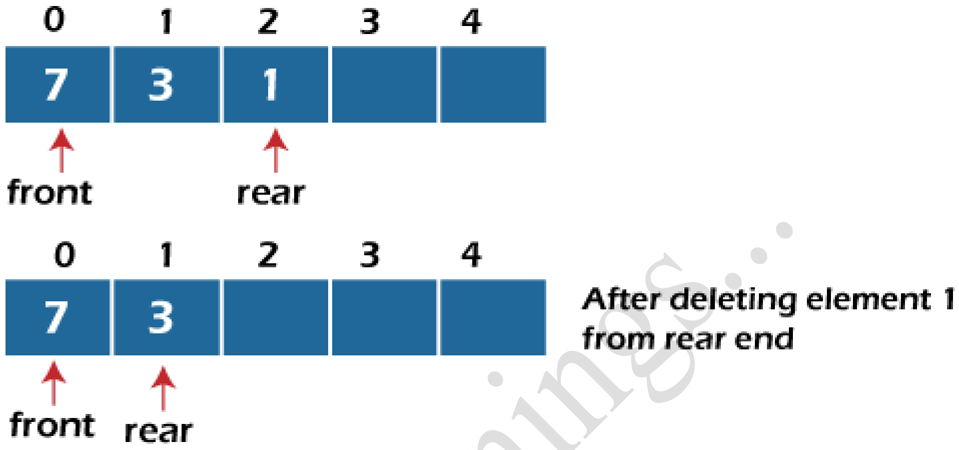
- In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.
- If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end.
    - If the deque has only one element, set rear = -1 and front = -1.
    - Else if front is at end (that means front = size - 1), set front = 0.
    - Else increment the front by 1, (i.e., front = front + 1).



DELETION AT THE REAR END

In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

- If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.
- If the deque has only one element, set rear = -1 and front = -1.
- If rear = 0 (rear is at front), then set rear = n - 1.
- Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

- ✓ This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.
- ✓ The time complexity of all of the above operations of the deque is O(1), i.e., constant.

APPLICATIONS OF DEQUE

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

IMPLEMENTATION OF DEQUE

```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
// insert_front function will insert the value from the front
void insert_front(int x)
{
```

OUTOUT

```
Elements in a deque
are: 10 20 30 50 80
The value of the
element at front is: 10
The value of the
```

```
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    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;
    }
}
// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }
}
// display function prints all the value of deque.
void display()
```

```
element at rear is 80
The deleted element
is 10
The deleted element
is 80
Elements in a deque
are: 20 30 50
```

```
{
   int i=f;
   printf("\nElements in a deque are: ");

   while(i!=r)
   {
      printf("%d ",deque[i]);
      i=(i+1)%size;
   }
    printf("%d",deque[r]);
}
// getfront function retrieves the first value of the deque.
void getfront()
{
   if((f==-1) && (r==-1))
   {
      printf("Deque is empty");
   }
   else
   {
      printf("\nThe value of the element at front is: %d", deque[f]);
   }
}
// getrear function retrieves the last value of the deque.
void getrear()
{
   if((f==-1) && (r==-1))
   {
      printf("Deque is empty");
   }
   else
   {
      printf("\nThe value of the element at rear is %d", deque[r]);
   }
}
// delete_front() function deletes the element from the front
void delete_front()
{
   if((f==-1) && (r==-1))
   {
      printf("Deque is empty");
   }
   else if(f==r)
   {
      printf("\nThe deleted element is %d", deque[f]);
```

```
        f=-1;
        r=-1;
    }
     else if(f==(size-1))
     {
         printf("\nThe deleted element is %d", deque[f]);
         f=0;
     }
     else
     {
          printf("\nThe deleted element is %d", deque[f]);
          f=f+1;
     }
}
// delete_rear() function deletes the element from the rear
void delete_rear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;
    }
     else if(r==0)
     {
         printf("\nThe deleted element is %d", deque[r]);
         r=size-1;
     }
     else
     {
          printf("\nThe deleted element is %d", deque[r]);
          r=r-1;
     }
}
int main()
{
    insert_front(20);
    insert_front(10);
    insert_rear(30);
    insert_rear(50);
    insert_rear(80);
    display();  // Calling the display function to retrieve the values of deque
    getfront();  // Retrieve the value at front-end
    getrear();  // Retrieve the value at rear-end
    delete_front();
    delete_rear();
    display(); // calling display function to retrieve values after deletion
    return 0;
}
```

APPLICATIONS OF DEQUE

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same**.**

PRIORITY QUEUES

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

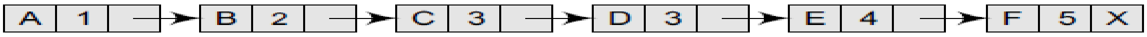
A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely. For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed. However, CPU time is not the only factor that determines the priority, rather it is just one among several factors. Another factor is the importance of one process over another. In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

Implementation of a Priority Queue

There are two ways to implement a priority queue. We can either use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority or we can use an unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed. While a sorted list takes O(n) time to insert an element in the list, it takes only O(1) time to delete an element. On the contrary, an unsorted list will take O(1) time to insert an element and O(n) time to delete an element from the list. Practically, both these techniques are inefficient and usually a blend of these two approaches is adopted that takes roughly O(log n) time or less.

Linked Representation of a Priority Queue

In the computer memory, a priority queue can be represented using arrays or linked lists. When a priority queue is implemented using a linked list, then every node of the list will have three parts:(a) the information or data part, (b) the priority number of the element, and (c) the address of the next element. If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.

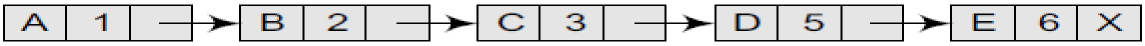


Priority queue

Lower priority number means higher priority. For example, if there are two elements A and B, where A has a priority number 1 and B has a priority number 5, then A will be processed before B as it has higher priority than B.

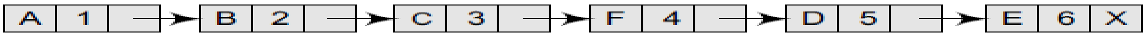
The priority queue is a sorted priority queue having six elements. From the queue, we cannot make out whether A was inserted before E or whether E joined the queue before A because the list is not sorted based on FCFS. Here, the element with a higher priority comes before the element with a lower priority. However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.

Insertion: when a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element.



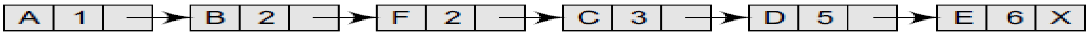
Priority queue

If we have to insert a new element with data = F and priority number = 4, then the element will beinserted before D that has priority number 5, which is lower priority than that of the new element.



Priority queue after insertion of a new node

However, if we have a new element with data = F and priority number = 2, then the element willbe inserted after B, as both these elements have the same priority but the insertions are done onFCFS basis



Priority queue after insertion of a new node

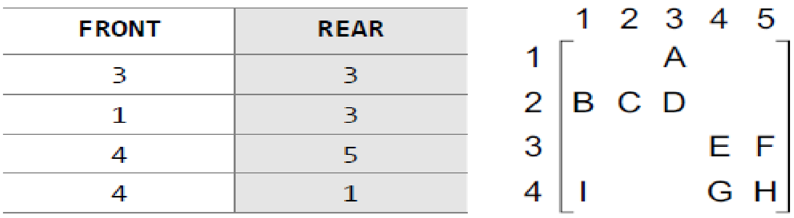
Deletion: Deletion is a very simple process in this case. The first node of the list will be deletedand the data of that node will be processed first.

Array Representation of a Priority Queue

When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.

We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the front and rear values of each queue, the two-dimensional matrix can be formed.

FRONT [K] and REAR[K] contain the front and rear values of row K, where K is the priority number. Note that here we are assuming that the row and column indices start from 1, not 0. Obviously, while programming, we will not take such assumptions.

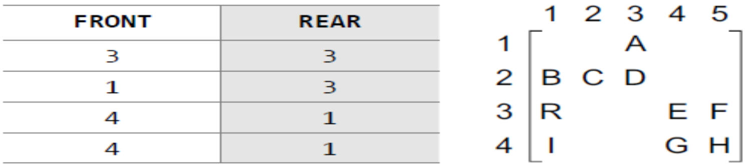


FRONT	REAR
3	3
1	3
4	5
4	1

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & & & & E & F \\ 4 & I & & & G & H \end{array}$$

Priority queue matrix

Insertion: To insert a new element with priority Kin the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if we have to insert an element R with priority number 3,then the priority queue will be.



FRONT	REAR
3	3
1	3
4	1
4	1

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & R & & & E & F \\ 4 & I & & & G & H \end{array}$$

Priority queue matrix after insertion of a new element

Deletion: To delete an element, we find the first non empty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number

1 and the front element is A, so A will be deleted and processed first. In technical terms, find the element with the smallest K, such that FRONT [K] != NULL.

Multiple Queues

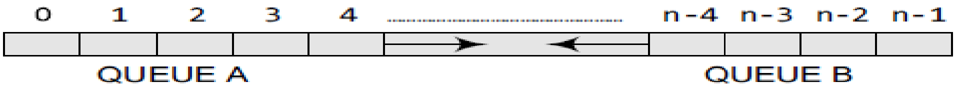
When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.

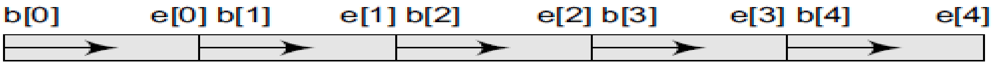
So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size. Figure 8.31 illustrates this concept.

In the figure, an array Queue[n] is used to represent two queues, Queue A and Queue B. The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing queue A will grow from left to right, whereas queue B will grow from right to left at the same time.

Extending the concept to multiple queues, a queue can also be used to represent n number of queues in the same array. That is, if we have a QUEUE[n],then each queue I will be allocated an equal amount of space bounded by indices b[i] and e[i].



Multiple queues



Multiple queues

****

UNIT-V

Trees: Introduction to Trees, Binary Search Tree –Insertion, Deletion & Traversal

Hashing: Brief introduction to hashing and hash functions, Collision resolution techniques: chaining and open addressing, **Hash tables**: basic implementation and operations, Applications of hashing in unique identifier generation, caching, etc.

