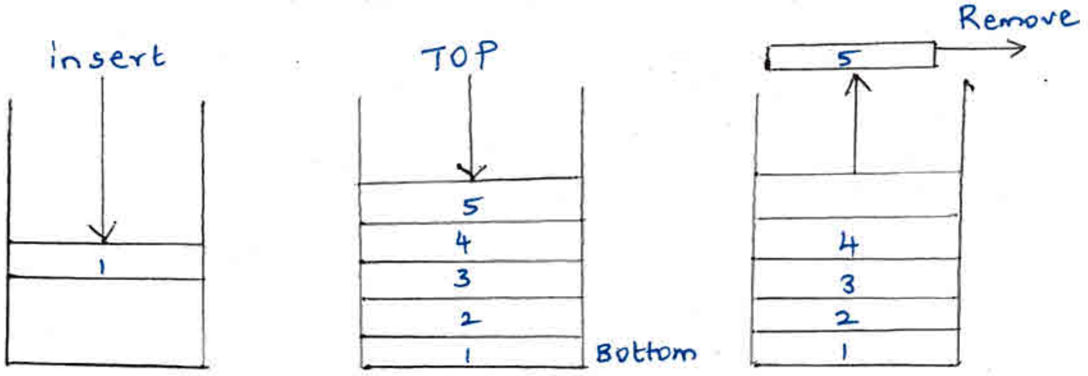
Stack ADT

a) A stack is a data structure in which an element may be inserted or deleted only at one end, called the top of the stack.

b) The fundamental operations on a stack are 'push', which is equivalent to an insert, and 'pop' which deletes the most recently inserted element.

c) Stacks are also referred as LIFO (Last-in-First-out) structure.

Stack scheme:



Algorithm for push operation:

a) Associated with each stack is 'Top of stack', which is -1 for an empty stack (this is how an empty stack is initialized).

```
Push ( stack, item)
{
   if ( s->top = = MAX - 1)
   {
    S -> top++;
    S -> arr[S -> top] = item;
  }
```

Algorithm for POP operation:

```
Pop(stack)
{
  int data;
  if (S -> top == -1)
   {
     Printf ("Stack is empty");
     return NULL;
   }
  data = S -> arr[S -> top];
  S -> top--;
  return data;
}
```

Applications of Stack:

1. matching of nested parentheses in arithmetic expressions.
2. Evaluation of arithmetic expressions
3. Conversion of infix expression to postfix expression.
4. Evaluation of postfix expression.

Program of stack implementation using Array

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
 int top = -1;
 int stack_arr[MAX];
 Void main ( )
 {
   int choice;
   while (choice != 4)
   {
     Printf (" 1. push  2. pop  3. Display  4. Quit \n");
     Printf ("Enter your choice:");
     Scanf ("%d", &choice);
     Switch (Choice)
     {
        case 1 : push ( );
                 break;
        case 2 : pop ( );
                 break;
        case 3 : display ( );
                 break;
        case 4 : exit (1);
     }
   }
 }
```

```
Push ( )
{
  int pushed_item;
  if (top = = MAX-1)
   {
     Printf ("Stack overflow\n");
   }
  else
   {
     Printf ("Enter the item to be pushed in stack\n");
     Scanf ("%d", &pushed_item);
     top = top+1;
     Stack_arr [top] = Pushed_item;
   }
 }

