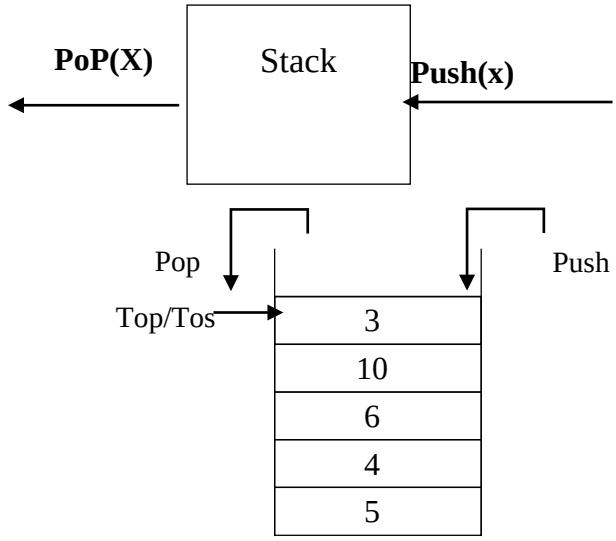
Stacks ADT:

- A stack is an ordered list in which all insertions and deletions are made at one end,called the top.
- Stack is a list with the restriction that insertions and deletions can be performed in only one position, namely the end of the list called Top.
- It follows **LIFO** approach. LIFO represents **"Last In First Out".** The basic operations
    - **Push**:Equivalent to insert.
    - **Pop :**Equivalent to delete. It deletes the most recently inserted element.

Stack Model:



The Basic Operations performed in the stack are:

- PUSH()
- POP()
- IsEmpty()
- IsFull()

EXCEPTION CONDITIONS IN STACK

Overflow:

This is the process of trying to insert an element when the stack is full.

Underflow:

This is the process of trying to delete an element when the stack is empty.

Stack can be implemented in following ways

- Array Implementation
- Linked List Implementation

Primitive operations on the stack

- To create a stack
- To insert an element on to the stack.
- To delete an element from the stack.
- To check which element is at the top of the stack.
- To check whether a stack is empty or not.
- To check whether the stack is full or not

Implementation of Stack:

There are two methods of implementing stack operations.

- Array implementation
- Linked List implementation

1.*Array Implementation of Stack*:

Global Declaration:

```
int *stack, top= -1,maxsize;
```

Function to Create Dynamic Stack:

Initially the Stack is declared as a pointer. Using create() function once the maximum size is known the stack array can be dynamically created.

```
void create()
{
      printf("Enter the size of Stack:");
      scanf("%d",&maxsize);
      Stack=(int)malloc(sizeof(int)*maxsize);
}
```

Function to Check IsFull or IsEmpty:

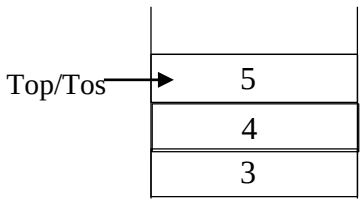
A Pop on an empty stack or a Push on a full stack will overflow the array bounds exception and cause a crash. To avoid this, the following functions are used.

```
int IsFull(int Stack[])
{
      return top==maxsize;
}
int IsEmpty(int Stack[])
{
      return top==-1;
}
```

PUSH Operation:

Inserting an element into the stack is called PUSH operation.

Example: Push 3,4,5 into the stack.



Function for Push Operation: Inserts an element into the stack.

```
void Push(int Stack[],int x)
{
      if(IsFull(Stack))
      printf("\nStack Overflow");
      else
      {
            top++;
            Stack[top]=x;
      }
}
```

Explanation: If top is maxsize, (Maxsize is the maximum size of the stack) it implies that the stack is full; no more elements can be added into the stack. Otherwise, increment top by 1. Store the data in stack[top].