Introduction

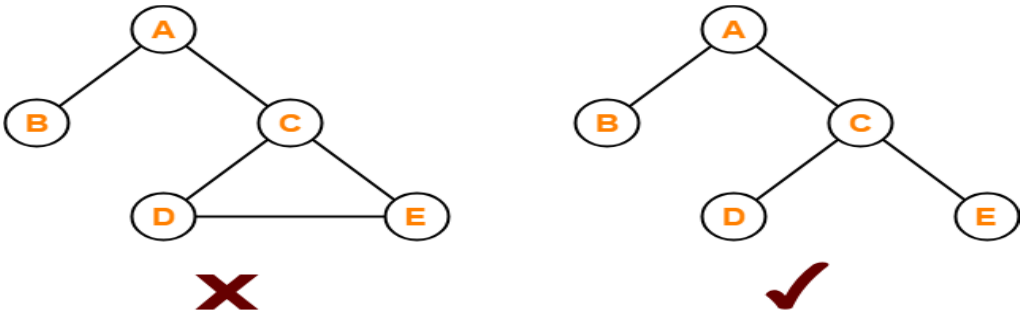
Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

OR

A tree is a connected graph without any circuits.

OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.



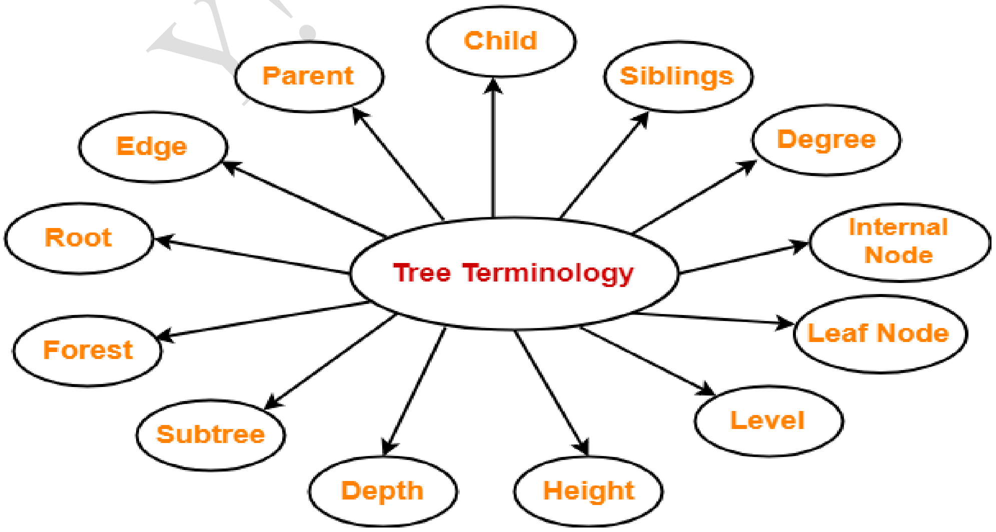
This graph is not a Tree

This graph is a Tree

Properties

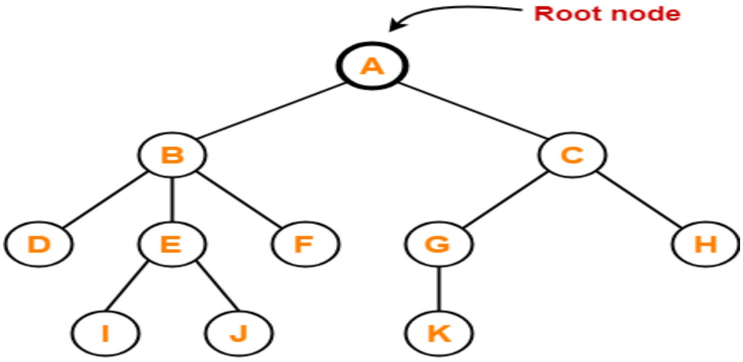
- ✓ There is one and only one path between every pair of vertices in a tree.
- ✓ A tree with n vertices has exactly (n-1) edges.
- ✓ A graph is a tree if and only if it is minimally connected.
- ✓ Any connected graph with n vertices and (n-1) edges is a tree.

Tree Terminology



Root

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

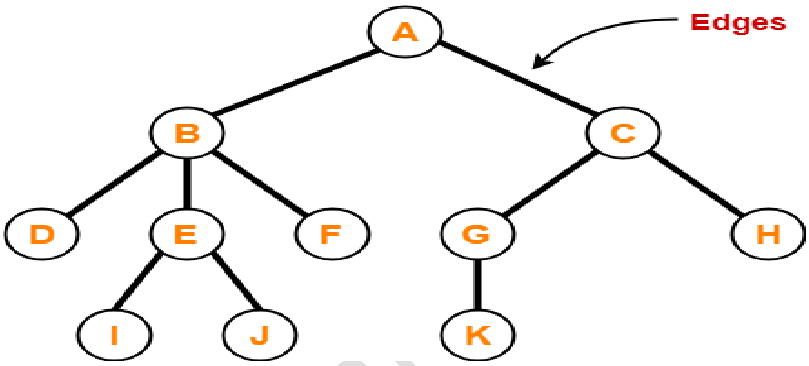


Here, node A is the only root node

Edge

- ✓ The connecting link between any two nodes is called as an edge.
- ✓ In a tree with n number of nodes, there is exactly (n-1) number of edges.

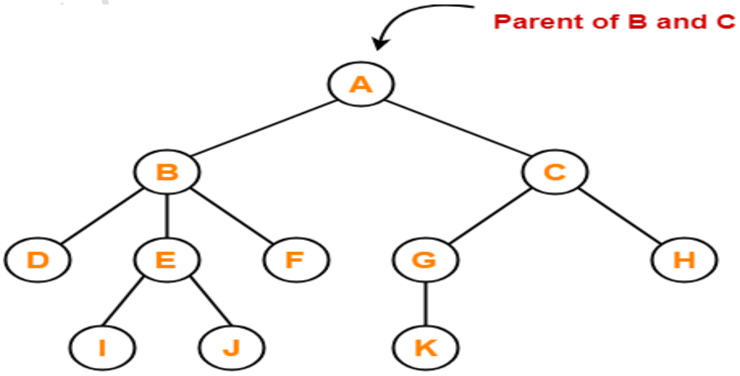
Example



Parent

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

Example



Here,

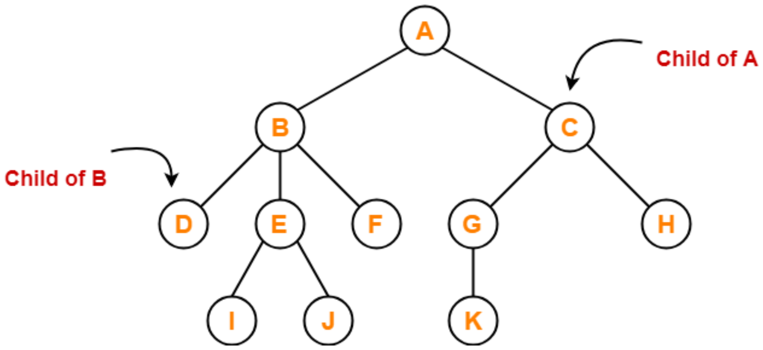
- ✓ Node A is the parent of nodes B and C
- ✓ Node B is the parent of nodes D, E and F
- ✓ Node C is the parent of nodes G and H
- ✓ Node E is the parent of nodes I and J
- ✓ Node G is the parent of node K

Child

- ✓ The node which is a descendant of some node is called as a child node.

- ✓ All the nodes except root node are child nodes.

Example



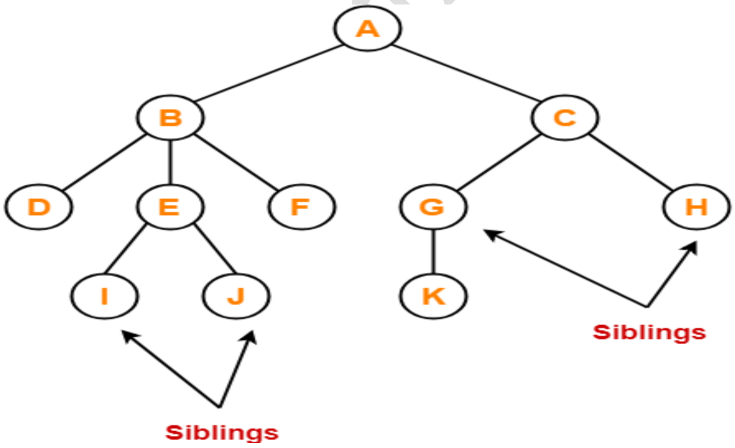
Here,

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

Siblings

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

Example



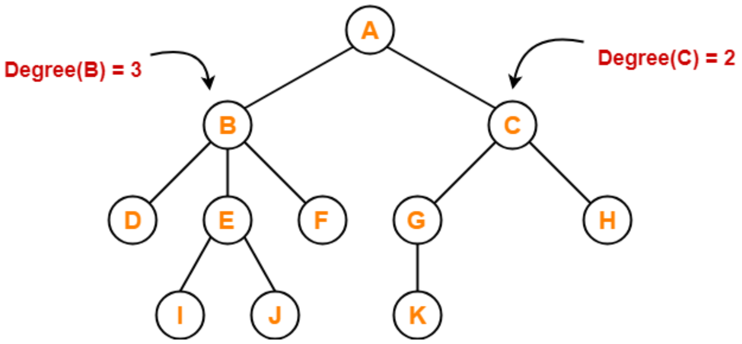
Here,

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

Degree

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

Example



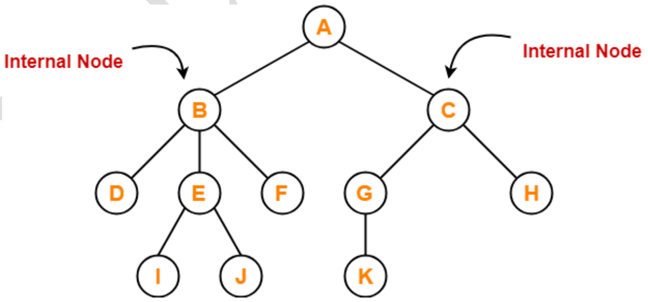
Here,

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

Internal Node

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

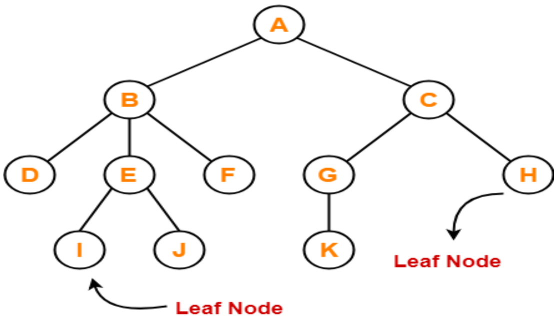
Example



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

Leaf Node

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

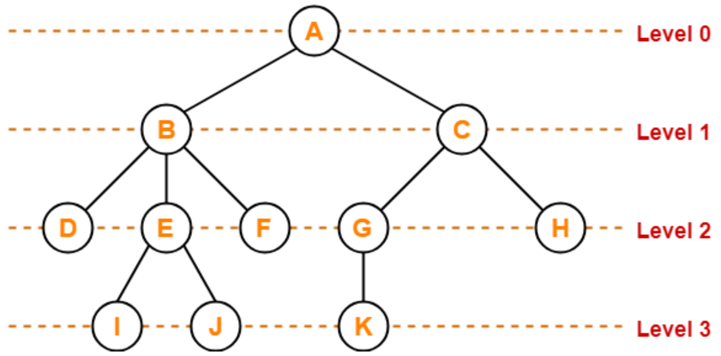


Here, nodes D, I, J, F, K and H are leaf nodes.