Pop ( )
{
   if (top = = -1)
   {
    Printf (" Stack underflow\n");
   }
  else
   {
    Printf ("Popped element is : %d \n", Stack_arr [top]);
    top = top-1;
   }
 }
display ( )
{
  int i;
  if (top= = -1)
else
{
 Printf (" stack elements : \n");
  for (i= top ; i >=0; i--)
   Printf ("%d\n", stack_arr [i]);
  }
}
```

Applications of Stack (Example).

Conversion of Infix Expression to post fix expression:

a) Consider the following arithmetic expressions,

i) a+b*c
ii) (a+b)*c
} How do computers evaluate expressions like these.

In case i) the Computer has to perform multiplication b*c first and then the add operation.

In case ii) the computer has to perform addition (a+b) first and then the multiplication.

b) The fact is that the Compiler can rework on your expression into a new form, which is called as Postfix notation (or reverse polish) which takes care of this evaluation problem.

c) The Conventional way of writing expressions is Called infix because the opeators Come in between the operands.

d) The postfix form of an expression Calls for each operator to appear after its operands.

e) The postfix form of expressions will not contain any Parentheses.

f) The postfix form of expressions specify the actual order of operations (no priority) without any parentheses (which are Unnessary)

Rules for Pushing operator or Parenthesis onto the stack and Popping from the stack.

Character Scanned	Stack TOP	What is to be done ?
operand	-	insert it to the postfix String.
operator +, -, *, /	empty	Push the operator scanned on to the stack
operator +, -, *, /	operator +, -, *, /	If the operator Scanned is of higher precedence Push it onto the stack.
operator +, -, *, /	left Parentheses '('	Push the operator scanned into the stack.

Character scanned	Stack top	What is to be done?
operator +, -, *, /	right parentheses ')'	This is not possible for we will not push any time ')' onto the stack.
left parentheses '('	+, -, *, /, (	Push the left Parenthesis '(' onto the stack.
left parenthesis '('	right parenthesis ')'	Not possible, we will not push any time ')' onto the stack
right parenthesis ')'	left Parenthesis '('	Pop the stack and add all operators into postfix string until you find '('. Discard or pop the left Parenthesis from the stack.
right parenthesis ')'	+, -, *, /	Pop the stack top element and insert it into the postfix string.

8) we are now in a position to define a function Precedence 'Prcd' that sets Precedence between operators and Parenthesis. The first argument being the stack top element and the second, the character scanned.

This function is defined as follows,

1. when the first argument is an operator with equal or higher Precedence,

   Prcd ('+','+') = Prcd ('+','-') = true

   Prcd ('-','+') = Prcd ('-','-') = true

   Prcd ('*','*') = Prcd ('*','/') = true

   Prcd ('/','*') = Prcd ('/','/') = true

   Prcd ('*','+') = Prcd ('*','-') = true

   Prcd ('/','+') = Prcd ('/','-') = true

2. when the first argument is an operator with lower precedence

   Prcd('+','*') = Prcd ('-','*') = False

   (or)

   Prcd ('+','/') = Prcd ('-','/') = False.

3 when the First argument is a left Parenthesis

   Prcd ('(', op) = false

   where 'op' is any operator (+, -, *, /) or another left Parenthesis.

4. when the second argument is a left Parenthesis, the function returns 'false' Provided the first argument is not a right Parenthesis.

5. when the Second argument is a right Parenthesis the function returns 'true' Provided the first argument is not

Example:-

Infix Expression :- a + b * c + (d * e + f) * g

Character Scanned	Remarks	Stack	output.
a	add it to postfix String.		a
+	Push it onto the Stack	+	a
b	add it to postfix String	+	ab
*	push it on to the Stack	* +	ab
c	add it to the postfix String	* +	abc
+	Push it onto stack after popping '*' and '+' because of higher Precedence.	+	abc*+

Character Scanned	Remarks	Stack	Output
'('	Push it onto the Stack	(+	abc*+
d	add it to postfix String	(+	abc*+d
*	Push it on to Stack	* (+	abc*+d
e	add it on to postfix String	* (+	abc*+de
