UNIT - I

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks-Operations, array and linked representations of stacks, stack applications, Queues-operations, array and linked representations.

Data Structure

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 a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Basic Terminology

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

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

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

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

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

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

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

Need of Data Structures

As applications are getting 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.

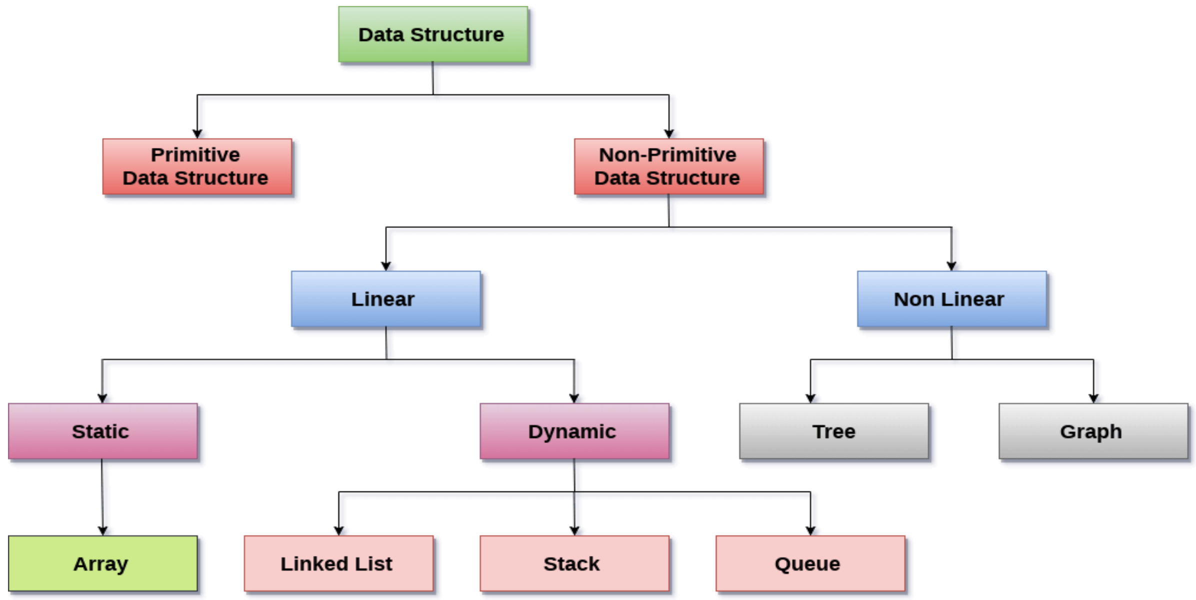
Advantages of Data Structures

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.

Linear Data Structures

If a data structure organizes the data in sequential order, then that data structure is called a Linear DataStructure.

Example

1. Arrays
2. List (Linked List)
3. Stack
4. Queue

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.
The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.
The individual elements of the array age are:
age[0], age[1], age[2], age[3],......... age[98], age[99].
Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.
A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.
It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

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

Non - Linear Data Structures

If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure.

Example

1. Tree
2. Graph
3. Dictionaries
4. Heaps
5. Tries, Etc.,

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.
Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

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

Operations on data structure

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

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

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.
If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.
If we try to delete an element from an empty data structure then **underflow** occurs.

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

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

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

Abstract Data Type:

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called information hiding. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

Stack

A Stack is linear data structure. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the **top of the stack**. Stack principle is **LIFO (last in, first out)**. Which element inserted last on to the stack that element deleted first from the stack.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

Real life examples of stacks are:

Stack of books

Stack of Plates

Stack of Toys

Operations on stack:

The two basic operations associated with stacks are:

1. Push
2. Pop

While performing push and pop operations the following test must be conducted on the stack.

a) Stack is empty or not b) stack is full or not

1. **Push:** Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

2. **Pop:** Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack (or) Implementation of stack:
The stack should be represented in two ways:

1. Stack using array
2. Stack using linked list

1. Stack using array:
Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a ***stack overflow*** condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a ***stack underflow*** condition.

1.push(): When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().

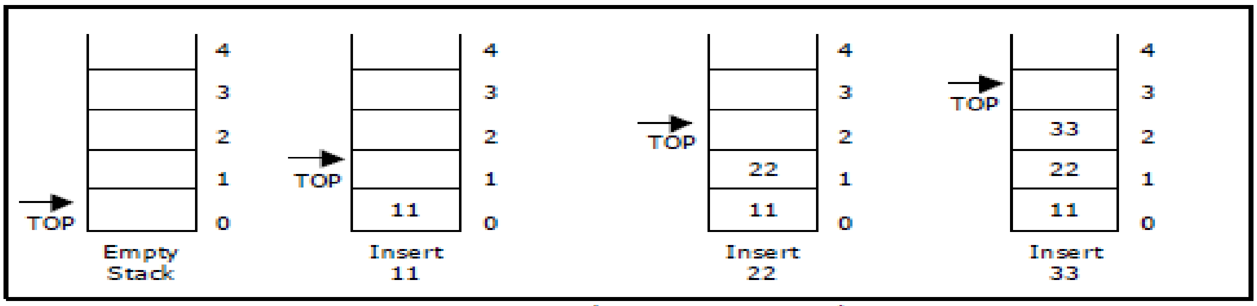


Figure . Push operations on stack

Initially **top=-1**, we can insert an element in to the stack, increment the top value i.e **top=top+1**. We can insert an element in to the stack first check the condition is stack is full or not. i.e **top>=size-1**. Otherwise add the element in to the stack.

Algorithm: Procedure for push():

```
Step 1: START
Step 2: if top>=size-1 then
                Write " Stack is Overflow"
Step 3: Otherwise
        3.1: read data value 'x'
        3.2: top=top+1;
        3.3: stack[top]=x;
Step 4: END
```

2.Pop(): When an element is taken off from the stack, the operation is performed by pop(). Below

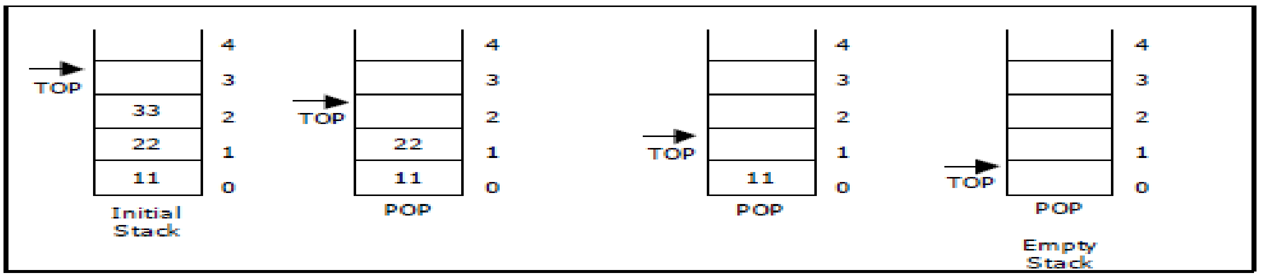


Figure Pop operations on stack

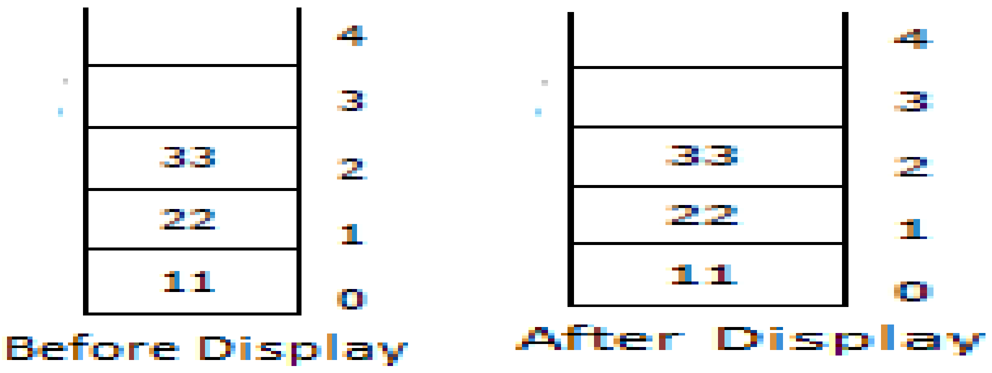
figure shows a stack initially with three elements and shows the deletion of elements using pop().

We can insert an element from the stack, decrement the top value i.e **top=top-1**.
We can delete an element from the stack first check the condition is stack is empty or not.
i.e **top==-1**. Otherwise remove the element from the stack.

Algorithm: procedure pop():

```
Step 1: START
Step 2: if top==-1 then
            Write "Stack is Underflow"
Step 3: otherwise
           3.1: print "deleted element"
            3.2: top=top-1;
Step 4: END
```

3.display(): This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.



Algorithm: procedure pop():

```
Step 1: START
Step 2: if top==-1 then
            Write "Stack is Underflow"
Step 3: otherwise
            3.1: print "Display elements are"
            3.2: for top to 0
            Print 'stack[i]'
Step 4: END
```

Stack Implementation Using Arrays

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

   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   printf("*********Stack operations using array\n*********");
   while(choice != 4)
   {
      printf("Chose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("Exiting....");
            break;
         }
         default:
         {
            printf("Please Enter valid choice ");
         }
      };
   }
}
 void push ()
{
   int val;
   if (top == n )
   printf("\n Stack Overflow");
   else
   {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = val;
   }
}
 void pop ()
{
   if(top == -1)
   printf("Stack Underflow");
   else
   top = top -1;
}
void display()
{
  if(top == -1)
   {
      printf("Stack is empty");
   }
   printf("stack elements are\n ")
   for (i=top;i>=0;i--)
   {
      printf(" %d ",stack[i]);
   }

}
```

OUTPUT:

```
Enter the number of elements in the stack 5
*********Stack operations using array*********
--------------------------------------------
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
5
```