Function for POP Operation:

Delete Operation is called POP operation in stack.

```
void Pop(int Stack[])
{
      if(IsEmpty(Stack))
            printf("\nStack underflow/empty");
      else
          top--;
}
```

Explanation:

If top=-1, it implies that the stack is empty; no element can be deleted from the stack.
Otherwise, decrement top by 1.

Function for Display Operation:

```
void display()
{
      int i;
      if(top==-1)
            printf("\n Stack is empty");
                  else
                  {
                        printf("\nThe elements in the stack are \n");
                        for(i=top;i>=0;i--)
                              printf("%d\t",Stack[i]);
                  }
               }
```

Explanation:

If top =-1, it implies that there are no elements in the stack; stack is empty.
Otherwise, display each element in the stack by formulating a loop; where i is initialized to zero, i is incremented till it reaches top.

Disadvantages of stack using array implementation:

- The size of the Stack is limited.
- If an element is to be inserted into the stack and if the array is fully utilized then the stack will overflow.

2. Linked List Implementation:

Code to Create a Stack:

```
struct Stack
{
      int Element;
      struct Stack *Next;
};
struct Stack *Ptr;
typedef struct Stack *STACK;
STACK create()
{
      STACK S;
      S=(STACK)malloc(sizeof(struct Stack));
      S->Next=NULL;
      return S;
}
```

The structure Stack is declared globally so that all the functions can access the structure. The pointer S->Next represents the address of top element of the Stack.

PUSH Operation:

```
void  Push(int x,STACK S)
{
      STACK TmpCell;
      TmpCell=(STACK)malloc(sizeof(struct Stack));
      TmpCell->Element=x;
      TmpCell->Next=S->Next;
      S->Next=TmpCell;
}
```

Explanation: This function creates a new node called TmpCell. The data is stored in the Element part of the TmpCell. The first element pointed by S-> Next is stored in TmpCell->Next. The address of TmpCell is stored in S->Next.

Deletion operation:

```
int IsEmpty(STACK S)
{
      return S==NULL;
}
void Pop(STACK S)
{
      if(IsEmpty(S))
            printf("\nStack is empty");
      else
            S->Next=S->Next->Next;
}
```

Explanation: First it checks whether Stack is Empty. If yes, it implies that the stack does not contain any nodes. Otherwise, Stack pointer S moved to Next pointer. It is only a logical deletion and not physical deletion.

Display operation

```
void display(STACK S)
{
      if(IsEmpty(S))
            printf("\nStack is empty.");
      else
      {
            printf("\nThe elements in the stack are \n");
            for(Ptr=S->Next;Ptr!=NULL;Ptr=Ptr->Next)
                  printf("%d\t",Ptr->Element);
      }
}
```

Explanation: This function displays all the elements in the element part of the nodes one after another.

Applications of Stack

1. Some of the applications which uses Stacks are,
2. Balancing Symbols
3. Evaluation of postfix expression
4. Conversion of infix to postfix expression
5. Function calls.

1. Balancing Symbols:

Compilers checks the program for syntax errors. During this process it checks for balancing of, parenthesis, braces and brackets. The number left parenthesis should be equal to the number of right parenthesis in the expression.

A stack is used to balance the parenthesis of the given expression .The simple algorithm uses a stack is as follows .

1. Make an empty stack. Read characters until end of file.
2. If the character is an opening symbol, push it onto stack.
3. If it is a closing symbol, then if the stack is empty, report an error. Otherwise, pop the stack.
4. If the symbol popped is not the corresponding opening symbol, then report an error.
5. At the end of file, if the stack is not empty, report an error.

2. Evaluation of Postfix Expressions:

Expression:

The collection of operators and operands are called expression.