Level

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

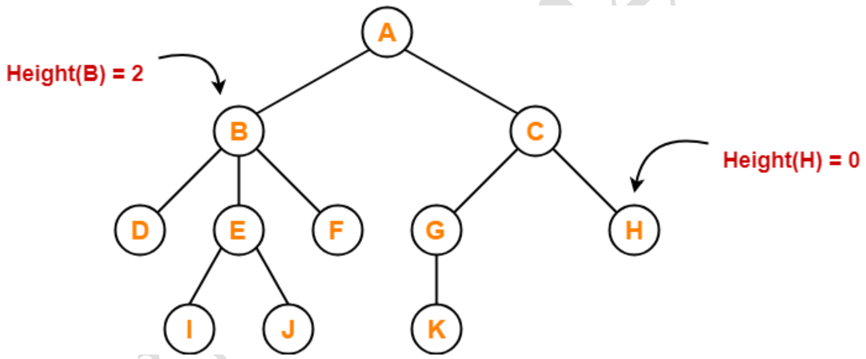
Example



Height

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

Example



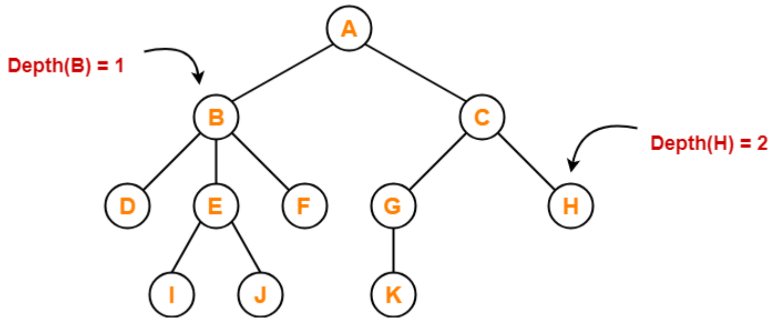
Here

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

Depth

- Total number of edges from root node to a particular node is called as depth of that node.
- Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms "level" and "depth" are used interchangeably.

Example



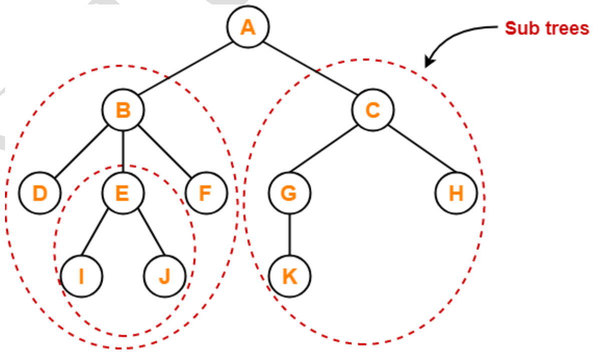
Here

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

Sub-tree

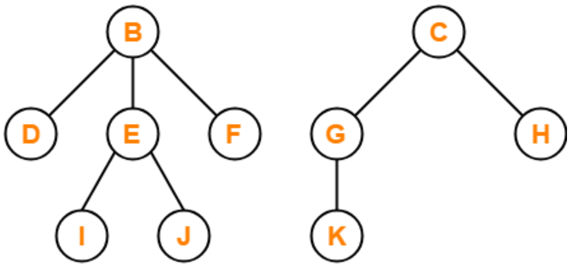
- ✓ In a tree, each child from a node forms a sub-tree recursively.
- ✓ Every child node forms a sub-tree on its parent node.

Example



Forest

- ✓ A forest is a set of disjoint trees.



Forest

Advantages of Tree

- ✓ Tree reflects structural relationships in the data.
- ✓ It is used to represent hierarchies.
- ✓ It provides an efficient insertion and searching operations.
- ✓ Trees are flexible. It allows to move subtrees around with minimum effort.

Types of Trees

1. General trees
2. Forests trees
3. Binary trees
4. Binary search trees
5. Expression trees
6. Tournament trees

General Trees

General trees are data structures that store elements hierarchically. The top node of a tree is the root node and each node, except the root, has a parent. A node in a general tree (except the leaf nodes) may have zero or more sub-trees. General trees which have 3 sub-trees per node are called ternary trees. However, the number of sub-trees for any node may be variable. For example, a node can have 1 sub-tree, whereas some other node can have 3 sub-trees.

Although general trees can be represented as ADTs, there is always a problem when another sub-tree is added to a node that already has the maximum number of sub-trees attached to it. Even the algorithms for searching, traversing, adding, and deleting nodes become much more complex as there are not just two possibilities for any node but multiple possibilities.

To overcome the complexities of a general tree, it may be represented as a graph data structure, thereby losing many of the advantages of the tree processes. Therefore, a better option is to convert general trees into binary trees.

A general tree when converted to a binary tree may not end up being well formed or full, but the advantages of such a conversion enable the programmer to use the algorithms for processes that are used for binary trees with minor modifications.

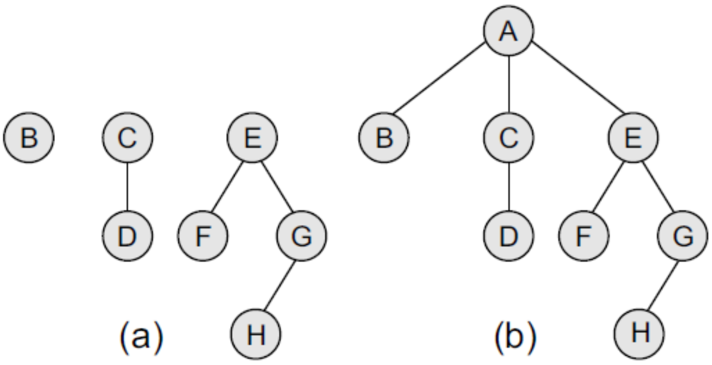
Forests trees

A forest tree is a disjoint union of trees. A set of disjoint trees (or forests) is obtained by deleting the root and the edges connecting the root node to nodes at level 1.

We have already seen that every node of a tree is the root of some sub-tree. Therefore, all the sub-trees immediately below a node form a forest.

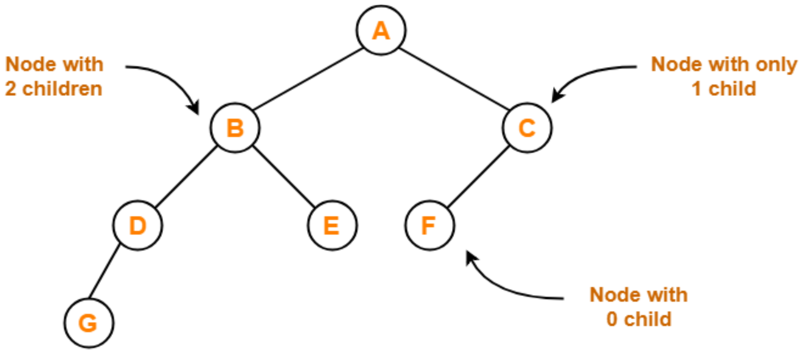
A forest can also be defined as an ordered set of zero or more general trees. While a general tree must have a root, a forest on the other hand may be empty because by definition it is a set, and sets can be empty.

We can convert a forest tree into a general tree by adding a single node as the root node of the tree. Similarly, we can convert a general tree into a forest by deleting the root node of the tree.



Binary Tree

- ✓ Binary tree is a special tree data structure in which each node can have at most 2 children.
- ✓ Thus, in a binary tree, each node has either 0 child or 1 child or 2 children.



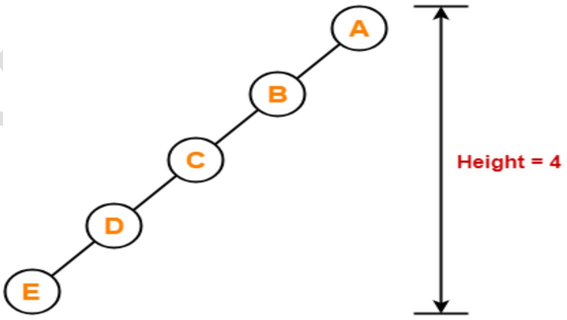
Binary Tree Example

Binary Tree Properties

1 Minimum number of nodes in a binary tree of height H = H + 1

Example

To construct a binary tree of height = 4, we need at least 4 + 1 = 5 nodes.



2. Maximum number of nodes in a binary tree of height H = 2H+1 – 1

Example

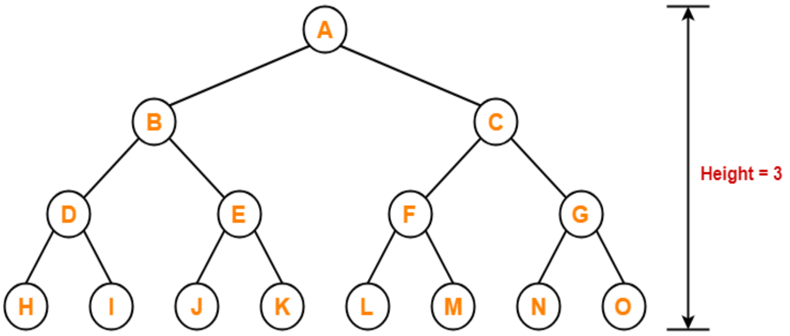
Maximum number of nodes in a binary tree of height 3

= 23+1 – 1

= 16 – 1

= 15 node

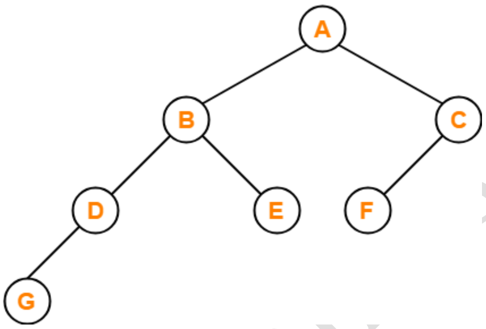
Thus, in a binary tree of height = 3, maximum number of nodes that can be inserted = 15.



We cannot insert more number of nodes in this binary tree

3. Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1

Example



Here

Number of leaf nodes = 3

Number of nodes with 2 children = 2

Clearly, number of leaf nodes is one greater than number of nodes with 2 children. This verifies the above relation.