```
Please Enter valid choice Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
stack elements are
 12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
stack elements are
 12
12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
4
Exiting....
```

Applications of STACK:

Application of Stack :

- Recursive Function.
- Expression Evaluation.
- Expression Conversion.
    - ➢ Infix to postfix
    - ➢ Infix to prefix
    - ➢ Postfix to infix
    - ➢ Postfix to prefix
    - ➢ Prefix to infix
    - ➢ Prefix to postfix
- Reverse a Data
- Processing Function Calls

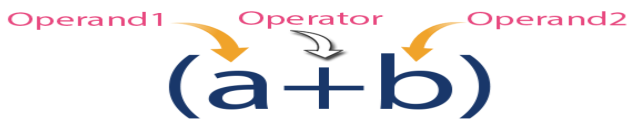
Expressions:

- An expression is a collection of operators and operands that represents a specific value.
- Operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,
- Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location

Expression types:

Based on the operator position, expressions are divided into THREE types. They are as follows.

- **Infix Expression**
    - In infix expression, operator is used in between operands.
    - Syntax : operand1 operator operand2
    - Example



- **Postfix Expression**

- In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".
- Syntax : operand1 operand2 operator
- Example:

Operand1
Operand2
Operator
ab+

- **Prefix Expression**
    - In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".
    - Syntax : operator operand1 operand2
    - Example:

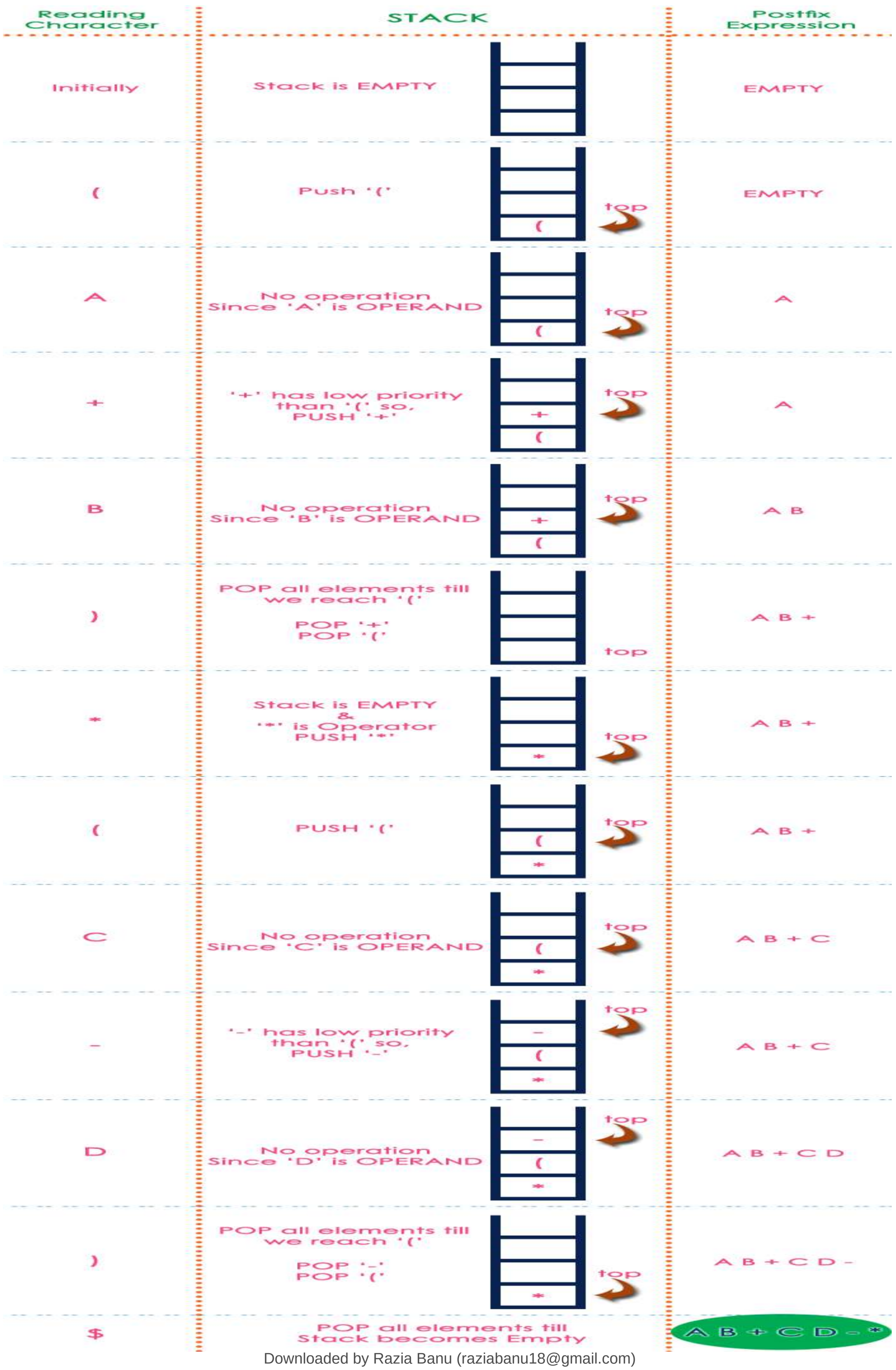
Operator
Operand1
Operand2
+ab

<u>Infix to postfix conversion using stack:</u>

- Procedure to convert from infix expression to postfix expression is as follows:
- Scan the infix expression from left to right.
- If the scanned symbol is left parenthesis, push it onto the stack.
- If the scanned symbol is an operand, then place directly in the postfix expression (output).
- If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
- If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example-1

[Type text]



Reading Character	STACK		Postfix Expression
Initially	Stack is EMPTY		EMPTY
(	Push '('	(top	EMPTY
A	No operation Since 'A' is OPERAND	(top	A
+	'+' has low priority than '(' so, PUSH '+'	+ top (	A
B	No operation Since 'B' is OPERAND	+ top (	A B
)	POP all elements till we reach '(' POP '+' POP '('	top	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*'	* top	A B +
(	PUSH '('	(top *	A B +
C	No operation Since 'C' is OPERAND	(top *	A B + C
–	'–' has low priority than '(' so, PUSH '–'	– top (*	A B + C
D	No operation Since 'D' is OPERAND	– top (*	A B + C D
)	POP all elements till we reach '(' POP '–' POP '('	* top	A B + C D –
$	POP all elements till Stack becomes Empty		A B + C D – *

Example2:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	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 +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example3

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
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	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	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 /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A+(B *C–(D/E↑F)*G)*H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
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	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Evaluation of postfix expression:

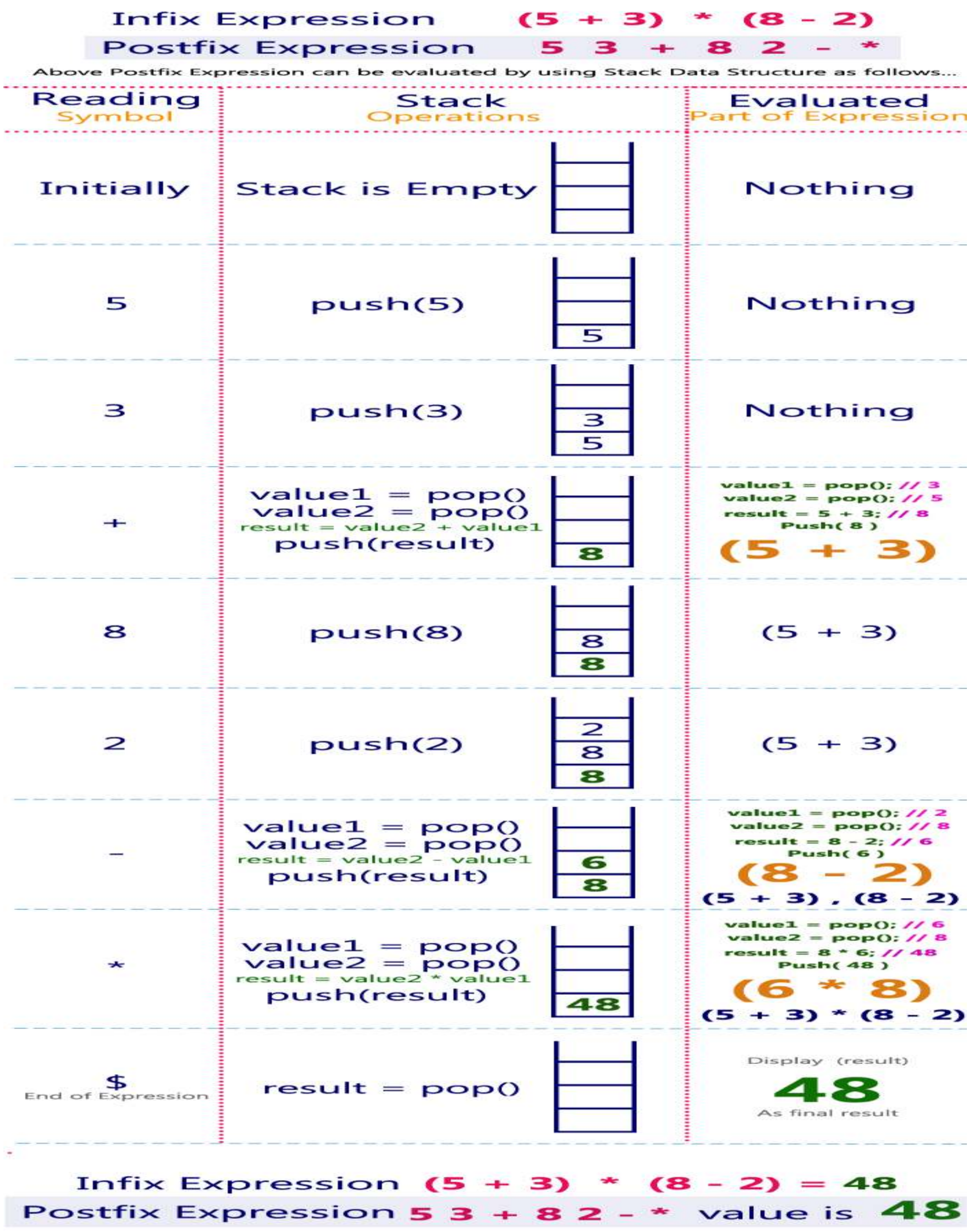
- The postfix expression is evaluated easily by the use of a stack.
- When a number is seen, it is pushed onto the stack;
- when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
- When an expression is given in postfix notation, there is no need to know any precedence rules.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example2



Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty	Nothing