- Operands: numbers or alphabets
- Operators: +, -, *, /
- Parenthesis: ( )
- Precedence:
- '(' and ')' have the highest precedence
- '*' and '/ 'have lower precedence than '(' and ')'
- '+' and '-' have lower precedence than '*' and '/'

Types of expression:

Infix: (operand 1) operator (operand 2)
Prefix: operator (operand 1) (operand 2)
Postfix: (operand 1) (operand 2) operator

The expressions in which the operators are placed at the end of operands on which they are going to operate is called **Postfix expression.**

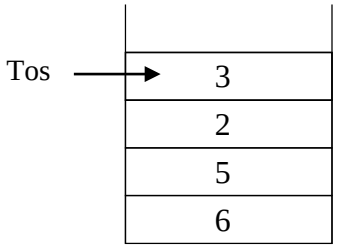
Algorithm to evaluate postfix expression:

1. When an operand is seen, it is pushed on to the stack.
2. When an operator is seen, the operator is applied to the two numbers that are popped from the top of the stack, and return the result pushed back on to the stack.

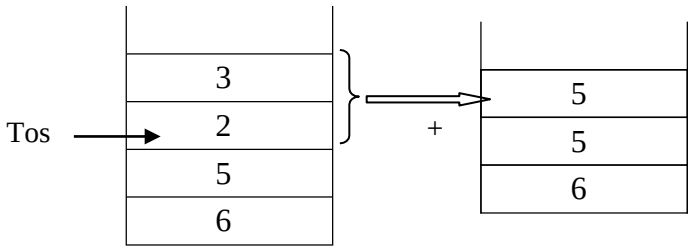
e.g : 6 5 2 3 + 8 * + 3 + *

Consider TOS as a stack pointer.

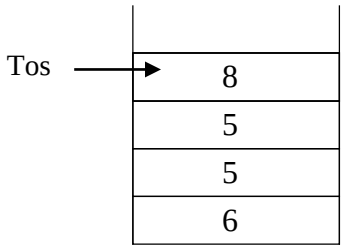
i. First four numbers are operand. So push it on to the stack.



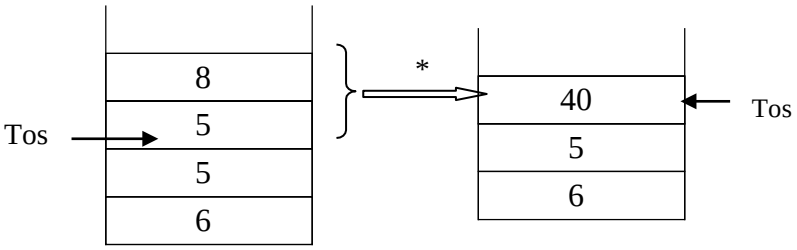
ii. Next symbol is a operator , '+'. So, pop the most recent two operands from the stack and do the operation. i.e., (3+2) = 5 and return the result(5) back to stack.



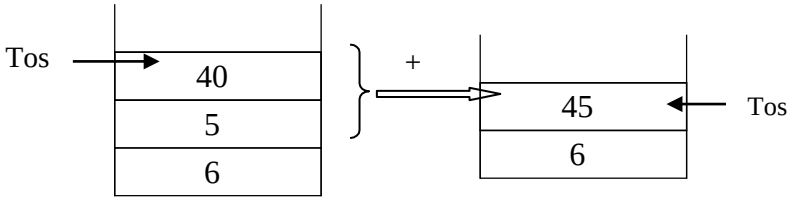
iii. Push the operand ‘8’ to stack.



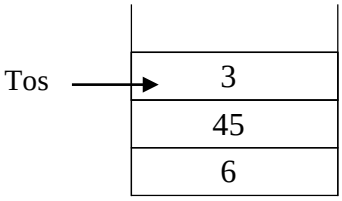
iv. Next symbol is a operator, ‘*’ so pop two elements from the stack and apply the operation i.e., 8*5=40 then push the result 40 again to the stack.



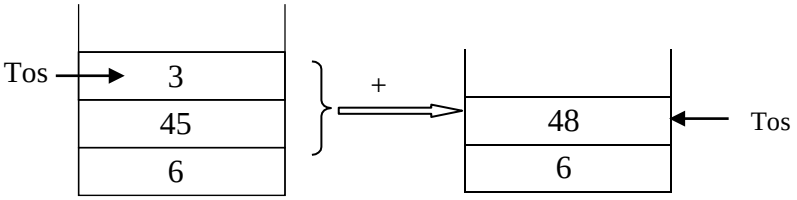
v. Next symbol is a operator, ‘+’ => (40+5 = 45).