+	Pop all elements Until '(' from stack and then push '+' onto stack	+ (+	abc*+de*
f	Add it to postfix String	+ (+	abc*+de*f
)	empty the stack till you find '('	+	abc*+de*f+
*	Push it onto Stack	* +	abc*+de*f+
g	add it to postfix String		abc*+de*f+g

Algorithm for Conversion from infix String to postfix String.

1. Input an infix String.

2. Scan the expression from left to right

3. During your scanning,

   a) If you found an operand, insert it to the Postfix String ; Continue Scanning.

   b) If you found an operator (or Parenthesis) having a higher Precedence over the stack top element, then Push the opelator (or parenthesis) onto the Stack, Provided it is not a right Parenthesis. when it is a right Parenthesis, Pop the stack top element and discard it. Initially the stack will be empty and you have to push any left Parenthesis encountered. Continue Scanning.

   c) If you found an operator (or Parenthesis) having a lower Precedence over the stack top element, then pop the stack top element and insert it into the postfix String. you must ensure that the Stack is non-empty before you pop elements from it. [illegible]

d) when the end of the string is reached, Pop out all the elements from the Stack and Insert them into the postfix string.

e) Finally, Print the postfix string that contains both operators and operands.

more Examples:

	Infix		postfix.
1.	$a+b*c$	→	$abc*+$
2.	$(a+b)*c$	→	$ab+c*$
3.	$a+b*c-d/e$	→	$abc*+de/-$
4	$a*b+c/d-e$	→	$ab*cd/+e-$
5	$a*(b+c)/d-e$	→	$abc+*b/e-$
6.	$a+b*(c+d)/e$	→	$abcd+*e/+$

cursor based implementation of Linked Lists.

a) many languages, such as BASIC and FORTRAN, do not support pointers.

b) If linked lists are required and pointers are not available, then an alternative implementation must be used. The alternate method is the cursor implementation.

c) The two important features present in a pointer implementation of linked lists are as follows,

1. The data are stored in a collection of structures. Each structure contains data and a pointer to the next structure.

2. A new structure can be obtained from the system's global memory by a call to 'malloc' and released by a call to 'free'.

d) our cursor implementation must be able to simulate this. The logical way to satisfy the First Feature is to have a global array of structures. For any cell in the array, its array index can be used in place of an address.

e) We must now simulate ~~condition~~ the second feature by allowing the equivalent of 'malloc' and 'free' for cells in the CursorSpace array.

f) To do this, we will keep a list (the Freelist) of cells that are not in any list. The list will use cell '0' as a header.

Initialized cursor space.

Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

Example of a cursor implementation of Linked Lists

Slot	Element	Next
0	–	6
1	b	9
2	f	0
3	header	7
4	–	0
5	header	10
6	–	4
7	c	8
8	d	2
9	e	0
10	a	1

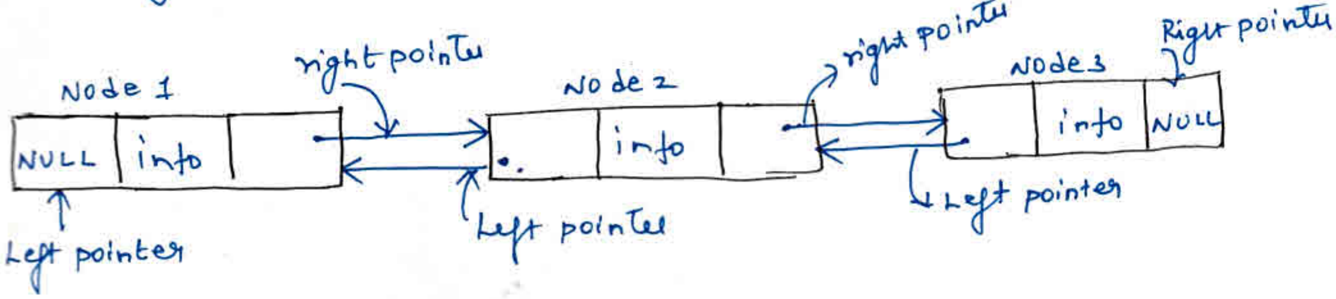
L (List) if value is '5'. then 'L' represents the list {a, b, e}

M (List) value is '3' then 'M' represents the list {c, d, f}

Doubly Linked List

a) A node in a doubly linked list will have two pointers, which we call by the names 'left' and 'right'. Doubly linked list can be used to traverse a list in both forward and backward directions.

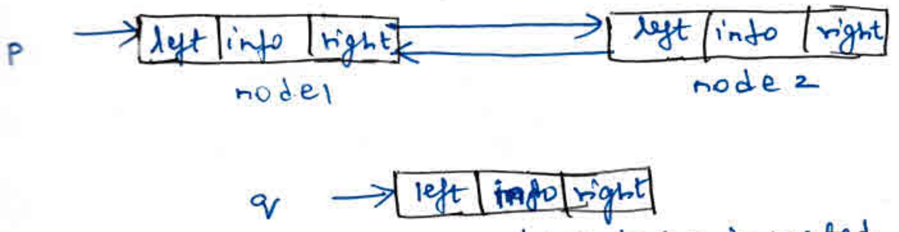
b) The right pointer will point to the next node whereas the left pointer will point to the previous node.



c) Left pointer = NULL indicates that it is a first node in the list. Similarly if Right pointer = NULL indicates that it is a last node in the list.

Inserting a Node into a doubly Linked List.

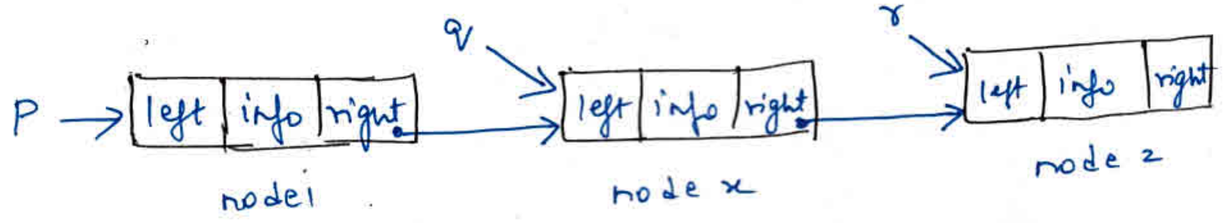
a) We can insert a node either to the right of a given node or to the left of a given node. This is possible because we have two pointers.



node x to be inserted after P (i.e, node1)

The operations required are Coded as,

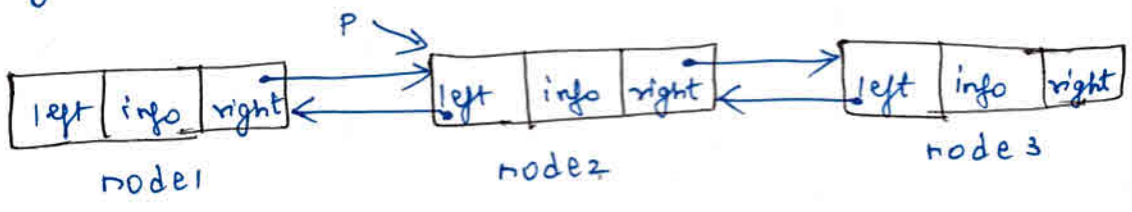
```
q = getnode ();
r = P->right;
P->right = q;
q->right = r;
```



To establish link in reverse direction, the code can be written as,