5	push(5) 5	Nothing
3	push(3) 3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
	8
		value1 = pop(); // 3
		value2 = pop(); // 5
		result = 5 + 3; // 8
		Push(8)
		(5 + 3)

8	push(8) 8 8	(5 + 3)
2	push(2) 2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
	6
	8
		value1 = pop(); // 2
		value2 = pop(); // 8
		result = 8 - 2; // 6
		Push(6)
		(8 - 2)
		(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
	48
		value1 = pop(); // 6
		value2 = pop(); // 8
		result = 8 * 6; // 48
		Push(48)
		(6 * 8)
		(5 + 3) * (8 - 2)

$ End of Expression	result = pop()	Display (result) 48 As final result

Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

Example 3:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

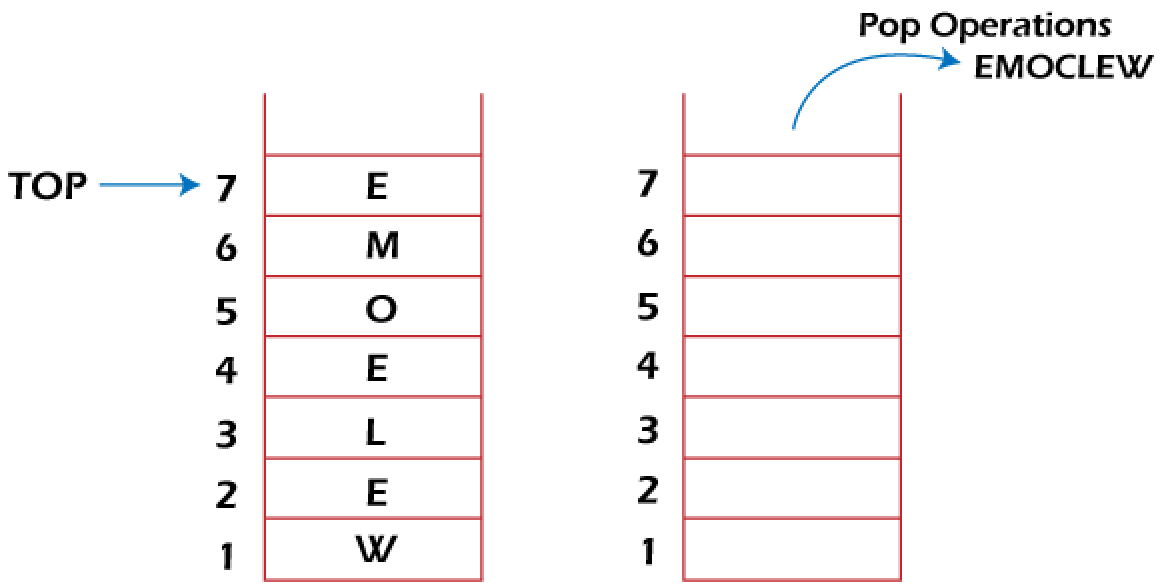
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

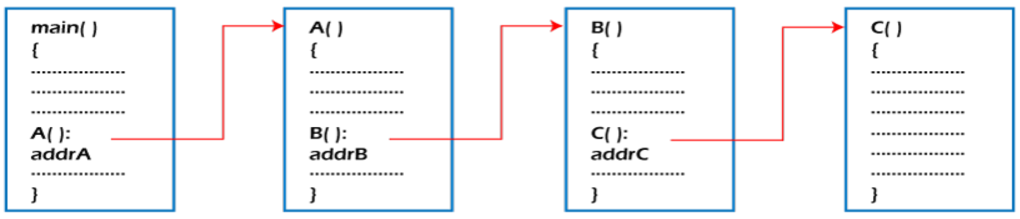
Reverse a String

A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Processing Function Calls:

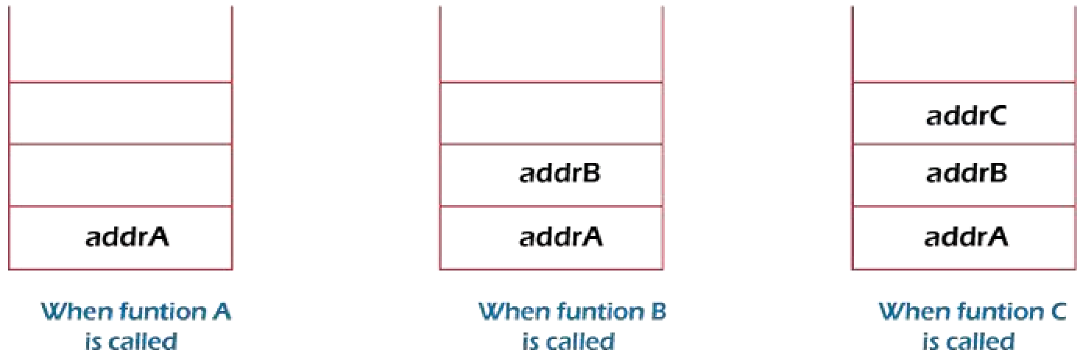
Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



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.

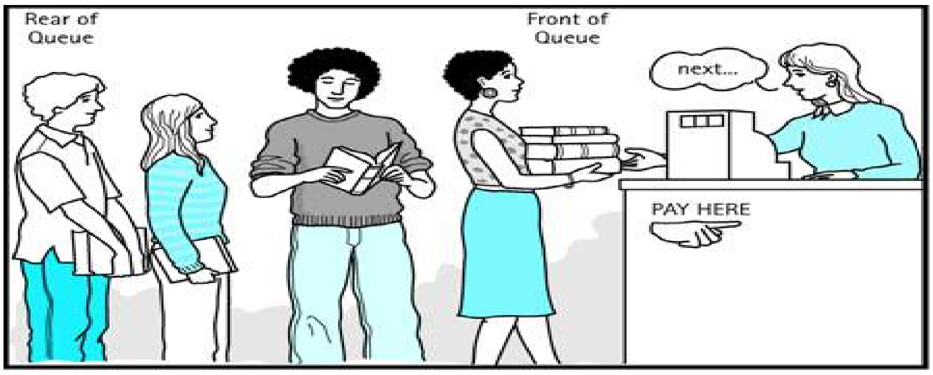
QUEUE

A queue is linear data structure and collection of elements. A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. The principle of queue is a "FIFO" or "First-in-first-out".

Queue is an abstract data structure. A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

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 and our college library.



The operations for a queue are analogues to those for a stack; the difference is that the insertions go at the end of the list, rather than the beginning.

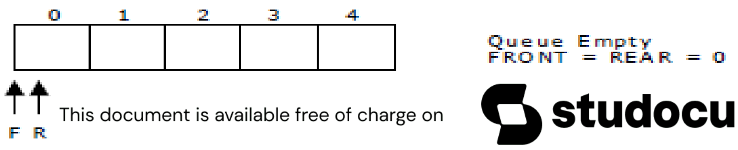
Operations on QUEUE:
A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

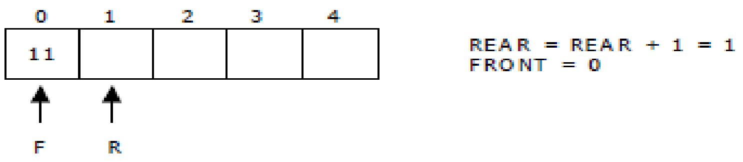
- **Enqueue or insertion**: which inserts an element at the end of the queue.
- **Dequeue or deletion**: which deletes an element at the start of the queue.

Representation of Queue (or) Implementation of Queue:
The queue can be represented in two ways:

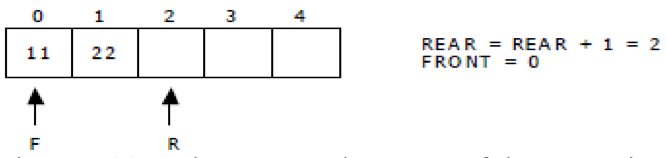
1. Queue using Array
2. Queue using Linked List

1.Queue using Array:
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.
Now, insert 11 to the queue. Then queue status will be:

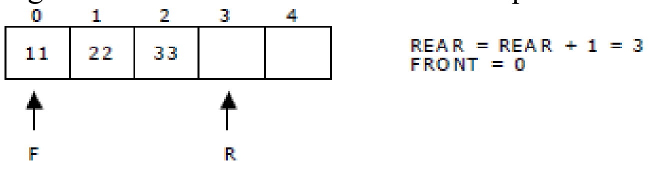




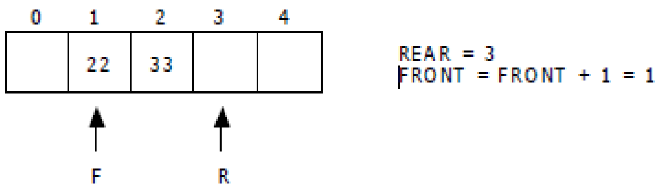
Next, insert 22 to the queue. Then the queue status is:



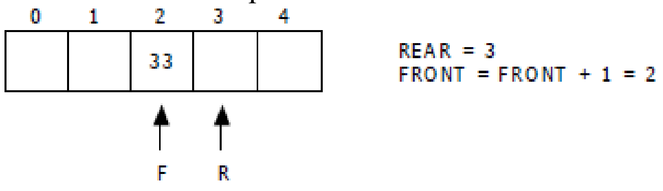
Again insert another element 33 to the queue. The status of the queue is:



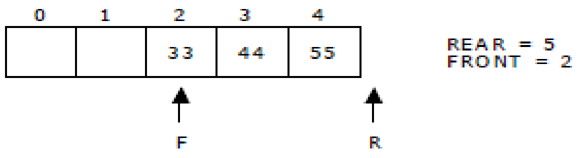
Now, delete an element. The element deleted is the element at the front of the queue.So the status of the queue is:



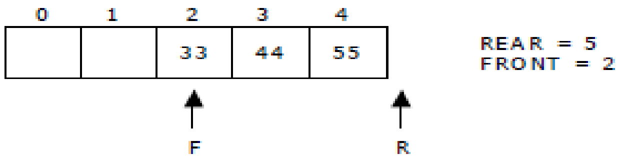
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



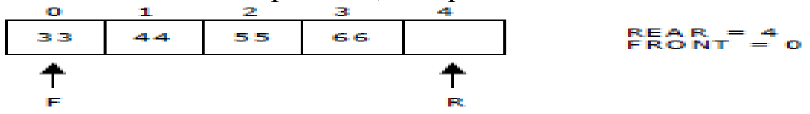
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Algorithm to insert any element in a queue

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

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

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

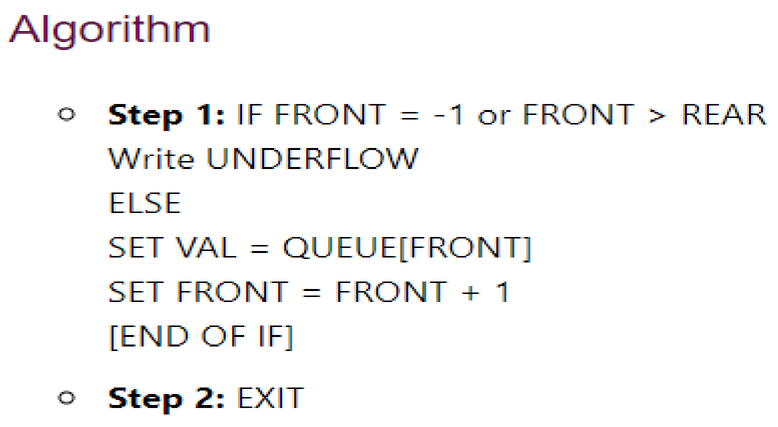
Algorithm

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

Algorithm to delete an element from the queue

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

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



Algorithm

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

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- **Step 1 -** Check whether **queue** is **EMPTY**.
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i**++). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

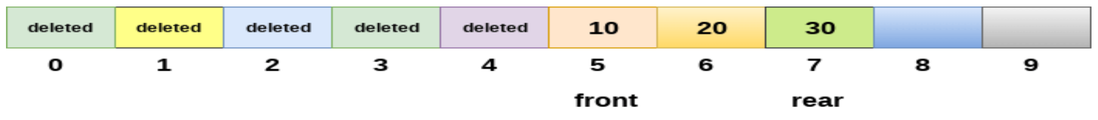
Queue Implementation using Arrays