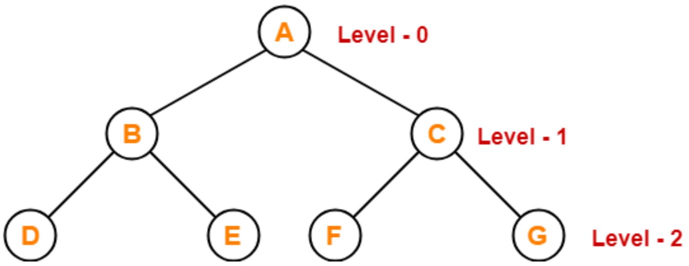
4. Maximum number of nodes at any level ‘L’ in a binary tree = 2L

Example

Maximum number of nodes at level-2 in a binary tree

= 22

= 4



Thus, in a binary tree, maximum number of nodes that can be present at level-2 = 4.

Types of Binary Trees

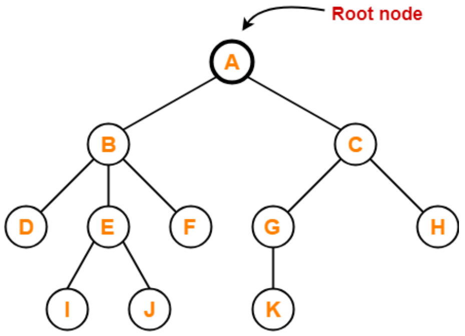
- ✓ Rooted Binary Tree
- ✓ Full / Strictly Binary Tree
- ✓ Complete / Perfect Binary Tree
- ✓ Almost Complete Binary Tree
- ✓ Skewed Binary Tree

1. Rooted Binary Tree

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

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

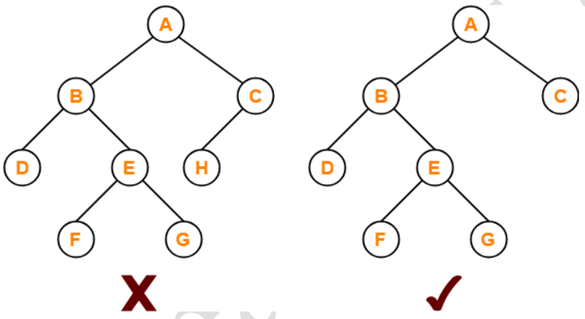
Example:



2. Full / Strictly Binary Tree-

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

Example:



Here

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

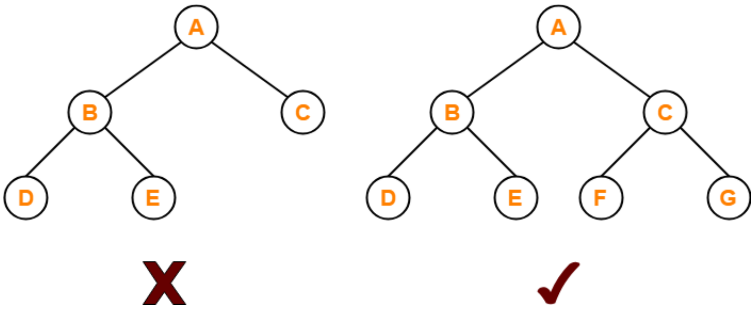
3. Complete / Perfect Binary Tree

A complete binary tree is a binary tree that satisfies the following 2 properties

- ✓ Every internal node has exactly 2 children.
- ✓ All the leaf nodes are at the same level.

Complete binary tree is also called as Perfect binary tree.

Example:



Here

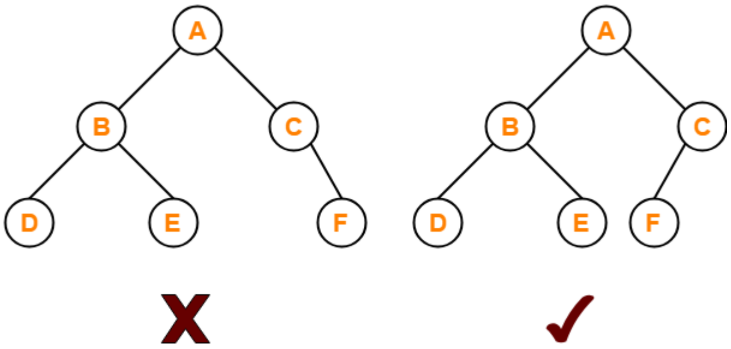
- ✓ First binary tree is not a complete binary tree.
- ✓ This is because all the leaf nodes are not at the same level.

4. Almost Complete Binary Tree

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

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

Example:



5. Skewed Binary Tree

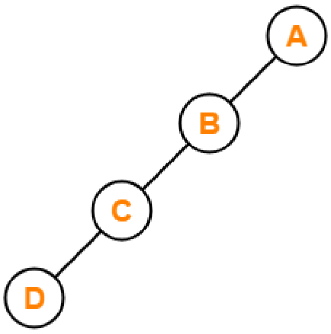
A skewed binary tree is a binary tree that satisfies the following 2 properties

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

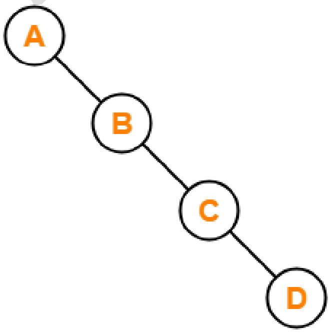
OR

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

Example:



Left Skewed Binary Tree



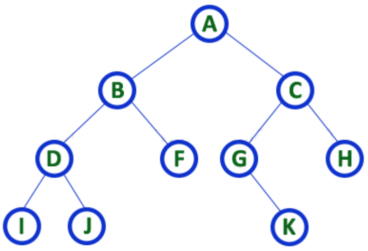
Right Skewed Binary Tree

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

- ✓ Array Representation
- ✓ Linked List Representation

Consider the following binary tree



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

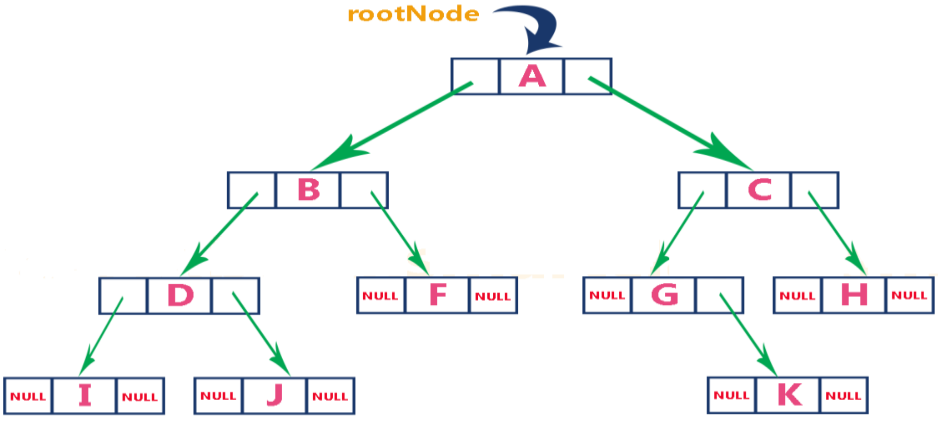
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

Left Child Address	Data	Right Child Address

The above example of the binary tree represented using Linked list representation is



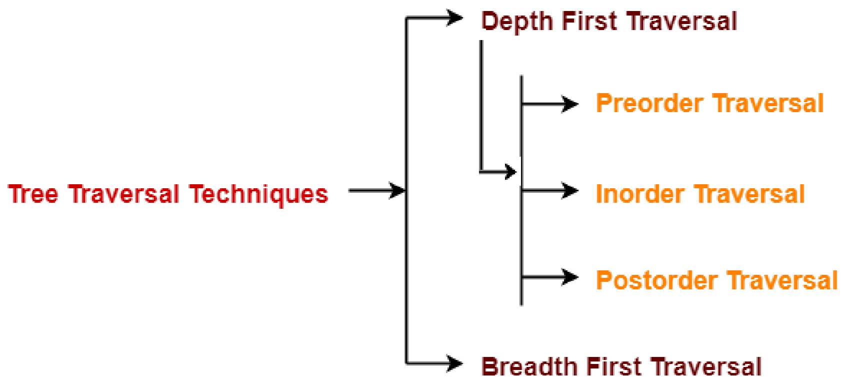
Linked Representation

```
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

Binary Tree Traversals

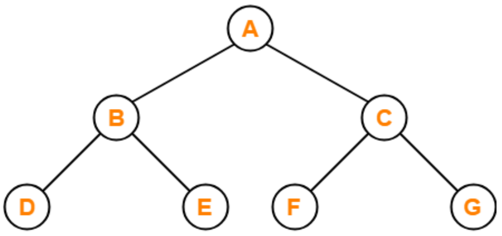
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree.

- ✓ **Pre-order Traversal**
- ✓ **In-order Traversal**
- ✓ **Post-order Traversal**



Pre-order Traversal

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



Preorder Traversal : A , B , D , E , C , F , G

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

A → B → D → E → C → F → G

Steps

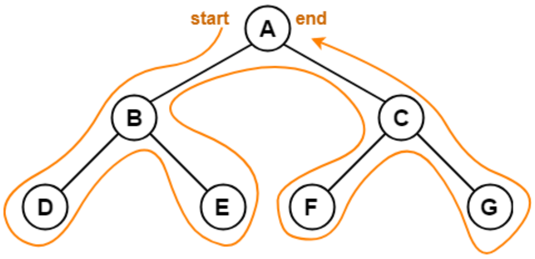
- ✓ Visit the root node
- ✓ traverse the left sub-tree in pre-order
- ✓ traverse the right sub-tree in pre-order

Root → Left → Right

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE! = NULL
Step 2: Write TREE -> DATA
Step 3: PREORDER (TREE -> LEFT)
Step 4: PREORDER (TREE -> RIGHT)
[END OF LOOP]
Step 5: END
```

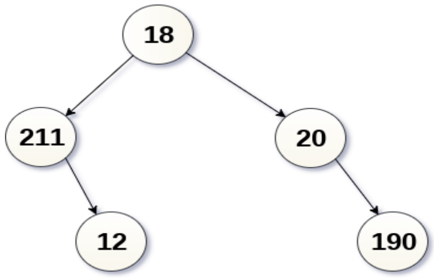
Traverse the entire tree starting from the root node keeping yourself to the left.



Preorder Traversal : A , B , D , E , C , F , G

Example

Traverse the following binary tree by using pre-order traversal



- ✓ Since, the traversal scheme, we are using is pre-order traversal, therefore, the first element to be printed is 18.
- ✓ Traverse the left sub-tree recursively. The root node of the left sub-tree is 211, print it and move to left.
- ✓ Left is empty therefore print the right children and move to the right sub-tree of the root.
- ✓ 20 are the root of sub-tree therefore, print it and move to its left. Since left sub-tree is empty therefore move to the right and print the only element present there i.e. 190.
- ✓ Therefore, the printing sequence will be 18, 211, 90, 20, and 190.