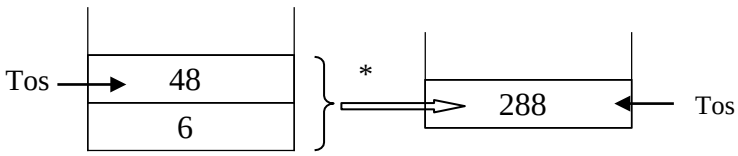
vi. Next symbol is a operand push it on to the stack.(i.e. 3)



vii. Next symbol is a operator, ‘+’ => (3+45 =48)



viii. Next symbol is a operator, ‘*’ => (48 * 6 = 288)



After Evaluation, The Compiler will result 288 as output.

3. Infix to postfix Conversion

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

Algorithm :

(1) Initialize a stack to empty.

(2) Read a symbol

(2.1) If it is an operand, place onto the output.

(2.2) If it is an operator

(a) If it is a right parenthesis, then pop the stack, write symbols, until a left parenthesis is encountered.

Remark: The '(' is popped, but not written as output.

(b) If it is any other symbol, such as '+', '*', '(', pop entries from the stack until an entry of lower priority is found, or '(' is found. When the popping is done, push the operator onto the stack.

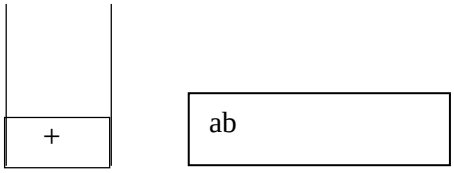
Remark: If '(' is found, don't pop it.

(3) Go to step (2).

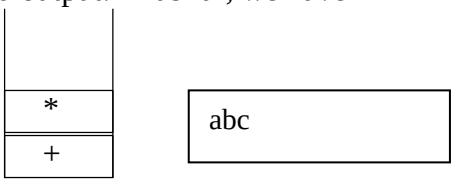
(4) If read the end of input, pop the stack until it is empty, writing symbols onto the output.

Input : a + b*c +(d*e+f)*g

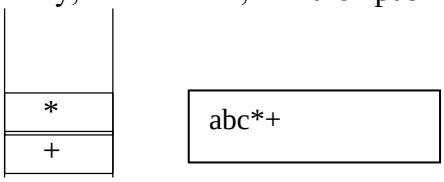
➢ First, the symbol 'a' is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next b is read and passed through to the output.



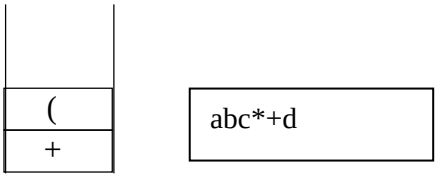
➢ Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, c is read and output. Thus far, we have



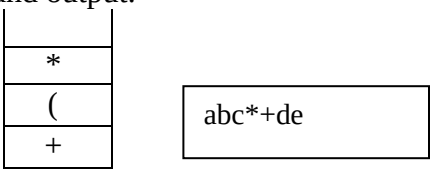
➢ The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output; pop the other '+', which is not of lower but equal priority, on the stack; and then push the '+'.



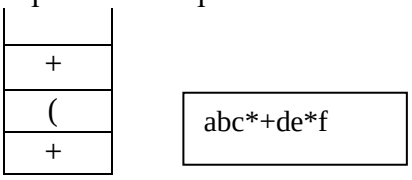
➢ The next symbol read is a '(', which, being of highest precedence, is placed on the Stack Then d is read and output.