```
#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 of Queue

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

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



limitation of array representation of queue

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

- **Deciding the array size**

One 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.

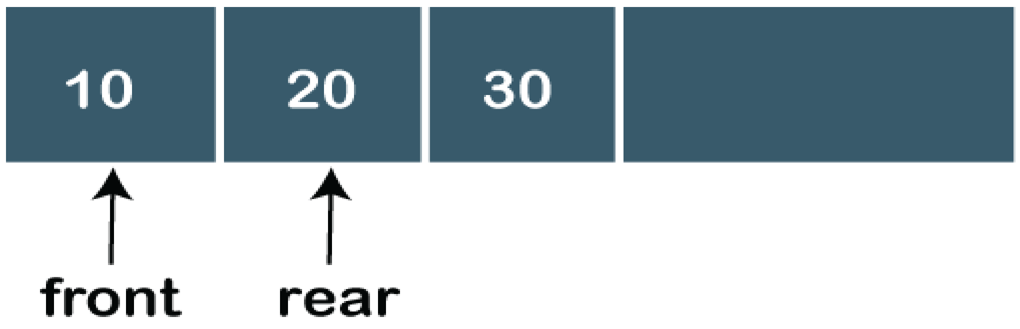
Types of Queues

There are four types of Queues**:**

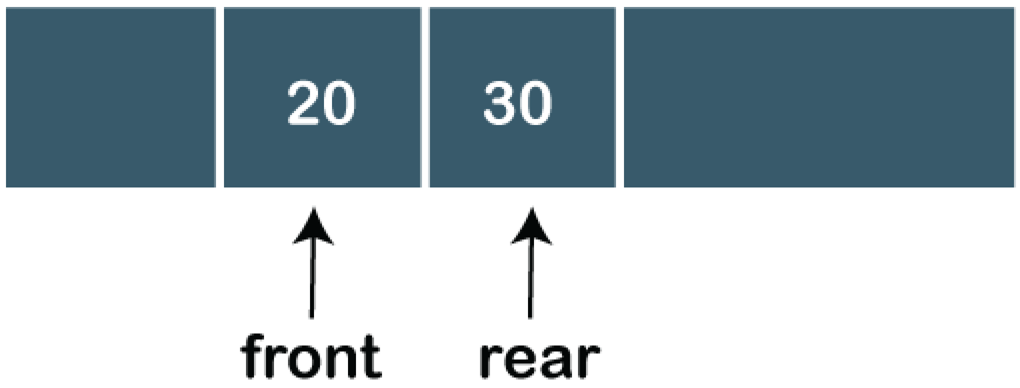
1. Linear Queue
2. Circular Queue
3. Priority Queue
4. Deque

1. **Linear Queue**

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



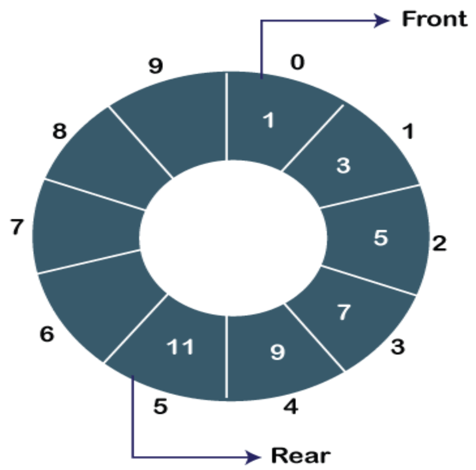
The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:



In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.
The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

2. **Circular Queue**

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



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

3. **Priority Queue**

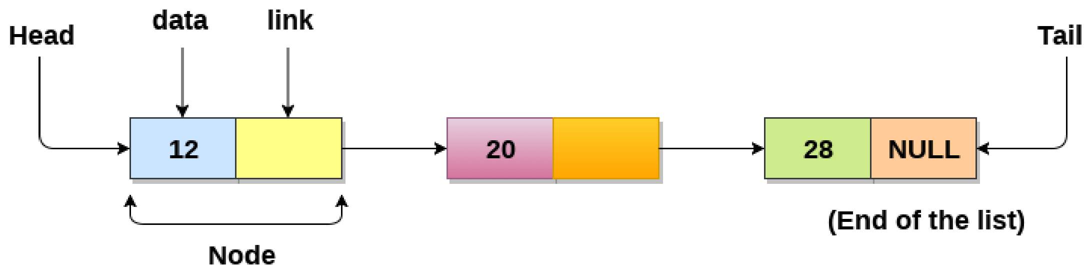
A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.
In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:
The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

4. **Deque**

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

Linked List

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



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can't be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

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

Array contains following limitations:

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

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

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

Differences between the array and linked list in a tabular form.

ARRAYS	LINKED LISTS
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address
Array elements store in a contiguous memory location	Linked list elements can be stored anywhere in the memory or randomly stored
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time	In the case of a linked list, memory is allocated at run time
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement

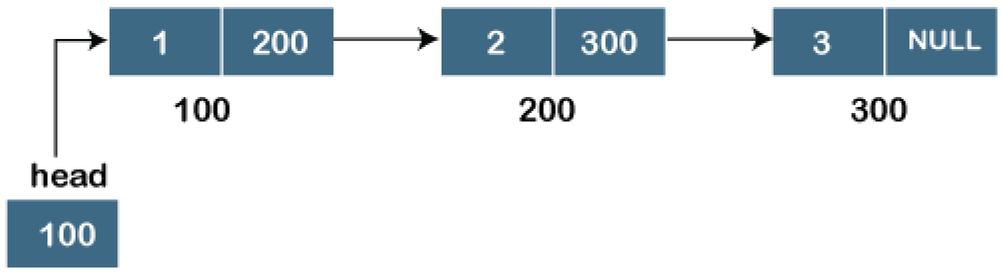
Types of Linked List

The following are the types of linked list:

1. **Singly linked list**
2. **Doubly linked list**
3. **Circular linked list**

Singly Linked list

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a pointer.
Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



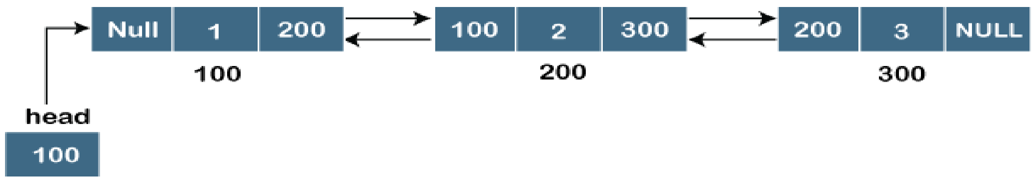
We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a ***head pointer***.
The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.
Representation of the node in a singly linked list

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

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

Doubly linked list

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.
Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below



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

Representation of the node in a doubly linked list

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

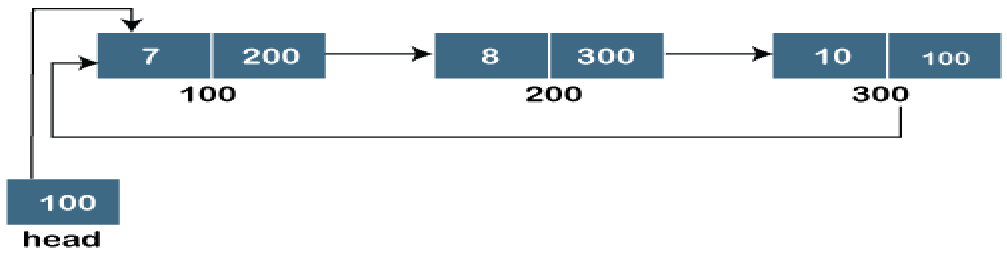
In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev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

Circular linked list

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

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

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

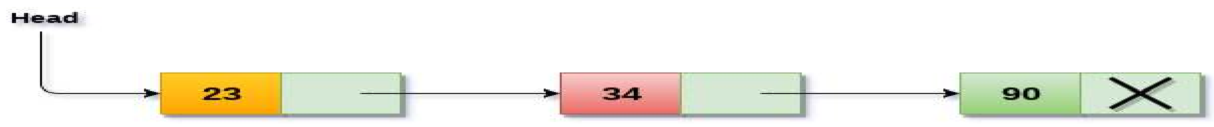


Singly linked list

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

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

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



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

Operations on Singly Linked List

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

Node Creation

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

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

1. Inserting at Beginning
2. Inserting at the End of the LIst
3. Inserting after specified node

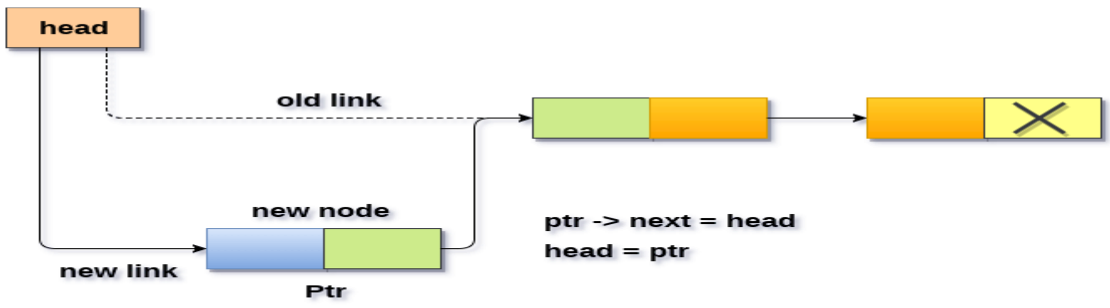
Insertion in singly linked list at beginning

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need to be followed in order to inser a new node in the list at beginning.

1. Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.
   ```
   ptr = (struct node *) malloc(sizeof(struct node *));
         ptr → data = item
   ```
2. Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.
   ```
   ptr->next = head
   ```

3. At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

```
head = ptr;
```



Algorithm

```
○ Step 1: IF PTR = NULL
      Write OVERFLOW
         Go to Step 7
        [END OF IF]
○ Step 2: SET NEW_NODE = PTR
○ Step 3: SET PTR = PTR → NEXT
○ Step 4: SET NEW_NODE → DATA = VAL
○ Step 5: SET NEW_NODE → NEXT = HEAD
○ Step 6: SET HEAD = NEW_NODE
○ Step 7: EXIT
```

Function for inserting element at beginning of the list

```c
void beginsert()
{
   struct node *ptr;
   int item;
   ptr = (struct node *) malloc(sizeof(struct node *));
   if(ptr == NULL)
   {
      printf("\n memory insufficient to allocate");
   }
   else
   {
      printf("\nEnter value\n");
      scanf("%d",&item);
      ptr->data = item;
      ptr->next = head;          head = ptr;
```

```
        printf("\nNode inserted");
      }
    }
```

2.Inserting at the End of the List

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list(CASE 1)
2. The node is being added to the end of the linked list(CASE2)

in the first case,(CASE1)

- The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node by using malloc statement in C. Data and the link part of the node are set up by using the following statements.

  ```
  ptr->data = item;
              ptr -> next = NULL;
  ```
- Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list. This will be done by using the following Statements.

  Head = ptr

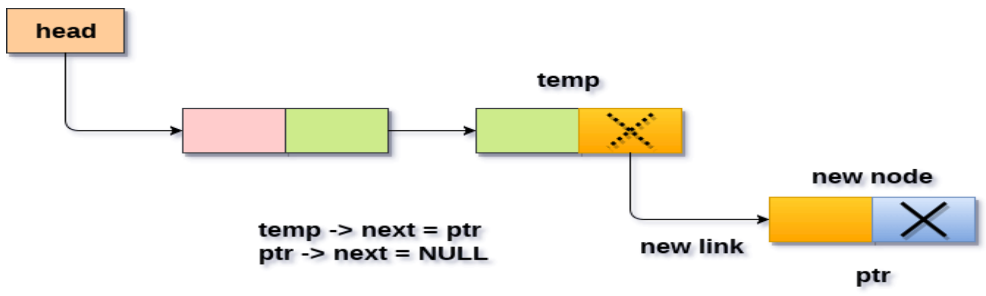
In the second case: CASE(2):

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.

  Temp = head
- Then, traverse through the entire linked list using the statements:

  ```
  while (temp→ next != NULL)
           temp = temp → next;
  ```
- At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**. We need to make the next part o
- If the temp node (which is currently the last node of the list) point to the new node (ptr) .temp = head;

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



Inserting node at the last into a non-empty list

Algorithm

```
Step 1: IF PTR = NULL Write OVERFLOW
    Go to Step 1
   [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR - > NEXT
Step 4: SET NEW_NODE - > DATA = VAL
Step 5: SET NEW_NODE - > NEXT = NULL
Step 6: SET PTR = HEAD
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
```

Function for inserting element at the end of the list

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

```
        {
          ptr -> next = NULL;
          head = ptr;
          printf("\nNode inserted");
        }
        else
        {
          temp = head;
          while (temp -> next != NULL)
          {
            temp = temp -> next;
          }
          temp->next = ptr;
          ptr->next = NULL;
          printf("\nNode inserted");

        }
    }
}
```

Insertion in singly linked list after specified Node

- In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted. This will be done by using the following statements.

```
emp=head;
        for(i=0;i<loc;i++)
        {
          temp = temp->next;
          if(temp == NULL)
          {
            return;
          }
        }
```

- Allocate the space for the new node and add the item to the data part of it. This will be done by using the following statements.

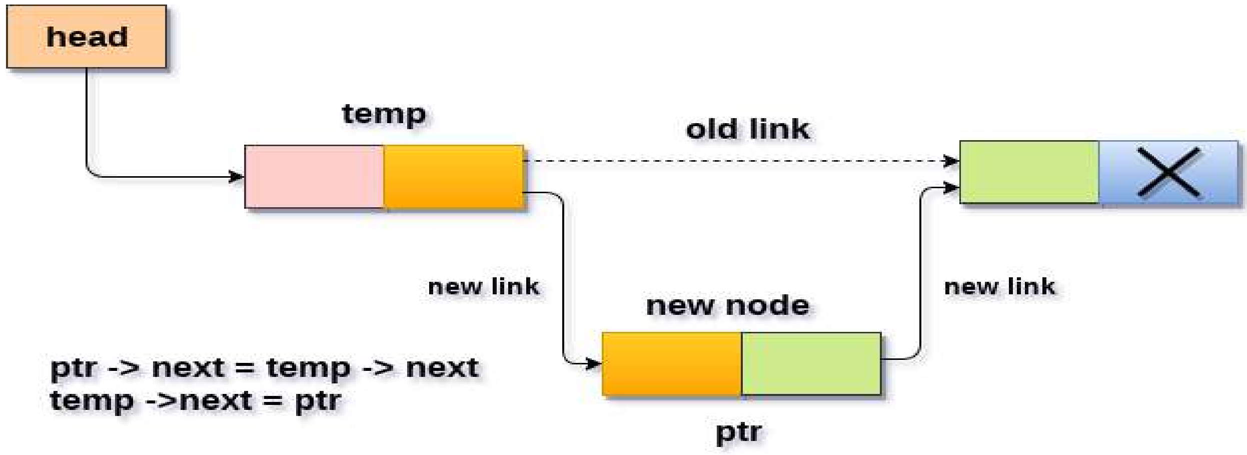
```
        ptr = (struct node *) malloc (sizeof(struct node));
        ptr->data = item;
```

- Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp). This will be done by using the following statements.

ptr→ next = temp → next

now, we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

temp ->next = ptr;



Algorithm

```
o   STEP 1: IF PTR = NULL
    WRITE OVERFLOW
        GOTO STEP 12
      END OF IF
o   STEP 2: SET NEW_NODE = PTR
o   STEP 3: NEW_NODE → DATA = VAL
o   STEP 4: SET TEMP = HEAD
o   STEP 5: SET I = 0
o   STEP 6: REPEAT STEP 5 AND 6 UNTIL I<loc< li=""></loc<>
o   STEP 7: TEMP = TEMP → NEXT
o   STEP 8: IF TEMP = NULL
    WRITE "DESIRED NODE NOT PRESENT"
         GOTO STEP 12
        END OF IF
     END OF LOOP
o   STEP 9: PTR → NEXT = TEMP → NEXT
o   STEP 10: TEMP → NEXT = PTR
o   STEP 11: SET PTR = NEW_NODE
o   STEP 12: EXIT
```

C Function

```
void randominsert()
{
   int i,loc,item;
```

```
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=1;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
```

Deletion

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

1. Deleting at Beginning
2. Deleting at the End of the List
3. Deleting after specified node

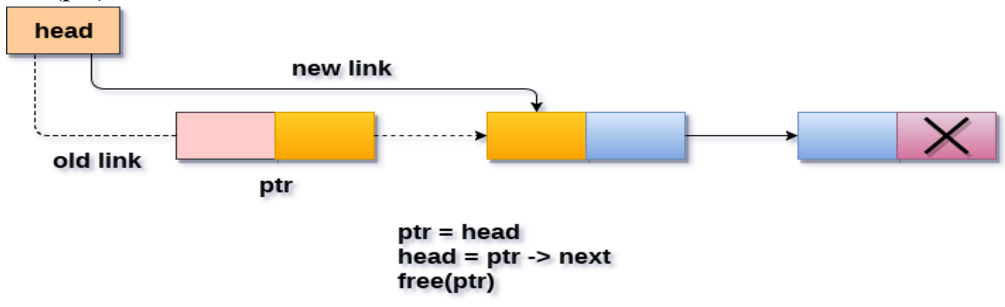
Deletion in singly linked list at beginning

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head. This will be done by using the following statements

```
ptr = head;
head = ptr->next;
```

Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.

```
free(ptr)
```



Deleting a node from the beginning

Algorithm

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

C function

```
void begdelete()
   {
       struct node *ptr;
       if(head == NULL)
       {
           printf("\nList is empty");
       }
       else
       {
           ptr = head;
           head = ptr->next;
           free(ptr);
           printf("\n Node deleted from the begining ...");
       }
```

}

Deletion in singly linked list at the end

Here are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

the condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null. This will be done by using the following statements.

```
ptr = head
head = NULL
free(ptr)
```

In the second scenario,

The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

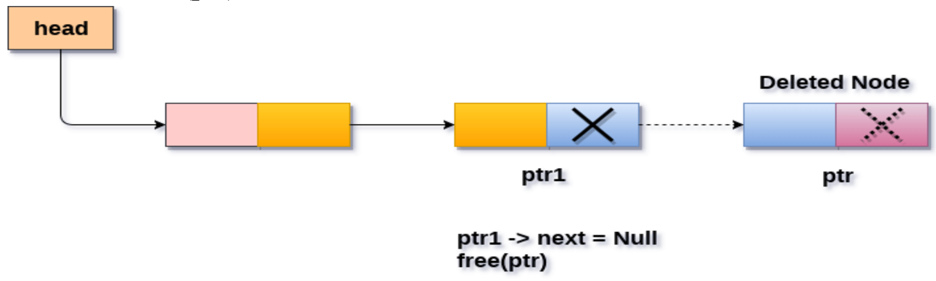
For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers ptr and ptr1 will be used where ptr will point to the last node and ptr1 will point to the second last node of the list.

this all will be done by using the following statements.

```
ptr = head;
        while(ptr->next != NULL)
        {
          ptr1 = ptr;
          ptr = ptr ->next;
        }
```

Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by ptr will become free. It will be done by using the following statements.

```
ptr1->next = NULL;
  free(ptr);
```



Deleting a node from the last

Algorithm

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

C Function

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

```
                }
            }
        }
```

Deletion in singly linked list after the specified node

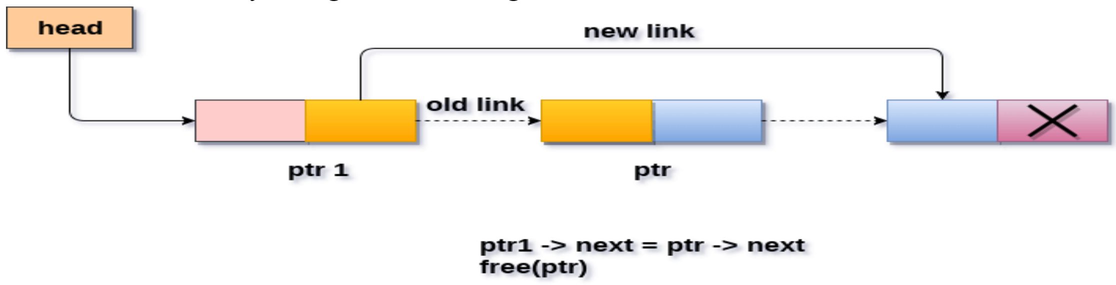
In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.