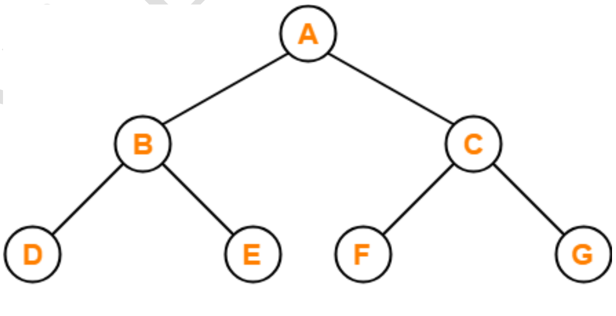
Applications

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

In order Traversal

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

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



Inorder Traversal : D , B , E , A , F , C , G

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

D → B → E → A → F → C → G

Steps

- ✓ Traverse the left sub-tree in in-order
- ✓ Visit the root
- ✓ Traverse the right sub-tree in in-order

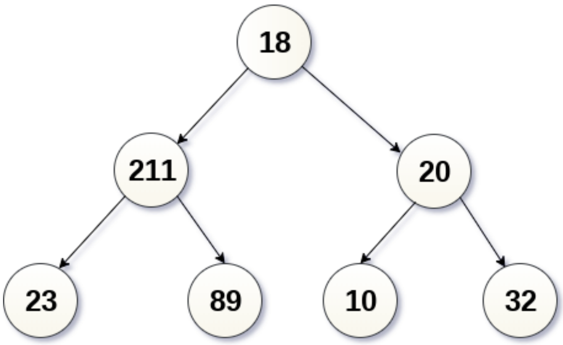
Left → Root → Right

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE! = NULL
Step 2: INORDER (TREE –> LEFT)
Step 3: Write TREE –> DATA
Step 4: INORDER (TREE –> RIGHT)
[END OF LOOP]
Step 5: END
```

Example

Traverse the following binary tree by using in-order traversal.



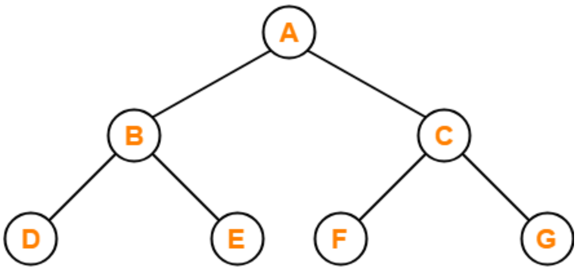
- ✓ Print the left most node of the left sub-tree i.e. 23.
- ✓ Print the root of the left sub-tree i.e. 211.
- ✓ Print the right child i.e. 89.
- ✓ Print the root node of the tree i.e. 18.
- ✓ Then, move to the right sub-tree of the binary tree and print the left most node i.e. 10.
- ✓ Print the root of the right sub-tree i.e. 20.
- ✓ Print the right child i.e. 32.
- ✓ Hence, the printing sequence will be 23, 211, 89, 18, 10, 20, and 32.

Application

In order traversal is used to get infix expression of an expression tree.

Post-order Traversal

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



Postorder Traversal : D , E , B , F , G , C , A

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

D → E → B → F → G → C → A

Steps

- Traverse the left sub-tree in post-order
- Traverse the right sub-tree in post-order
- visit the root

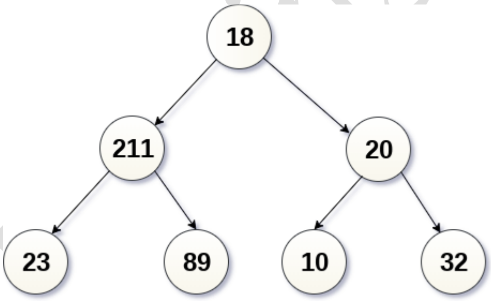
Left → Right → Root

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE! = NULL
Step 2: POSTORDER (TREE -> LEFT)
Step 3: POSTORDER (TREE -> RIGHT)
Step 4: Write TREE -> DATA
[END OF LOOP]
Step 5: END
```

Example

Traverse the following tree by using post-order traversal



- Print the left child of the left sub-tree of binary tree i.e. 23.
- Print the right child of the left sub-tree of binary tree i.e. 89.
- Print the root node of the left sub-tree i.e. 211.
- Now, before printing the root node, move to right sub-tree and print the left child i.e. 10.
- Print 32 i.e. right child.
- Print the root node 20.
- Now, at the last, print the root of the tree i.e. 18.
- The printing sequence will be 23, 89, 211, 10, 32, and 18.

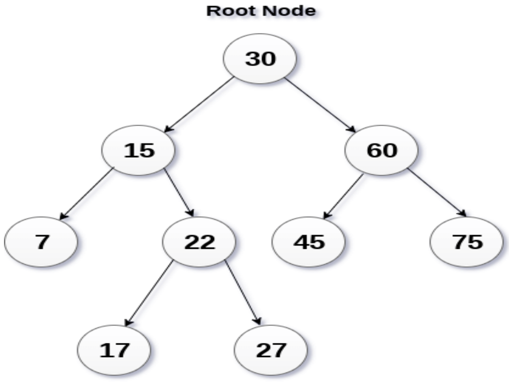
Applications

- Post order traversal is used to get postfix expression of an expression tree.
- Post order traversal is used to delete the tree.
- This is because it deletes the children first and then it deletes the parent.

Binary Search Trees

- 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.
- In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.

- ✓ 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.

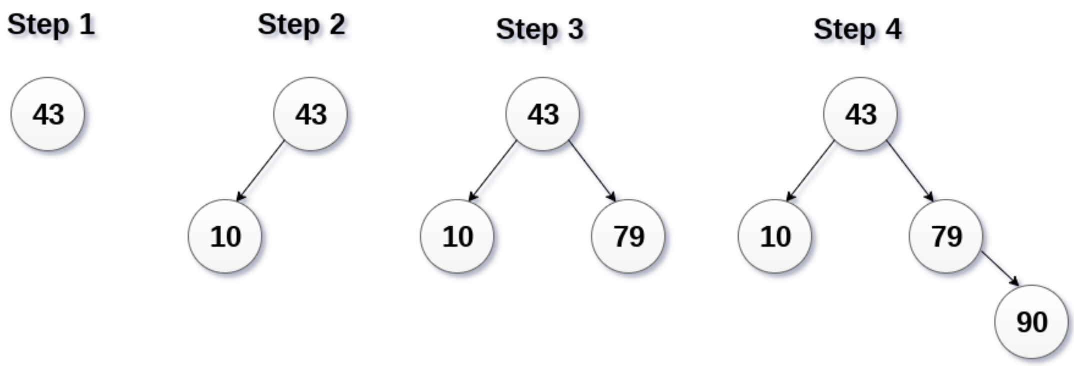
Advantages of using binary search tree

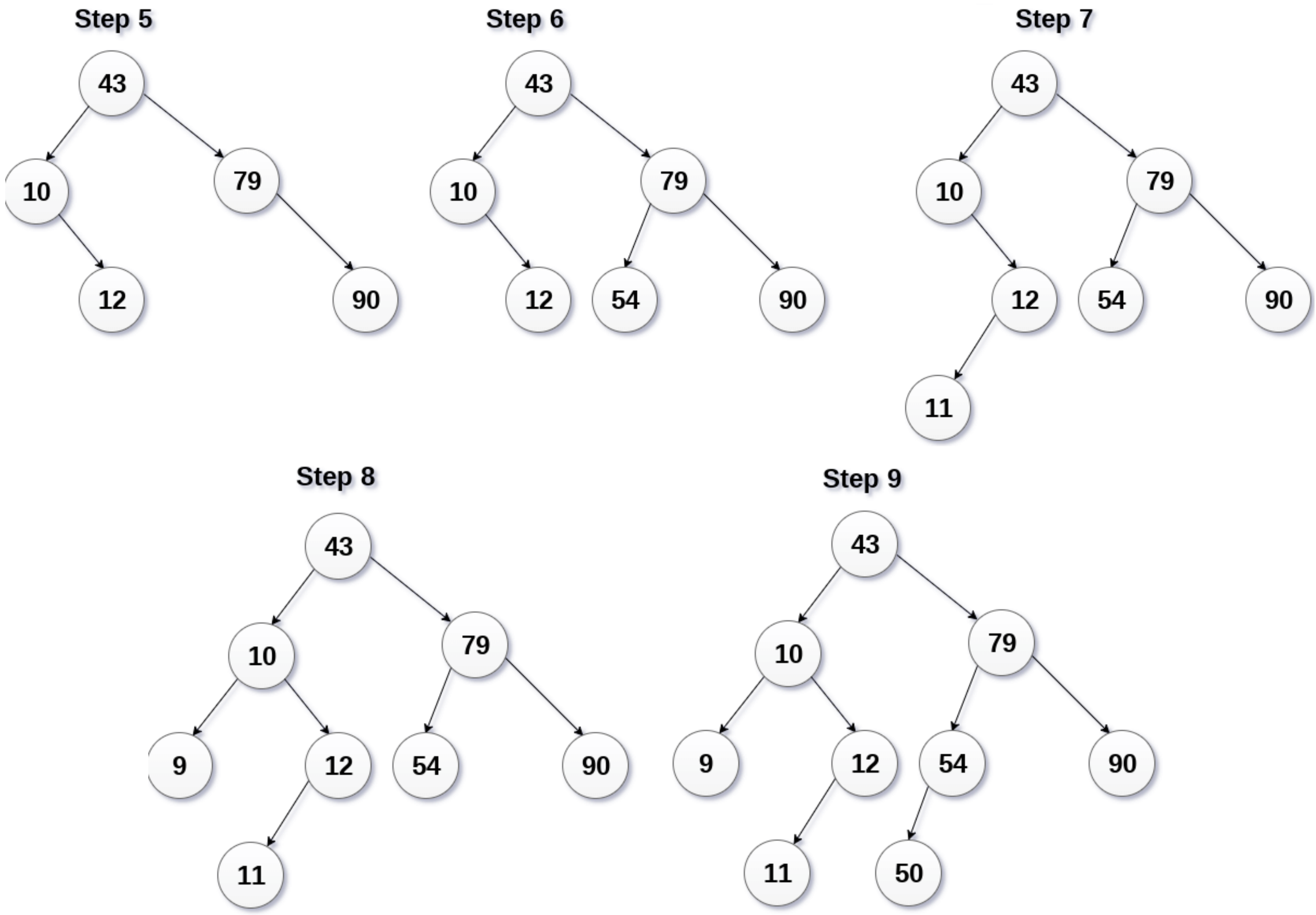
- ✓ Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
- ✓ The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n).
- ✓ It also speed up the insertion and deletion operations as compare to that in array and linked list.

Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

- ✓ Insert 43 into the tree as the root of the tree.
- ✓ Read the next element, if it is lesser than the root node element insert it as the root of the left sub-tree.
- ✓ Otherwise, insert it as the root of the right of the right sub-tree.





Operations on Binary Search Tree

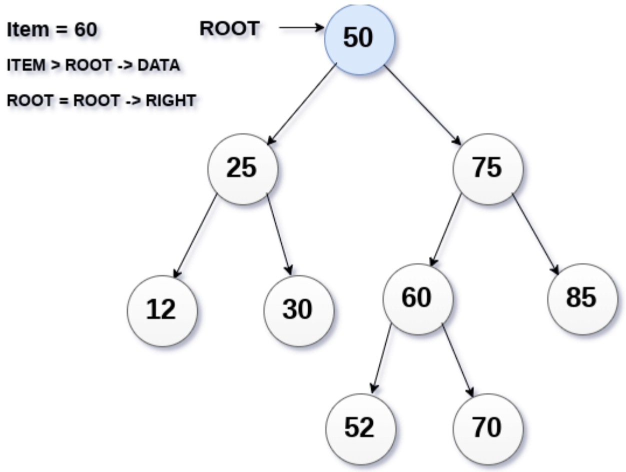
Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST is stored in a particular order.

- ✓ Compare the element with the root of the tree.
- ✓ If the item is matched then return the location of the node.
- ✓ Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
- ✓ If not, then move to the right sub-tree.
- ✓ Repeat this procedure recursively until match found.
- ✓ If element is not found then return NULL.

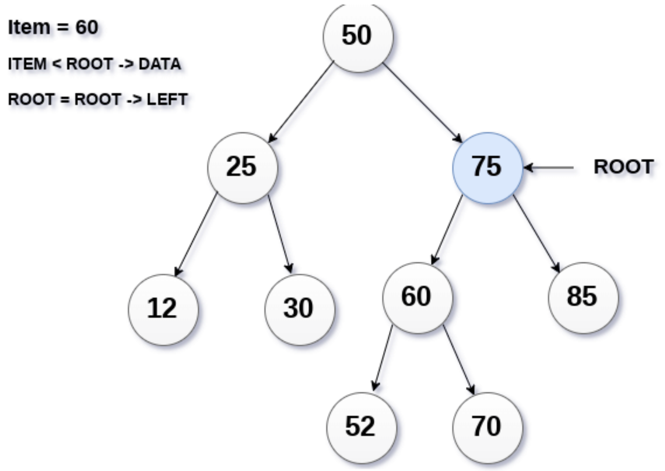
Algorithm:

```
Search (ROOT, ITEM)

Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL
    Return ROOT
   ELSE
   IF ROOT < ROOT -> DATA
   Return search (ROOT -> LEFT, ITEM)
  ELSE
   Return search (ROOT -> RIGHT, ITEM)
  [END OF IF]
  [END OF IF]
Step 2: END
```

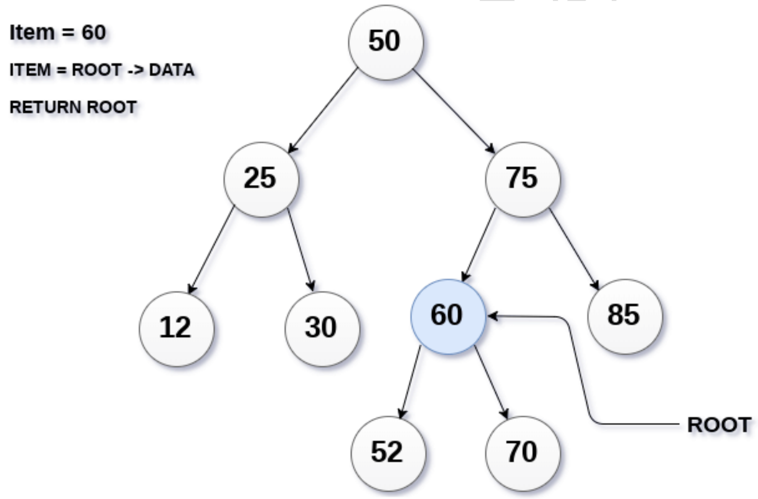
Example



STEP 1



STEP 2



STEP 3

Insertion

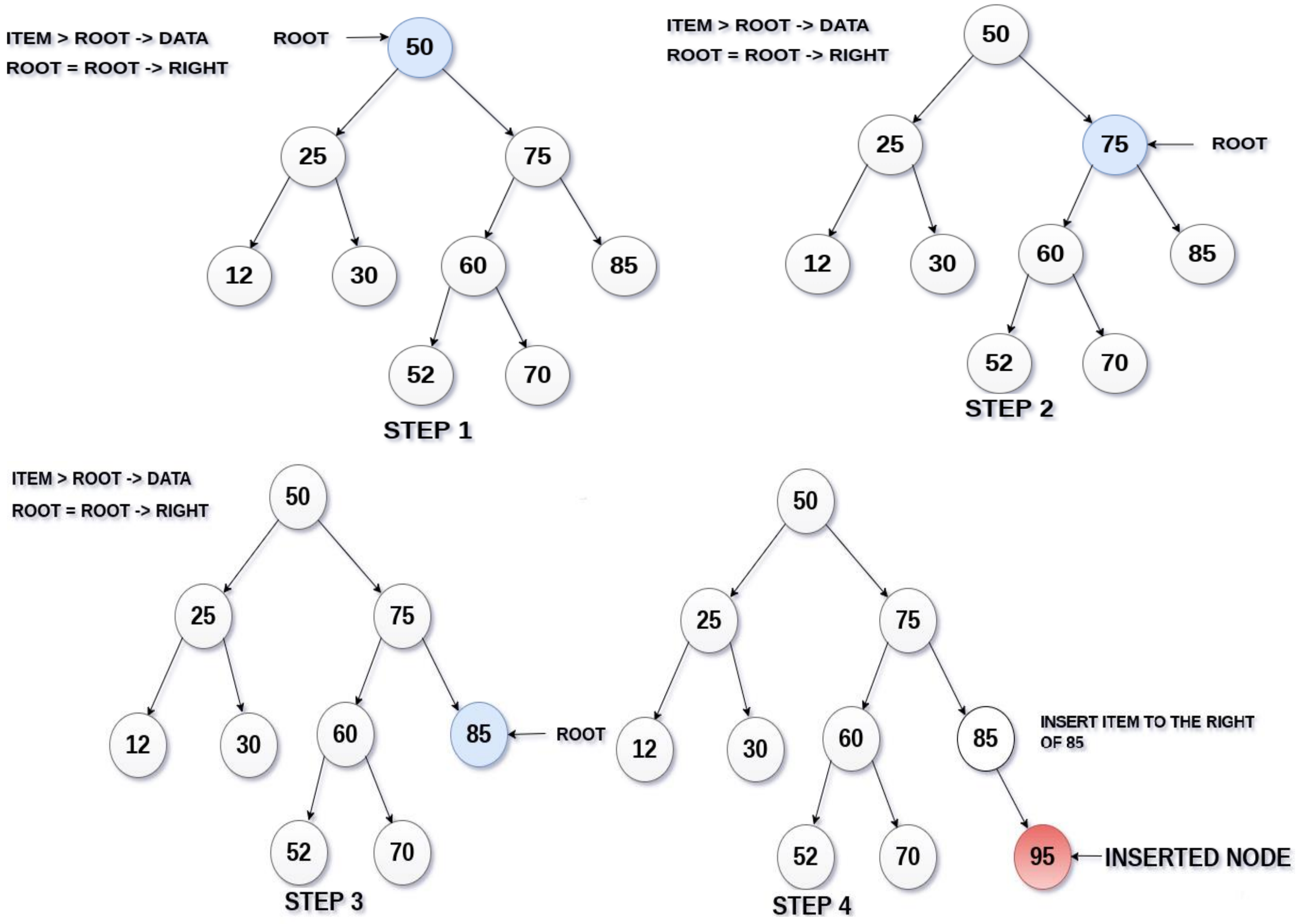
Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value**.**

- ✓ Allocate the memory for tree.
- ✓ Set the data part to the value and set the left and right pointer of tree, point to NULL.
- ✓ If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
- ✓ Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
- ✓ If this is false, then perform this operation recursively with the right sub-tree of the root.

Algorithm for Insert (TREE, ITEM)

```
        Step 1: IF TREE = NULL
                        Allocate memory for TREE
                        SET TREE -> DATA = ITEM
                       SET TREE -> LEFT = TREE -> RIGHT = NULL
                       ELSE
                        IF ITEM < TREE -> DATA
                       Insert (TREE -> LEFT, ITEM)
                        ELSE
                       Insert (TREE -> RIGHT, ITEM)
                        [END OF IF]
                       [END OF IF]
Step 2: END
```