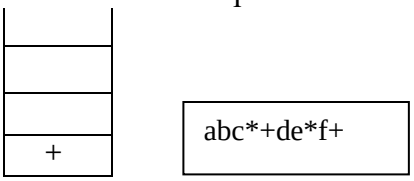
➢ Next by reading a '*', the open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



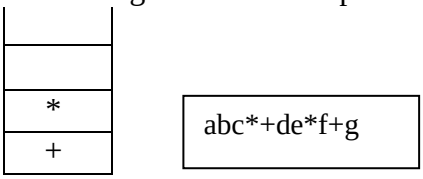
➢ The next symbol read is a '+'. We pop and output and then push '+'. Thenread and output f.



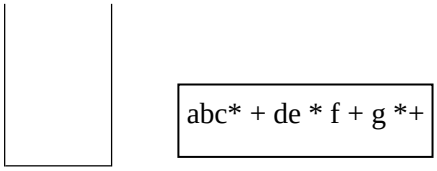
➢ Now we read a ')', so the stack is emptied back to the' We output a '+'



➢ Now read a '*' next; it is pushed onto the stack. Then g is read and output.



➢ The input is now empty, so we pop and output symbols from the stack until itempty.



4. Function Calls:

- The algorithm to check balanced symbols suggests a way to implement function calls.

- The problem here is that when a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables.

- Furthermore, the current location in the routine must be saved so that the new function knows where to go after it is done. When there is a function call, all the important information that needs to be saved such as register values and the return address is saved "on a piece of paper" in an abstract way and put at the top of a pile.
- Then the control is transferred to the new function, which is free to replace the registers with its valued. If it makes other function calls, it follows the same procedure.

- When the function wants to return, it looks at the "paper" at the top of the pile and restores all the registers. It then makes the return jump.

- Clearly, all of this work can be done using a stack, and that is exactly what happens in virtually every programming language that implements recursion. The information saved is called either an **activation record** or **stack frame**.

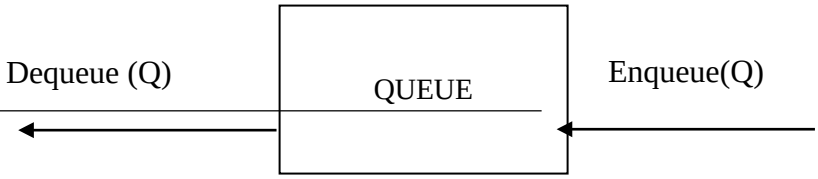
Queue ADT:

Queue is an ordered collection of data items. It delete item at front of the queue. It inserts item at rear of the queue. It has FIFO structure i.e. "**First In First Out".**

3	**5**	**2**	**7**	**6**

front **rear**

Queue Model:



The basic operations are,

- **Enqueue :**which inserts an element at the end of the list called **rear end**.
- **Dequeue:**Which deletes an element at the other end (front) of the list called **front end.**

Types of Queue:

- Simple Queue
- Circular Queue
- Double Ended Queue
- Priority Queue

Implementation of Simple Queue(Queue):

Like stack, queue can also be implemented in two methods.

- Using Array
- Using Linked list

1. Array Implementation of Queue(Simple Queue):

Implementation of Operations:

Global Declaration:

```
int *Queue, Front=0,Rear=-1,maxsize,i;
```

Explanation: The initial values of Front is zero and Rear is -1.

Global Declaration:

```
int *Queue, Rear=-1,front=-1,maxsize;
```

Code to Create Dynamic Queue:

Initially the Queue is declared as a pointer. Using create() function, once the maximum size is known the queue array can be dynamically created.

```
void create()
{
      printf("Enter the size of Queue:");
      scanf("%d",&maxsize);
      Queue=(int)malloc(sizeof(int)*maxsize);
}
```

Code to Check IsFull or IsEmpty:

A Dequeue on an empty queue or a Enqueue on a full queue will overflow the array bounds exception and cause a crash. To avoid this, the following functions are used.

```
int IsFull(int Queue[])
{
      Rear+1==maxsize;
}
int IsEmpty(int Queue[])
{
      return Rear==-1;
}
```

Enqueue(Insert) Operation : To insert element at Rear side.

```
void Enqueue(int Queue[],int x)
{
      if(IsFull(Queue))
            printf("\nQueue Overflow");
      else
      {     Rear++;
            Queue[Rear]=x;
      }
}
```

Explanation:

- The element to be inserted is passed to the function as argument x.
- If Rear reaches the maximum size of the Queue, it implies Queue is full.
- Otherwise rear is incremented and data is stored in Queue[rear].