Use the following statements to do so.

```
        ptr=head;
            for(i=0;i<loc;i++)
            {
                ptr1 = ptr;
                ptr = ptr->next;

                if(ptr == NULL)
                {
                    printf("\nThere are less than %d elements in the list..",loc);
                    return;
                }
            }
```

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

This will be done by using the following statements.



Deletion a node from specified position

Algorithm

```
o   STEP 1: IF HEAD = NULL

        WRITE UNDERFLOW
           GOTO STEP 10
          END OF IF
```

```
o STEP 2: SET TEMP = HEAD
o STEP 3: SET I = 0
o STEP 4: REPEAT STEP 5 TO 8 UNTIL I<loc< li=""></loc<>
o STEP 5: TEMP1 = TEMP
o STEP 6: TEMP = TEMP → NEXT
o STEP 7: IF TEMP = NULL
          WRITE "DESIRED NODE NOT PRESENT"
          GOTO STEP 12
            END OF IF
o STEP 8: I = I+1
          END OF LOOP
o STEP 9: TEMP1 → NEXT = TEMP → NEXT
o STEP 10: FREE TEMP
o STEP 11: EXIT
```

Searching in singly linked list

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

```
o Step 1: SET PTR = HEAD
o Step 2: Set I = 0
o STEP 3: IF PTR = NULL

    WRITE "EMPTY LIST"
    GOTO STEP 8
    END OF IF

o STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL
o STEP 5: if ptr → data = item

   write i+1
   End of IF

o STEP 6: I = I + 1
o STEP 7: PTR = PTR → NEXT

  [END OF LOOP]
```

- **STEP 8:** EXIT

C Function

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

Traversing in singly linked list

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
ptr = head;
```

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

Algorithm

```
o STEP 1: SET PTR = HEAD
o STEP 2: IF PTR = NULL
     WRITE "EMPTY LIST"
    GOTO STEP 7
    END OF IF
o STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL
o STEP 5: PRINT PTR→ DATA
o STEP 6: PTR = PTR → NEXT
  [END OF LOOP]
o STEP 7: EXIT
```

SINGLY LINKED LIST ADVANTAGE

1) Insertions and Deletions can be done easily.
2) It does not need movement of elements for insertion and deletion.
3) It space is not wasted as we can get space according to our requirements.
4) Its size is not fixed.
5) It can be extended or reduced according to requirements.
6) Elements may or may not be stored in consecutive memory available
7) It is less expensive.

DISADVANTAGE

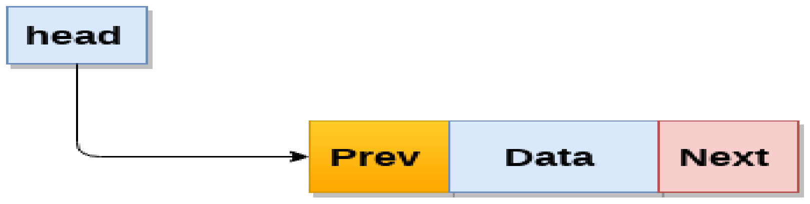
1) It requires more space as pointers are also stored with information.
2) Different amount of time is required to access each element.
3) If we have to go to a particular element then we have to go through all those elements that come before that element.
4) we can not traverse it from last & only from the beginning.
5) It is not easy to sort the elements stored in the linear linked list.

Applications of Linked Lists

Graphs, queues, and stacks can be implemented by using Linked List.

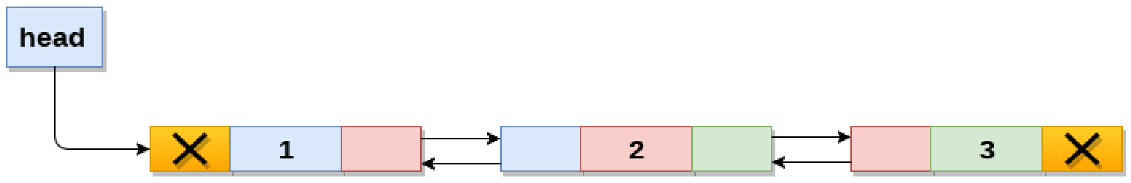
DOUBLE LINKED LIST

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



Node

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



Doubly Linked List

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

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

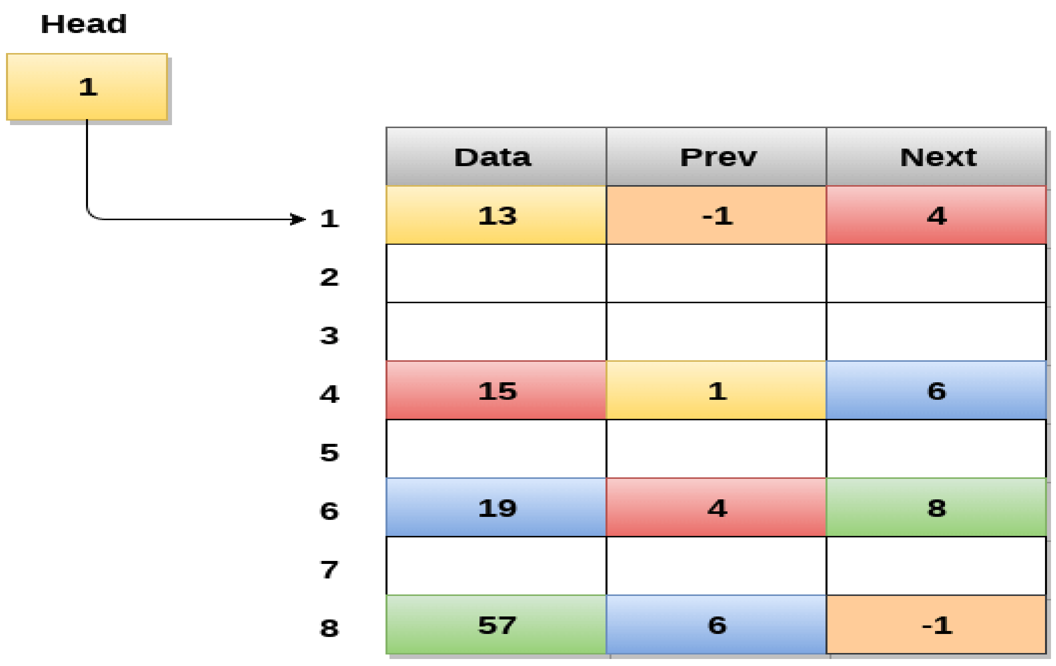
The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

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

Memory Representation of a doubly linked list

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

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.
We can traverse the list in this way until we find any node containing null or -1 in its next part.



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Operations on doubly linked list

The following operations are performed on double linked list
1) Insertion

- Insertion at beginning
- Insertion at End
- Insertion at specified position

1) Deletion
    - Deletion from the Beginning
    - Deletion from the End
    - Deletion of the node having specified data
2) Searching
3) Traversing

Node Creation

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

INSERTION

Insertion in doubly linked list at beginning

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.
There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory. This will be done by using the following statement.
  ```
  ptr = (struct node *)malloc(sizeof(struct node));
  ```
- Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.
  ```
  ptr->next = NULL;
  ptr->prev=NULL;
  ptr->data=item;
  head=ptr;
  ```
- In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer

of the node. The prev pointer of the existing head will point to the new node being inserted.

- This will be done by using the following statements.

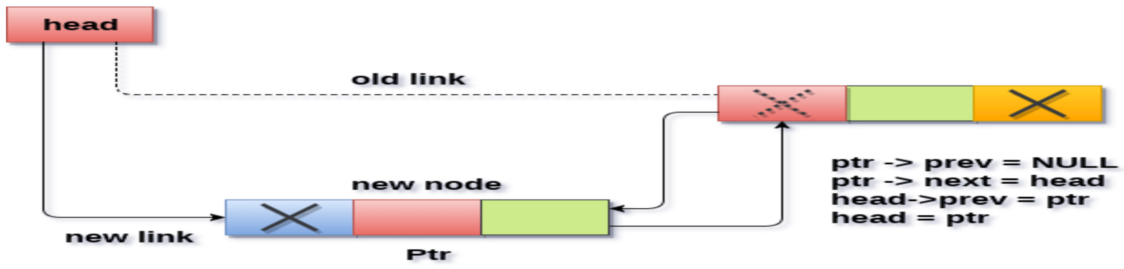
  ptr->next = head;
  head→prev=ptr;

  Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

  ptr→prev =NULL
  head = ptr

Algorithm :

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



Insertion into doubly linked list at beginning

C Function

```
void insertbeginning( )
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  int item;
  printf("enter the value");
  scanf("%d",&item);
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
        if(head==NULL)
        {
                 ptr->next = NULL;
                 ptr->prev=NULL;
                 ptr->data=item;
                 head=ptr;
        }
        else
        {
                 ptr->data=item;
                 ptr->prev=NULL;
                 ptr->next = head;
                 head->prev=ptr;
                 head=ptr;
         }
}
```

Insertion in doubly linked list at the end

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

- Allocate the memory for the new node. Make the pointer **ptr** point to the new node being inserted.

  ```
  ptr = (struct node *) malloc(sizeof(struct node));
  ```

- Check whether the list is empty or not. The list is empty if the condition **head == NULL** holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

  ```
  ptr->next = NULL;
   ptr->prev=NULL;
   ptr->data=item;
    head=ptr;
  ```

- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

```
Temp = head;
while (temp != NULL)
{
        temp = temp → next;
 }
```

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.

```
temp→next =ptr;
```

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

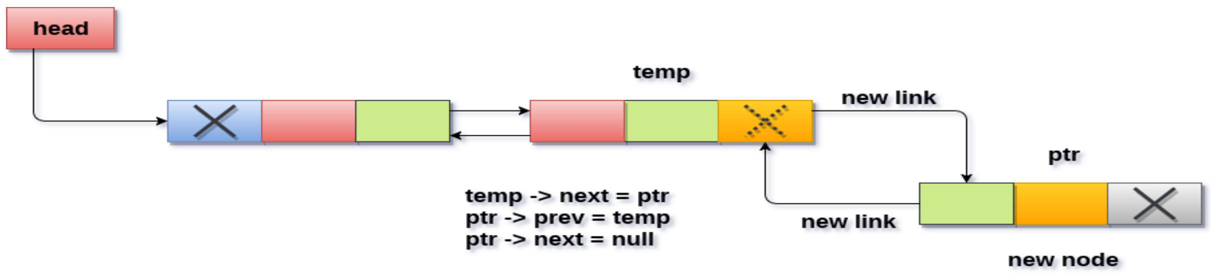
```
ptr → prev = temp;
```

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

```
ptr → next = NULL
```

Algorithm

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



Insertion into doubly linked list at the end

C Program

```
void insertlast()
{
  struct node *ptr = (struct node *) malloc(sizeof(struct node));
  int item;
  printf("enter the value");
  scanf("%d",&item);
  struct node *temp;
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
      ptr->data=item;
     if(head == NULL)
     {
        ptr->next = NULL;
        ptr->prev = NULL;
        head = ptr;
     }
     else
     {
       temp = head;
       while(temp->next!=NULL)
       {
          temp = temp->next;
       }
       temp->next = ptr;
```

```
        ptr ->prev=temp;
        ptr->next = NULL;
      }
printf("\nNode Inserted\n");
  }
}
```

Insertion in doubly linked list after Specified node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required. Use the following steps for this purpose.

- Allocate the memory for the new node. Use the following statements for this.