```
r->left = q;
q->left = P;
```

Deleting a node from a doubly linked List.



a) we use two auxillary pointers 'q' and 'r'. In 'q' we store the value of P->left and in 'r', we store the value of P->right For this we write,

```
q = P->left;    (i.e node 1).
r = P->right;   (i.e node 3)
q->right = r;
r->left = q;
```

Applications of List.

1. Representation of polynomials and performing operations like additions or multiplications.

2. Sorting.

3. Searching.

Example: Radix Sort (or card sort or Bucket sort).

a) Consider that we have 'N' integers in the range 1 to M or (0 to M-1), we can use this information to obtain a fast sort known as bucket sort.

b) The input is 64, 8, 216, 512, 27, 729, 0, 1, 343, 125 (i.e, example of first 10 cubes, arranged randomly)

Buckets after first step of radix sort

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

sorts by Least Significant Digit. (Base 10).

Buckets after Second Pass of radix sort

8 1 0	216 512	729 27 125		343	£	64			
0	1	2	3	4	5	6	7	8	9

Sorted with respect to two LSD

Buckets after Final Pass of radix sort

64 27 8 1 0	125	216	343		512		729		
0	1	2	3	4	5	6	7	8	9

Queue:

a) Queue is data structure in which insertion takes place at the end called as 'rear' (i.e, Back) and deletion takes place at 'front'. (i.e, Two ends are there, one end for insertion operation and another for deletion operation).

b) The first item inserted into the queue will be serviced first (i.e, removed first from the queue). Hence, we can also call Queue as FIFO (First In First out) data structure

c) we need two variables 'Front' and 'rear' to represent the two ends of a queue.

Example:-

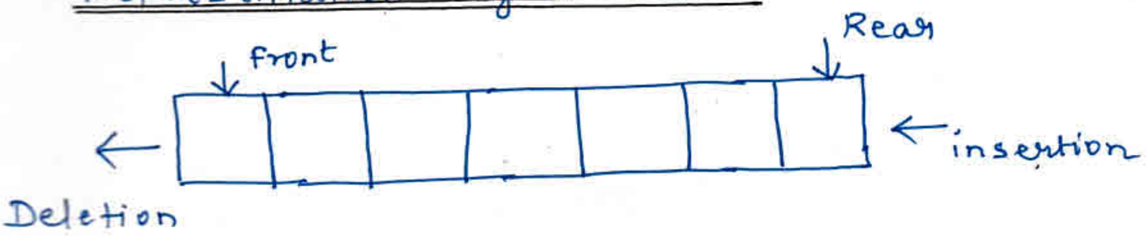
Consider a queue with 5 integer items. Initially the queue is empty as shown.

Initially we set rear = -1 and Front = 0

item [0]	item [1]	item [2]	item [3]	item [4]

rear = -1
Front = 0				

Representation of a Queue



Insertion of an Element

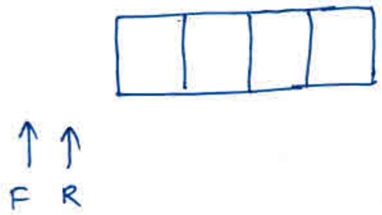
```
Qinsert (Q, F, R, N, Y)                Q → Queue
                                        F → Front pointer
Initialize F = R = 0                   R → Rear pointer
                                        N → Maximum size of
  1. [overflow ?]                            Array
        if R ≥ N                         Y → Element to be
        then                                  inserted.
          write ('overflow')
          Return