ITEM=95



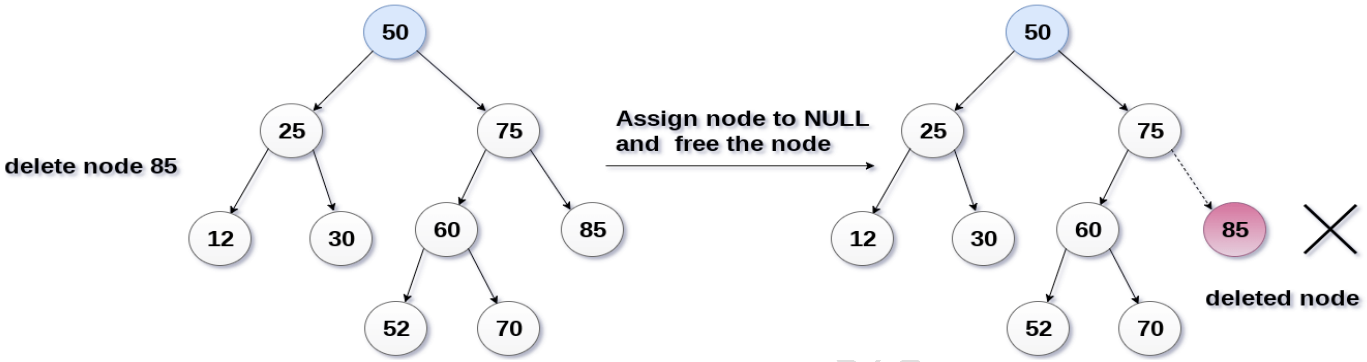
Deletion

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

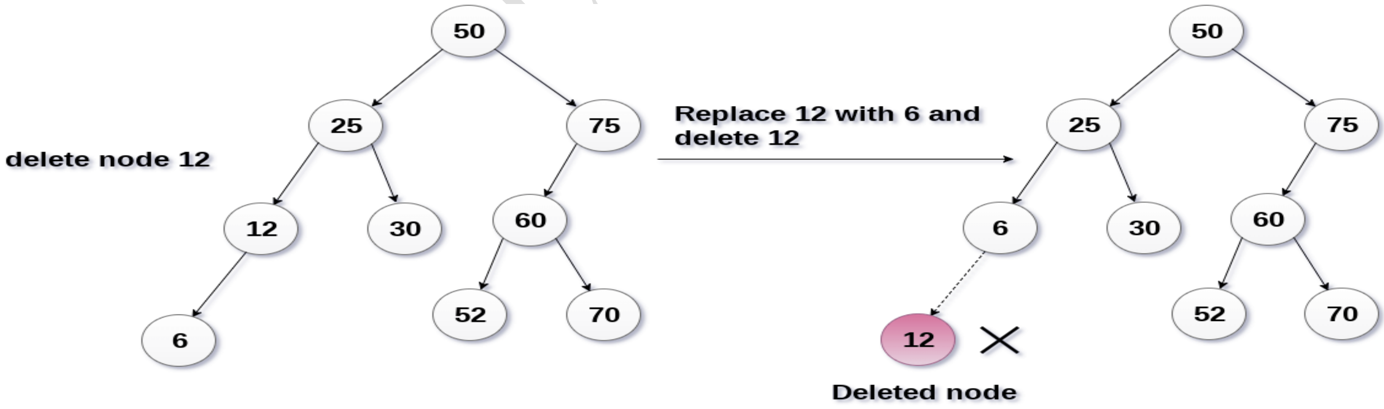
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following Example, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



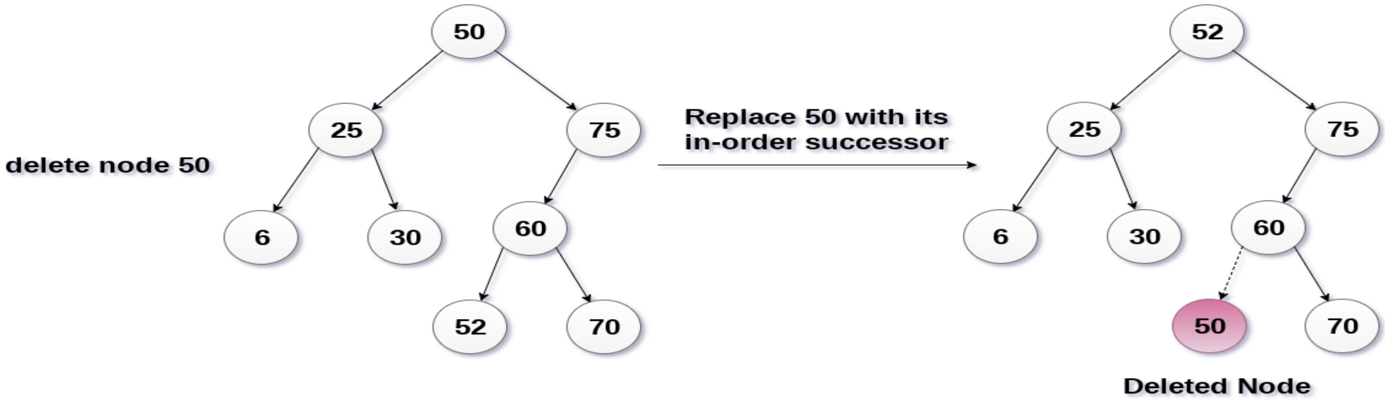
The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following Example, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

Replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Algorithm for Delete (TREE, ITEM)

```
Step 1: IF TREE = NULL
             Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
              Delete (TREE->LEFT, ITEM)
            ELSE IF ITEM > TREE -> DATA
            Delete (TREE -> RIGHT, ITEM)
            ELSE IF TREE -> LEFT AND TREE -> RIGHT
            SET TEMP = find Largest Node (TREE -> LEFT)
            SET TREE -> DATA = TEMP -> DATA
            Delete (TREE -> LEFT, TEMP -> DATA)
            ELSE
            SET TEMP = TREE
            IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            SET TREE = NULL
            ELSE IF TREE -> LEFT! = NULL
             SET TREE = TREE -> LEFT
             ELSE
            SET TREE = TREE -> RIGHT
            [END OF IF]
              FREE TEMP
            [END OF IF]
Step 2: END
```

Counting Binary Trees

The recursive structure of a binary tree makes it easy to count nodes recursively. There are 3 things we can count:

✓ The total number of nodes

- ✓ The number of leaf nodes
- ✓ The number of internal nodes

Counting all nodes

The number of nodes in a binary tree is the number of nodes in the root’s left sub tree, plus the number of nodes in its right sub tree, plus one (for the root itself).

Counting leaf nodes

This is similar, except that we only return 1 if we are a leaf node. Otherwise, we recursive into the left and right sub trees and return the sum of the leaves in them.

Counting internal nodes

This is the counterpart of counting leaves. If we are an internal node, we count 1 for ourselves, then recursive into the left and right sub trees and sum the count of internal nodes in them.

HASH TABLES

- ➢ Hash table is a data structure used for storing and retrieving data quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associate with some key. For example, for an employee record in the hash table employee ID will works as a key.
- ➢ Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is dependent upon the size of the hash table.
- ➢ The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.
- ➢ **Buck and Home bucket**
- ➢ The hash function H(key) is used to map a several dictionary entries in the hash table. Each function of hash table is called bucket. The function H(k) is home bucket for the dictionary with pail whose value is key.

	5
15	4
	3
10	2
	1
	0

Hash table

In the above diagram or hash table location 2 or 4 is called as home bucket and location 0,1,3,5 are called as bucket.

Static and Dynamic hashing

There are two types of hashing. They are:

1. Static hashing
2. Dynamic hashing

Static hashing

Static hashing is a hashing technique in which keys are stored in which keys are stored in hash table with fixed size.

Dynamic hashing

In this hashing table, the hash function is modified dynamically number of records grow.

Hash function

Hash function is a function which is used to put data into hash table. Hence one can use the same as function to retrieve the data from hash table. Thus, hash function is used implement a hash table.

There are several types of hash function.

1. Division hash function method
2. Mid square hash function method
3. Multiplication or multiplicative hash function
4. Digit folding or folding hash function

Division hash function method

The hash function depends upon the remainder of the division. Typically the division is the table length.

Syntax or Formula

H (key) = K % table size

Example:-

Insert following values or records

54, 72, 89, 37 into hash table. The hash table size is 10.

The following determines hash table with size 10.

	9
	8
	7
	6
	5
	4
	3
	2
	1
	0

The record 54 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key) =54%10=4

The record 54 is inserted at 4th location.

The record 72 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key)=72%10=2

The record 72 is inserted at 2nd location.

The record 89 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key) =89%10=9

The record 89 is inserted at 9th position or location.

The record 37 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key) =37%10=7

The record 37 is inserted at 7th position.

The following hash table determines the inserting records 54, 72, 89, 37 into hash table.

89
37
54
72

Mid square hash function

In the mid square method, the key is squared and the middle or mid part of the result is used as index or position or location.

Example the records 311, 3112, 3114 are inserted to hash table. Assume that hash table size is 1000.

Syntax or formula

$H(Key) = K^2$

The record 3111 by using mid square.

$H(key) =K^2$

$= (3111)^2$

$=9678321$

783 is the middle part of 9678321. So, 783 is the index of 3111.

The record 3112 by using mid square

$H (Key) = (3112)^2$

$= 9684544$

845 is the middle part of 9684544. So, 845 is the index of 3112.

The record 3113 by using mid square

$H (Key) = (3113)^2$

$= 9690769$

907 is the middle part of 9690769. So, 907 is the index of 3113.

3111	783
3112	845
3113	907
	999

Multiplicative hash function

The given record is multiplied by some constant value. The formula computing hash key is

H (Key) = floor (P*(fractional part of key*A))

Where ‘P’ is an integer constant and ‘A’ is real constant.

Donald Knuth suggested to use constant A = 0.61803398987.

Example:-

Insert the following records 107, 108, 109, 110 into hash table . Here P =50.

107 inserted into hash table by using multiplicative hash function.

H (Key) = floor (P*(fractional part of key*A))
= floor (50*(107* 0.61803398987)
= floor (3306.4818)
=3306

108 inserted into hash table by using multiplicative hash function.

H (Key) = floor (50*(108* 0.61803398987)
= floor (3337.3835)
= 3337

109 inserted into hash table by using multiplicative hash function.

H (Key) = floor (50*(109* 0.61803398987)
= floor (3368.2852)
= 3368

110 inserted into hash table by using multiplicative hash function.

H (Key) = floor (50*(110* 0.61803398987)
= floor (3399.1869)
= 3399

The following diagram determines the 107, 108, 109, 110 values into hash table.

	0
107	3306
108	3337
109	3368
110	3399
	3999

Digit folding or folding hash function

The key value is divided into separate parts and using some simple operation this parts are combined to produce hash key.

Example:-

Consider the record 1, 2, 3, 6, 5, 4, 1, 2 then it is divided into separate parts 123, 654, 12 and this all are added together.

H (Key) = 123+ 654 + 12 +789

The record 123, 654, 12 will be placed at a location 789 in the hash table.

Collision Resolution Technique

If collision occurs then it should be handled by applying some techniques. Such techniques are called collision resolution technique.

The goal of collision resolution techniques is to minimize collisions. There are two methods of handling collisions.

1. Open hashing or Separate Chain hashing
2. Closed hashing or Open addressing

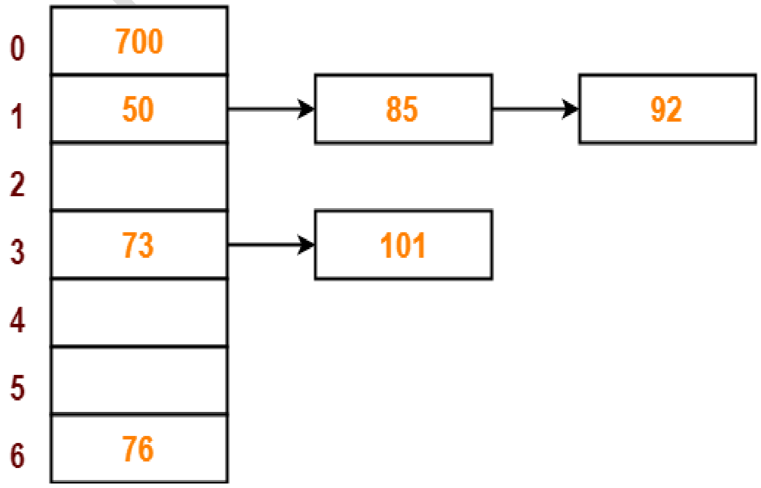
The difference between open hashing and closed hashing is that in Open hashing the collision are stored outside table and in Closed hashing the collisions are stored in the same table at some another slot.