```
ptr = (struct node *)malloc(sizeof(struct node));
```

- Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.

```
        temp=head;
  for(i=0;i<loc;i++)
  {
    temp = temp->next;
    if(temp == NULL) // the temp will be //null if the list doesn't last long //up to mentio
        ned location
    {
        return;
    }
  }
```

- The temp would point to the specified node at the end of the **for** loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

```
ptr → next = temp → next;
```

make the **prev** of the new node ptr point to temp.

```
ptr → prev = temp;
```

make the **next** pointer of temp point to the new node ptr.

```
temp → next = ptr;
```

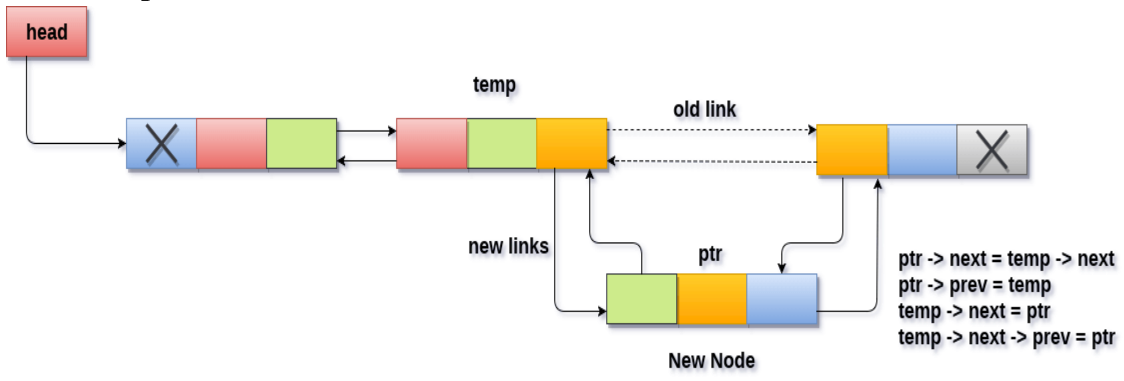
make the **previous** pointer of the next node of temp point to the new node.

```
temp → next → prev = ptr;
```

Algorithm

```
o Step 1: IF PTR = NULL
      Write OVERFLOW
      Go to Step 15
      [END OF IF]
```

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = START
- **Step 6:** SET I = 0
- **Step 7:** REPEAT 8 to 10 until I<="" li="">
- **Step 8:** SET TEMP = TEMP -> NEXT
- **STEP 9:** IF TEMP = NULL
- **STEP 10:** WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
  GOTO STEP 15
  [END OF IF]
  [END OF LOOP]
- **Step 11:** SET NEW_NODE -> NEXT = TEMP -> NEXT
- **Step 12:** SET NEW_NODE -> PREV = TEMP
- **Step 13 :** SET TEMP -> NEXT = NEW_NODE
- **Step 14:** SET TEMP -> NEXT -> PREV = NEW_NODE
- **Step 15:** EXIT



Insertion into doubly linked list after specified node

C Function

```
void insert_specified(int item)
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  struct node *temp;
  int i, loc;
  if(ptr == NULL)
  {
     printf("\n OVERFLOW");
```

```
 }
 else
 {
    printf("\nEnter the location\n");
    scanf("%d",&loc);
    temp=head;
    for(i=0;i<loc;i++)
    {
       temp = temp->next;
       if(temp == NULL)
       {
          printf("\ncan't insert\n");
          return;
       }
    }
    ptr->data = item;
    ptr->next = temp->next;
    ptr -> prev = temp;
    temp->next = ptr;
    temp->next->prev=ptr;
    printf("Node Inserted\n");
 }
}
```

DELETION OPERATION

Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

```
Ptr = head;
head = head → next;
```

now make the prev of this new head node point to NULL. This will be done by using the following statements.

```
head → prev = NULL
```

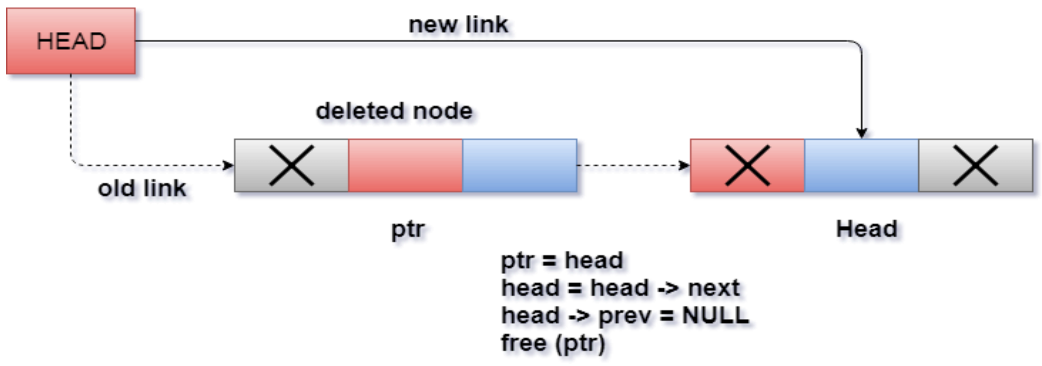
Now free the pointer ptr by using the **free** function.

```
free(ptr)
```

Algorithm

```
○ STEP 1: IF HEAD = NULL
          WRITE UNDERFLOW
          GOTO STEP 6
```

- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

C FUNCTION

```
void beginning_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW\n");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```

Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
if(ptr->next != NULL)
{
        ptr = ptr -> next;
}
```

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

```
ptr → prev → next = NULL
```

  free the pointer as this the node which is to be deleted.

```
free(ptr)
```

ALGORITHM

- **Step 1:** IF HEAD = NULL

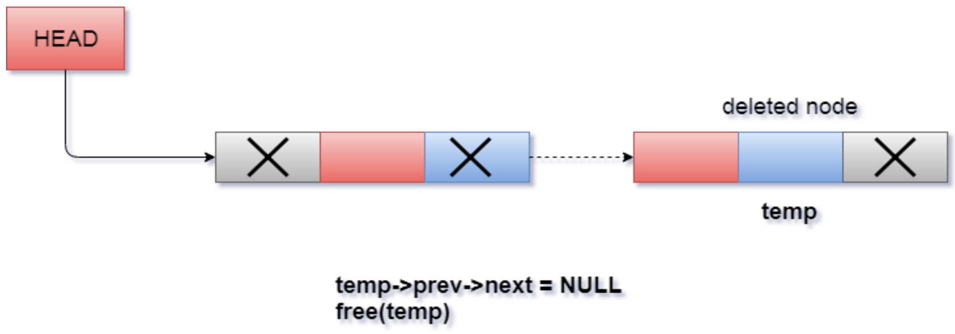
  Write UNDERFLOW
  Go to Step 7
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL
- **Step 4:** SET TEMP = TEMP->NEXT

  [END OF LOOP]

- **Step 5:** SET TEMP ->PREV-> NEXT = NULL

- **Step 6:** FREE TEMP
- **Step 7:** EXIT



Deletion in doubly linked list at the end

C PROGRAM

```
void last_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW\n");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      if(ptr->next != NULL)
      {
         ptr = ptr -> next;
      }
      ptr -> prev -> next = NULL;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```

Deletion in doubly linked list after the specified node

In order to delete the node after the specified data, we need to perform the following steps.

- Copy the head pointer into a temporary pointer temp.

```
temp = head
```

- Traverse the list until we find the desired data value.

```
while(temp -> data != val)
temp = temp -> next;
```

- Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
{
        return;
 }
```

- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
{
        temp ->next = NULL;
 }
```

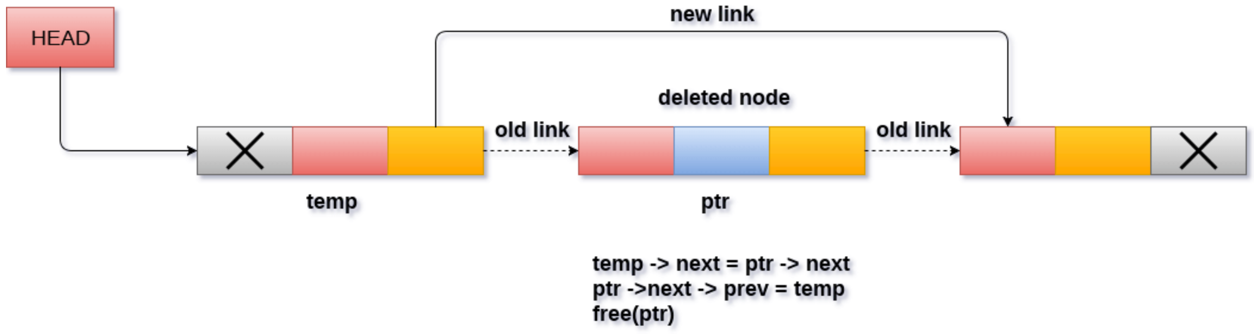
- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

```
ptr = temp -> next;
temp -> next = ptr -> next;
ptr -> next -> prev = temp;
free(ptr);
```

Algorithm

```
o Step 1: IF HEAD = NULL
  Write UNDERFLOW
 Go to Step 9
[END OF IF]
o Step 2: SET TEMP = HEAD
o Step 3: Repeat Step 4 while TEMP -> DATA != ITEM
o Step 4: SET TEMP = TEMP -> NEXT
 [END OF LOOP]
o Step 5: SET PTR = TEMP -> NEXT
o Step 6: SET TEMP -> NEXT = PTR -> NEXT
```

- **Step 7:** SET PTR -> NEXT -> PREV = TEMP
- **Step 8:** FREE PTR
- **Step 9:** EXIT