  2. [increment Rear pointer]
        R ← R + 1

  3. [insert Element]
        Q [R] ← Y
  4  [Is Front Pointer properly set ?]
        if F = 0
        then F ← 1
        Return
```

Deleting an element from Queue.

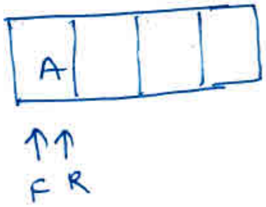
Q Delete (Q, F, R)

```
1. [underflow?]
      if $F = 0$
      then write ('underflow')
       Return (0)

2. [Delete Element]

       $Y \leftarrow Q[F]$        [y is a temporary variable]

3  [Queue Empty?]
         if $F = R$
         then $F \leftarrow R \leftarrow 0$
         else
             $F \leftarrow F + 1$      [increment front pointer]

4. [Return Element]

       Return (Y)
```

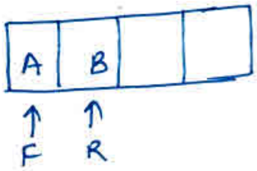
Enample: for Queue Insertion and Deletion



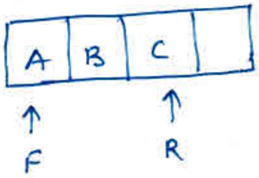
Empty



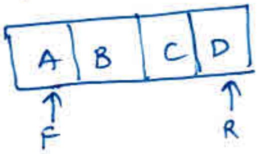
Insert `A'



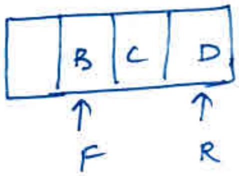
Insert `B'



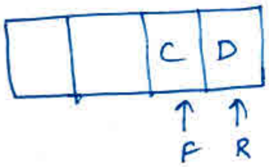
Insert `C'



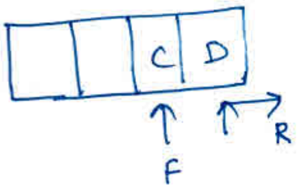
Insert `D'



Delete A



Delete B

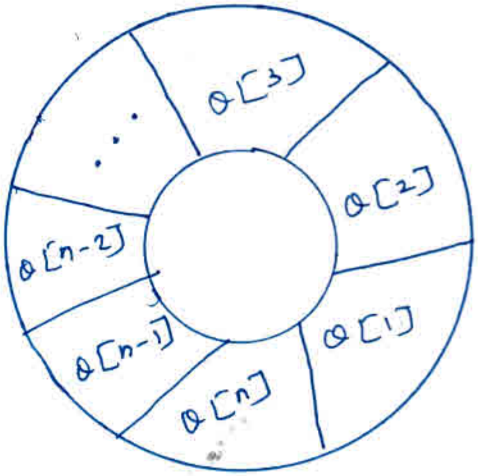


Insert E

Overflow occurs Eventhough the Queue has space. To avoid this Circular Queue implementation can be used.

Circular Queue

A more suitable method for representing a queue, Which Prevents an encessive use of memory, is to arrange the elements Q[1], Q[2] ... Q[n] in a circular fashion with Q[1] following Q[n].



Inserting an element into an circular Queue.

```
CQINSERT(F,R,Q,N,Y)
Initialize F=R=0
1. [Reset Rear pointer]
      if R=N
      then R=1
      else
       R=R+1
2. [overflow]
      if F=R
      then write("overflow")
       Return
3. [insert Element]
      Q[R]=Y.
```

Deleting an element from a Circular Queue.

```
CQDELETE (F, R, Q, N)

1.  [Underflow?]
        if F = 0
        then write ('underflow')
         return (0)

2.  [Delete element]
          y = Q [F]

3.  [Queue Empty?]
          if F = R
          then F = R = 0
          Return (y)

4.  [increment front pointer]
           if  F = N
           then  F = 1
           else
             F = F + 1
            Return (y).
```