Dequeue(Delete) Operation: To Dequeue or delete element at front.

```
void Dequeue(int Queue[])
{
      if(IsEmpty(Queue))
            printf("\nQueue underflow/empty");
      else
      front=front+1;
}
```

Explanation:

- If front is -1, it implies that there are no elements in the queue. Hence, Queue is empty.
- Otherwise, delete queue[front] and then shifts all the remaining elements into one position forward.
- Decreases Rear by 1.

Display Operation:

```
void display()
{
      int i;
      if(Rear==-1)
            printf("\n Queue is empty");
      else
      {
            printf("\nThe elements in the stack are \n");
            for(i=0;i<=Rear;i++)
                  printf("%d\t",Queue[i]);
      }
}
```

Explanation:

- If rear = -1, it implies that there are no elements in the queue and it is empty.
- Otherwise, display the elements one by one, from Front to Rear.

Disadvantages of Queue using array implementation:

- The size of the Queue is limited.
- If an element is deleted from the queue, all the remaining elements to be shifted by 1 position forward. This operation takes more cost.
- This problem can be overcome by Circular Queue.

Linked List implementation of Simple Queue:

```
struct Queue
{
      intdata;
      struct Queue *next;
};
struct Queue *queue,*front=NULL,*rear=NULL;
typedefstruct Queue *QUEUE;
QUEUEcreate()
{
      structqueue *temp;
      temp=(structqueue*)malloc(sizeof(structqueue));
      return temp;
}
```

The structure is declared gloabally so that all the functions can access the structure. Front and rearare initially made to NULL. It implies that there is no Queue created.

Enqueue Operation:

```
void  enqueue(int x)
{     int data;
      temp=create();
      temp->data=x;
      temp->next=NULL;
      if(front==NULL)
            Front=rear=temp;
      else
      {
            Rear->next=temp;
            rear=rear->next;
      }
}
```

Explanation: This function creates a new node called temp. The data is stored in the data part of the temp. If front==NULL, it implies that the queue is not created. The node temp becomes front and rear. The next contains NULL. Otherwise, insert the new node at rear->next and make temp as rear.

Dequeue (Deletion) operation:

```
void dequeue()
{
        if(front==NULL)
                printf("\nQueue is empty");
        else
                Front=front->next;
}
```

Explanation: This function deletes the topmost node in the stack. First it checks whether top is NULL. If yes, it implies that the stack does not contain any nodes. Otherwise, top is moved to top->next. It is only a logical deletion and not physical deletion

Display operation

```
void display()
{
        structqueue *ptr=NULL;
        if(front==NULL)
                printf("\nQueue is empty.");
        else
        {
                printf("\nThe elements in the Queue are \n");
                for(ptr=front;ptr!=NULL;ptr=ptr->next)
                        printf("%d\t",ptr->data);
        }
}
```

Explanation: This function displays all the elements in the data part of the nodes one after another.

Circular Queue:

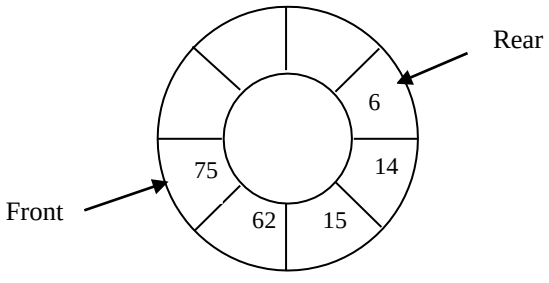
The drawbacks of simple queues are, time consuming and there is a chance to declare queue is full even if there is a empty space at front.

These can be overcome by circular queue.