Open hashing

In collision handling method chaining is a concept which introduces an additional fields with data i.e., chain. A separate chain table is maintained for colliding data when collision occurs then linked list is maintained at home bucket.

Example:-

Consider the keys to be placed in the in their home buckets are 50, 700, 76, 85, 92, 73 and 101



A chain is maintained for colliding elements. For distance 131 has a home bucket index 1. Similarly, keys 21 and 61 demand for home bucket index 1. Hence a chain is maintained at index 1. Similarly, the chain at index 4 and 7 is maintained.

Closed hashing

Closed hashing collision resolution strategy or technique which users following technique.

1. Linear probing
2. Quadratic probing
3. Double probing or Double hashing

Linear probing

This is the easiest method of handling collision. When collision occurs i.e., when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing the hash table is represented as a one-dimensional array with indices that range from 0 to desired table size-1.

Example: -

Consider that following keys are to be inserted in the hash table 131, 4, 8, 7, 21, 5, 31, 61, 9, 29.The hash table size is 10.

Initially we will put the following keys in the hash table 131, 4, 8, 7.

We will use division hash function. That means that keys are placed using formula.

H (Key) = key % table size

For instance, the element 131 can be placed at H (Key) = 131 % 10 =1.

Index 1 will be the home bucket for 131. Continuing in the fashion we will place 4,8,7.

0	Null
1	131
2	Null
3	Null
4	4
5	Null
6	Null
7	7
8	8
9	Null

Now the next to be inserted is 21. According to hash function H (Key) = 21 % 10 =1.

But the index 1 location already occupied with 131 i.e., collision occurs. To resolve this collision we will linearly move down from 1 to empty location is found. Therefore 21 will be placed at index 2. If the next element is 5 then we get home bucket for 5 as index 5 this bucket is empty so, we will put the element 5 at index 5.

0	Null
1	131
2	21
3	Null

4	4
5	5
6	Null
7	7
8	8
9	Null

After placing record keys 31, 61 the hash table will be

0	Null
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	Null

The next record key that comes is 9. According to decision as function it demands for the home bucket 9. Hence we will place 9 at index 9.

0	Null
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

Now the next final record key is 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs to handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0th index.

0	29
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

Quadratic probing

Quadratic probing operates by taking original hash value and adding successive values of quadratic polynomial to the stating value.

This method uses following formula.

$H(Key) = (H\ (Key) + i^2)\ \%\ m$

Where ‘m’ can be table size or any prime number.

Example: - If we have insert following elements in the hash table with table size 10.

37, 19, 55, 22, 17, 49, 87.

Initially we will put following keys into hash table.

37,19,55,22.

0	
1	
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now, if you want to place 17 a collision will be occurs 17. 17 % 10 = 7, but bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

$$H\ (Key) = (H\ (Key) + i^2)\ \%\ m$$

Consider I =0

$H\ (key) = (17+0^2)\ \%\ 10 = 17\ \%\ 10 = 7.$

Then i=1

$H\ (Key) = (17 + 1^2)\ \%\ 10 =18\ \%\ 10 = 8.$

The bucket 8 is empty. Hence, we will place the element of the index 8.

Now if you want to place 49 a collision will be occurred 49 % 10 = 9 and bucket 9 as already occupied with 19. Hence we will applying quadratic probing to insert this record in the hash table.

$H_i\ (Key) = (H\ (Key) + i^2)\ \%\ m$
I =0
$= (49 + 0)\ \%\ 10 = 49\ \%\ 10 = 9$
I = 1
$= (49 + 1^2)\ \%\ 10 = 50\ \%\ 10 = 0$

0	49
1	
2	22
3	
4	
5	55
6	
7	37
8	17
9	19

The bucket 0 is empty.

Hence the value 49 is inserted at a 0th position.

Now to place 87 we will use quadratic probing.

H(Key) = $(87 + 0^2)$ % 10 = 87 % 10 = 7

H(Key) = $(87 + 1^2)$ % 10 = 88 % 10 = 8

H(Key) = $(87 + 2^2)$ % 10 = 91 % 10 = 1

0	49
1	87
2	22
3	
4	
5	55
6	
7	37
8	17
9	19

Double probing or double hashing

Double hashing is a technique in which a second hash function is applied to key when a collision occurs by applying the second has function we will get number of positions from the point of Collision inserted.

By using following formulas, we can find out the double hashing.

H_1 (key) = k % table size

H_2(key) = M - (K % M)

Where M is prime number smaller than the size of the table.

Example: consider the following elements to be placed in the Hash table of size 10.

37, 90, 45, 22, 17, 49, 55.

Inside Initially the elements using the formula for H_1 (key). Insert 37, 90, 45, 22.

37 % 10 = 7

90 % 10 = 0

45 % 10 = 5

22 % 10 = 2

0	90
1	
2	22
3	
4	
5	45
6	
7	37
8	
9	

Now if 17 is to be inserted then

H_1 (17) = 17 % 10 = 7

Here collision will be occur because 7th position already occupied with element 37 or record 37. So we can apply second hash function to key.

$$H_2 \text{(key)} = M-(K\%M)$$

Here M is prime number smaller than the size of the table.

Let us prime number is M = 7

H_2 (17) =7-(17%7)

= 7 - 3 = 4

That means we have to insert the elements on 10 at 4 places from value 37 or 7th position.

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	

17 will be placed at index 1.

Now to insert number 49 at location 9th position that is 49 % 10 = 9.

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	49

Now to insert number 55.

H_1 (55) = 55 % 10 = 5 that is collision will be occur. Because the location 5 already occupied with 45. So, we can apply second hash function.

$$H_2 (55) = 7 - (55 \% 7) = 7 - 6 = 1$$

That means we have to take one jump from index 5 to place 55.

0	90
1	17
2	22
3	
4	
5	45
6	55
7	37

8	
9	49

Rehashing

Rehashing is a technique in which table is resized that is the size of table is double by creating a new table. It is preferable if the total size of new table is a prime number. There are situation in which rehashing is required.

i) When the table size is completely full.

ii) With Quadratic probing when the table is filled half.

iii) When insertion fail due to over flow.

In such situations, we have to transfer entries from old table to new table.

Example:

Consider we have to insert the elements 37, 90, 55, 22, 17, 49 and 87 the table size is 10 and will use hash function.

H (key) = K % Table size

Initially insert following elements 37, 90, 55, 22

0	90
1	
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now you can insert 17 into hash table. Here collision will be occur. Because the 7th location already occupied with 37. So, by using linear probing the element 17 is insert at 8th position.

0	90
1	
2	22
3	
4	
5	55
6	
7	37
8	17
9	

Now this table is almost full. So, next element 87 is not inserted into hash table because the hash table is overflow. Hence we will rehashing by double the size for new table that becomes 20. But 20 is not prime number we will prefer to make table size as 23 and new hash function will be

H (key) = k % 23

37 % 23 = 14

90 % 23 = 21

55 % 23 = 9

22 % 23 = 22

17 % 23 = 17

49 % 23 = 3

87 % 23 = 18

0	
1	
2	
3	49
4	
5	
6	
7	
8	
9	55
10	
11	
12	
13	
14	37
15	
16	
17	17
18	87
19	
20	
21	90
22	22

Applications of hashing in unique identifier generation

- ✓ **Database indexing:** Hashing is used to index and retrieve data efficiently in databases and other data storage systems.
- ✓ **Password storage:** Hashing is used to store passwords securely by applying a hash function to the password and storing the hashed result, rather than the plain text password.
- ✓ **Data compression:** Hashing is used in data compression algorithms, such as the Huffman coding algorithm, to encode data efficiently.
- ✓ **Search algorithms:** Hashing is used to implement search algorithms, such as hash tables and bloom filters, for fast lookups and queries.
- ✓ **Cryptography:** Hashing is used in cryptography to generate digital signatures, message authentication codes (MACs), and key derivation functions.

- ✓ **Load balancing:** Hashing is used in load-balancing algorithms, such as consistent hashing, to distribute requests to servers in a network.
- ✓ **Blockchain:** Hashing is used in blockchain technology, such as the proof-of-work algorithm, to secure the integrity and consensus of the blockchain.
- ✓ **Image processing:** Hashing is used in image processing applications, such as perceptual hashing, to detect and prevent image duplicates and modifications.
- ✓ **File comparison:** Hashing is used in file comparison algorithms, such as the MD5 and SHA-1 hash functions, to compare and verify the integrity of files.
- ✓ **Fraud detection:** Hashing is used in fraud detection and cybersecurity applications, such as intrusion detection and antivirus software, to detect and prevent malicious activities.

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc.

There are many other applications of hashing, including modern-day cryptography hash functions. Some of these applications are listed below:

- ➢ Message Digest
- ➢ Password Verification
- ➢ Data Structures(Programming Languages)
- ➢ Compiler Operation
- ➢ Rabin-Karp Algorithm
- ➢ Linking File name and path together
- ➢ Game Boards
- ➢ Graphics

Data Structures(Programming Languages):

Various programming languages have hash table based Data Structures. The basic idea is to create a key-value pair where key is supposed to be a unique value, whereas value can be same for different keys. This implementation is seen in unordered_set & unordered_map in C++, HashSet & HashMap in java, dict in python etc.

Advantages of Applications of Hashing

- ➢ **Efficiency:** Hashing allows for fast lookups, searches, and retrievals of data, with an average time complexity of O(1) for hash table lookups.
- ➢ **Dynamic:** Hashing is a dynamic data structure that can be easily resized, making it suitable for growing and changing datasets.

- **Secure:** Hashing provides a secure method for storing and retrieving sensitive information, such as passwords, as the original data is transformed into a hash value that is difficult to reverse.
- **Simple:** Hashing is a simple and straightforward concept, making it easy to implement and understand.
- **Scalable:** Hashing can be scaled to handle large amounts of data, making it suitable for big data applications.
- **Uniqueness:** Hashing ensures the uniqueness of data, as two different inputs will result in two different hash values, avoiding collisions.
- **Verification:** Hashing can be used for data verification, such as checking the integrity of files, as even a small change in the input data will result in a different hash value.
- **Space-efficient:** Hashing is a space-efficient method for storing and retrieving data, as it only stores the hash values, reducing the amount of memory required.
- **Error detection:** Hashing can be used for error detection, as it can detect errors in data transmission, storage, or processing.
- **Speed:** Hashing is a fast and efficient method for processing data, making it suitable for real-time and high-performance applications.

****