Deletion of a specified node in doubly linked list

C FUNCTION

```
void delete_specified( )
{
   struct node *ptr, *temp;
   int val;
   printf("Enter the value");
   scanf("%d",&val);
   temp = head;
   while(temp -> data != val)
   temp = temp -> next;
   if(temp -> next == NULL)
   {
      printf("\nCan't delete\n");
   }
   else if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = temp -> next;
      temp -> next = ptr -> next;
      ptr -> next -> prev = temp;
      free(ptr);
      printf("\nNode Deleted\n");
```

```
        }
      }
```

Searching for a specific node in Doubly Linked List

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.
  ptr = head
- declare a local variable I and assign it to 0.
  i=0
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm

```
o Step 1: IF HEAD == NULL
    WRITE "UNDERFLOW"
   GOTO STEP 8
   [END OF IF]
o Step 2: Set PTR = HEAD
o Step 3: Set i = 0
o Step 4: Repeat step 5 to 7 while PTR != NULL
o Step 5: IF PTR → data = item
  return i
  [END OF IF]
o Step 6: i = i + 1
o Step 7: PTR = PTR → next
o Step 8: Exit
```

C FUNCTION

```
void search()
{
   struct node *ptr;
   int item,i=0,flag;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
```

```
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
        }
    }
```

Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

```
            Ptr = head
```

then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

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

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

- **Step 1:** IF HEAD == NULL

  ```
   WRITE "UNDERFLOW"
  GOTO STEP 6
  [END OF IF]
  ```

- **Step 2:** Set PTR = HEAD
- **Step 3:** Repeat step 4 and 5 while PTR != NULL
- **Step 4:** Write PTR → data
- **Step 5:** PTR = PTR → next
- **Step 6:** Exit

C Function

```
int traverse()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      ptr = head;
      while(ptr != NULL)
      {
         printf("%d\n",ptr->data);
         ptr=ptr->next;
      }
   }
 }
```

Differences between Singly linked list and Doubly linked list

Singly linked list (SLL)	**Doubly linked list (DLL)**
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.

Singly linked list (SLL)	**Doubly linked list (DLL)**
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.

3.Write a program that uses functions to perform the following operations on circular linked list:

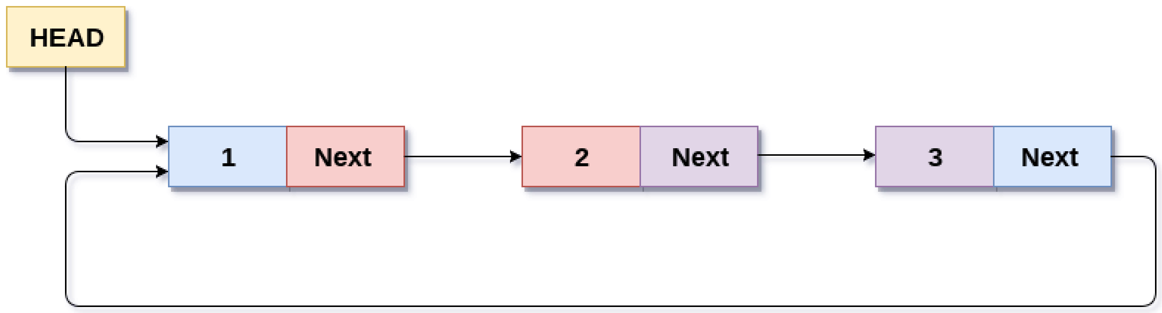
i) Creation ii) Insertion iii) Deletion iv) Traversal

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.

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

The following image shows a circular singly linked list.



Circular Singly Linked List

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

Write a program that uses functions to perform the following operations on circular linked list:

i) Creation ii) Insertion iii) Deletion iv) Traversal

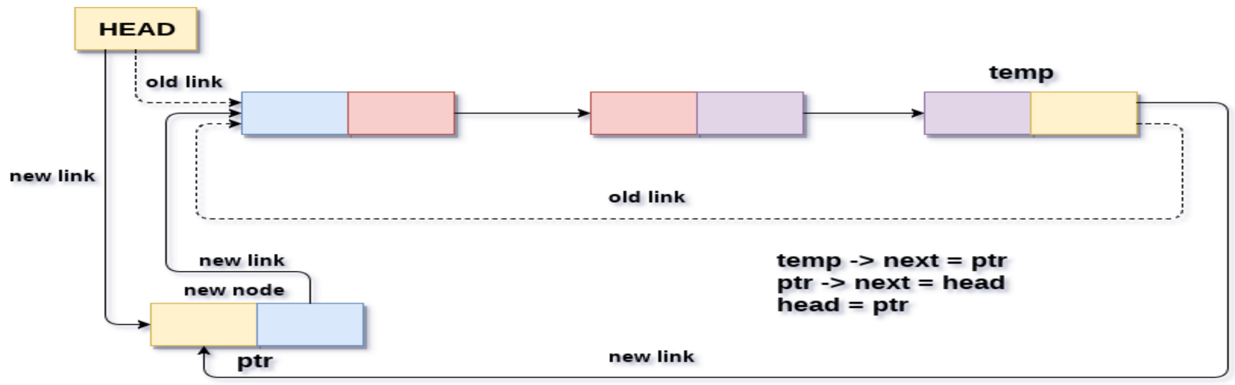
i)Creation

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("1.Append List\n2.Exit\n3.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }

   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      ptr -> data = item;
```

```
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
    printf("\nNode Inserted\n");
    }
}
```

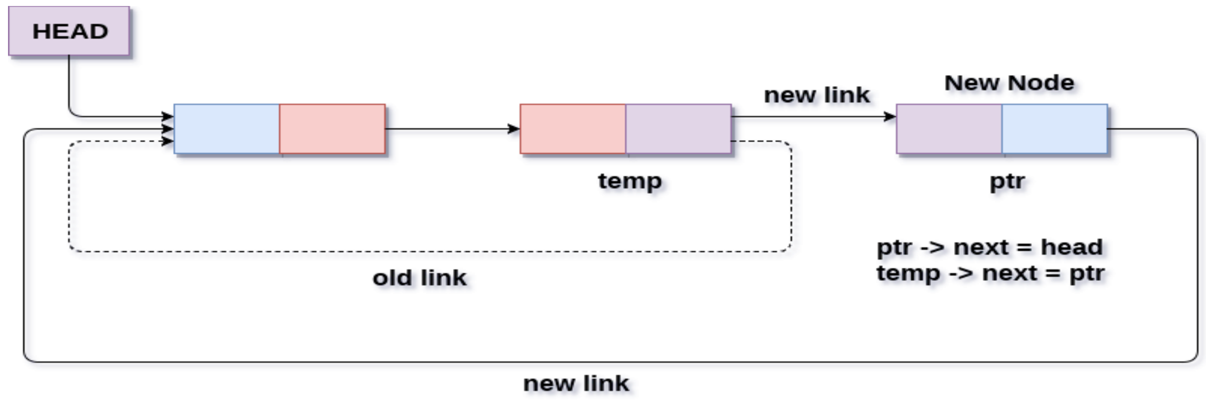
ii)Insertion

Insertion into circular singly linked list at beginning



Insertion into circular singly linked list at beginning

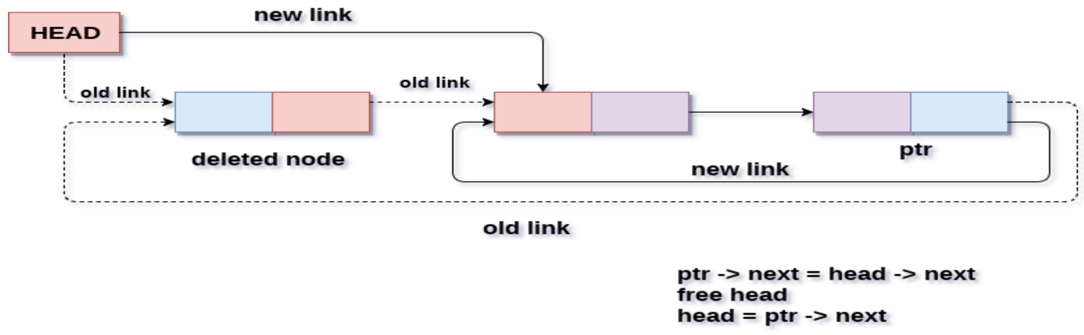
Insertion into circular singly linked list at the end



Insertion into circular singly linked list at end

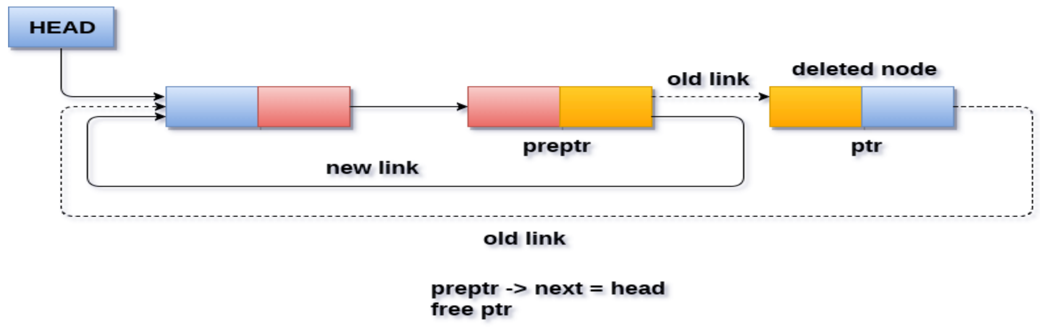
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void beginsert ();
void lastinsert ();
void display();
void main ()
{
   int choice =0;
   while(choice != 4)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.display\n4.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         beginsert();
         break;
         case 2:
         lastinsert();
         break;
         case 3:
         display();
         case 4:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
    }

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

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

  }

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

        while(ptr -> next != head)
        {

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

Deletion in circular singly linked list at beginning



Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end



Deletion in circular singly linked list at end

iii) Deletion

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void create();
void begin_delete();
void last_delete();
void display();
void main ()
{
   int choice =0;
   while(choice != 5)
```

```
    {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.create\n2.Delete from Beginning\n3.Delete from last\n4.Show\n5.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {

            case 1:
            create();
            break;
            case 2:
            begin_delete();
            break;
            case 3:
            last_delete();
            break;
            case 4:
            display();
            break;
            case 5:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}

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

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

}

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

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

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

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

    }
}

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

        while(ptr -> next != head)
        {

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

}
```

iv) Traversal

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void traverse();
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         traverse();
         break;
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }

   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      ptr -> data = item;
```

```
        if(head == NULL)
        {
          head = ptr;
          ptr -> next = head;
        }
        else
        {
          temp = head;
          while(temp->next != head)
            temp = temp->next;
          ptr->next = head;
          temp -> next = ptr;
          head = ptr;
        }
    printf("\nNode Inserted\n");
    }

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

        while(ptr -> next != head)
        {

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

}
```