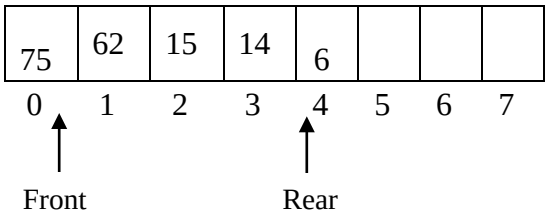
Circular queue is a wrap around queue, after insertion of last position of the queue, if there is an empty space at first, the insertion pointer will insert at first. ie., it won't show queue full message.

It is possible to insert new elements into the circular queue if the array slots at the beginning of the queue is empty.

Pictorial representation of Circular Queue:



Logical representation of Circular Queue:



Circular queue operations are,

- Enqueue(Insertion)
- Dequeue(Deletion)
- Display

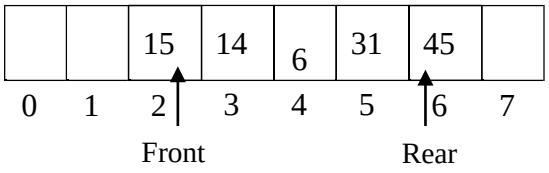
Global declarations:

```
int CQueue[100],front=-1,rear=-1,count=0,maxsize=99;
```

Function to Enqueue at Rear:

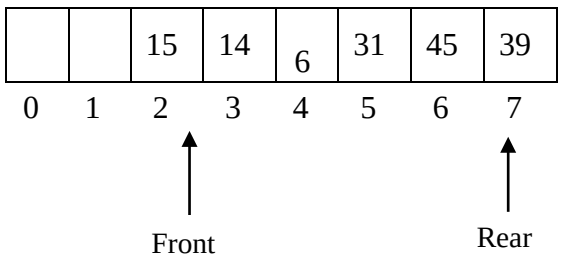
```
void enqueue_rear(int x)
{
      If(count==maxsize)
            Printf("Queue is full");
      Else
      {
            Rear=(rear+1)%maxsize;
            CQueue[rear]=x;
            Count++;
      }
}
```

Example:
Maxsize=8
Initial status of the queue is,



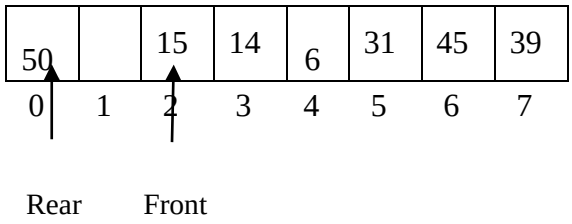
Enqueue 39.

Front=2, rear=(6+1)%8=>*7*



Enqueue 50

Front=2, rear=(7+1)%8=>0



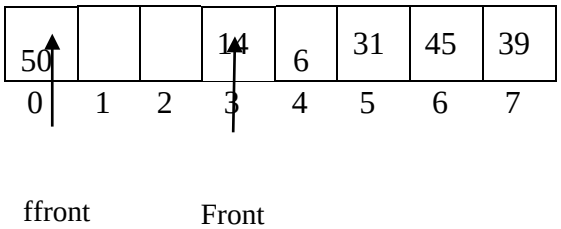
Function to dequeue at front:

```
void dequeue(int x)
{
        If(count==0)
                Printf("Queue is empty");
        Else
        {
                Front=(front+1)%maxsize;
                Count--;

        }
}
```

Example:
Dequeue,

Front=(front+1)%maxsize;



Function to display Queue elements:

```
Void display()
{
        int i;
        for(i=1;i<=count;i++)
        {
                printf(" %d ",CQueue[j]);
                j=(j+1)%maxsize;
        }
}
```

Double Ended Queue.

It is another type of queue and this is also called as Deque. A Deque is a special type of data structure in which insertions and deletions can be performed in both the ends. ie., at the front end and the rear end of the queue. There are 4 operations, 1. Insertion at front, 2. Insertion at rear, 3. Deletion at front, 4. Deletion at rear.

Function to enqueue at front:

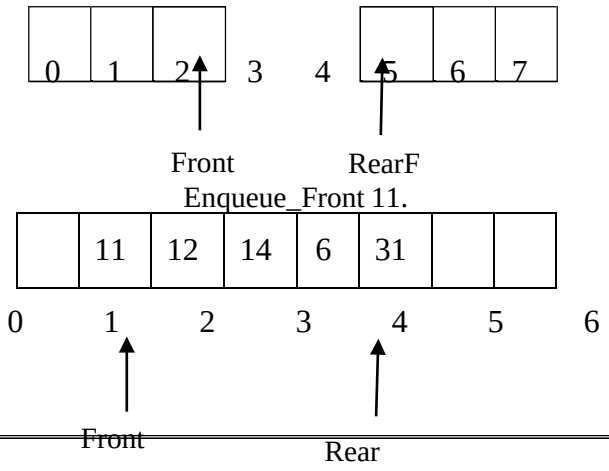
```
void enqueue_front(int x)
{       if(front==0)
                Printf("Insertion at front is not possible.");
        Else
        {


                Front=front-1;

                Deque[front]=x;
                Count++;
        }
```

7

Example:



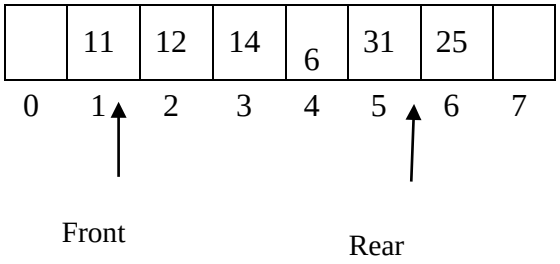
Function to enqueue at rear:

```
void enqueue_rear(int x)
{       if(rear==maxsize)
                printf("Insertion at rear is not possible:");
        else
        {

                Rear=rear+1;
                Deque[rear]=x;
                Count++;

        }
}
```

Enqueue_Rear 25

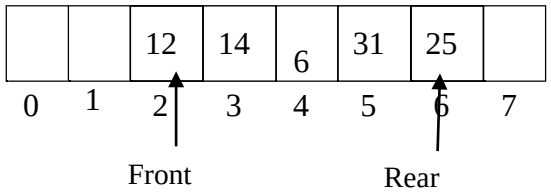


Function to dequeue at front:

```
Void dequeue_front()
{
        if(front>rear)
                Printf("Queue is empty");
        else
        {       Front=front+1;
                Count--;

        }
}
```

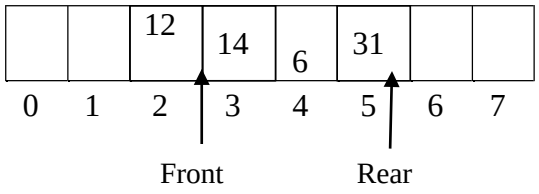
Dequeue_Front



Function to dequeue at rear:

```
Void dequeue_rear()
{
        if(front>rear)
                Printf("Queue is empty");
        Else
        {       Rear=rear-1;
                Count--;
                        }
}
```

Dequeue_Rear



Applications of Queues.

Print jobs

When jobs are submitted to a printer, they are arranged in the order they arrive. Then jobs sent to a line printer are placed on a queue.

Computer networks

There are many network setup of personal computers in which the disk is attached to one machine called file server. Users on other machines are given access to files on a first come first served basis where the data structure is queue.

OS Operating system performs various tasks namely memory management, CPU management etc. The operating system follows a technique called **First Come Fist Serve (FCFS)** to allocate CPU for the waiting processes.

Real-life waiting lines:

i. Calls to large companies are generally placed on a Queue when all the operators are busy.

ii. In large Universities, where resources are limited, students must sign a waiting list if all terminals are occupied.

Mathematics:

There is a branch of Mathematics called Queuing Theory, which deals with computing, probabilistically, how long users expect to wait on a line.