STACK

A stack is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added /inserted and from which items may be deleted at only one end, called the top of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called

Last-in-First-out (LIFO)

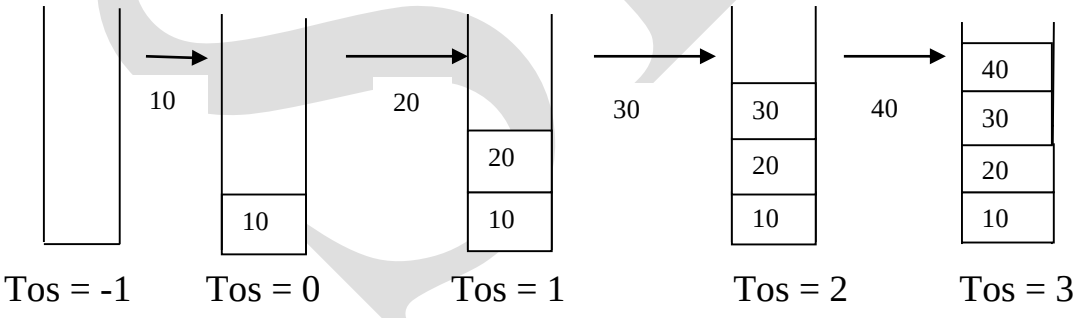
Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.

1) PUSH OPERATION:

Procedure of adding new element at top of stack (tos) is called push operation. Suppose after adding element maximum size is reach i.e. stack is full and if you want to add more element then it is not possible. This situation is called stack overflow.

Algorithm for Push Operation

1. Start
2. if (tos = (max -1))
   then PRINT "stack overflow"
3. Else read data
4. tos = tos+1
5. stack[tos]=data
6. Stop

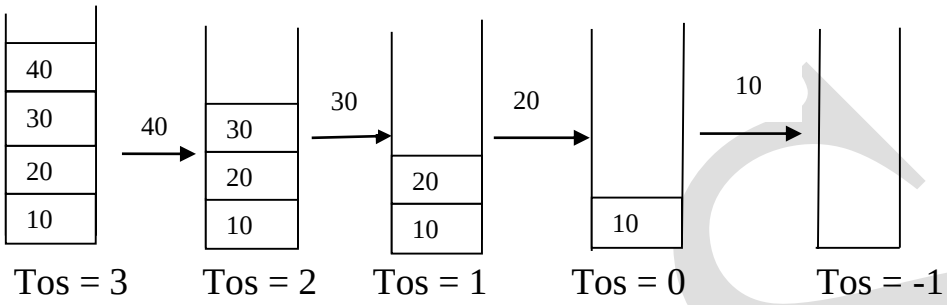


2) POP OPERATION

The procedure for removing element from top of stack is called pop operation. Every time when we delete element tos is decremented by 1. When all elements are deleted then tos points to the bottom of stack then it is not possible to delete more element .This situation is called stack underflow.

ALGORITHM FOR POP OPERATION

1. Start
2. if (tos = -1)
   then PRINT "stack underflow"
3. else stack[tos]=NULL
4. tos=tos -1
5. Stop



C Program (using array)

```
#include<stdio.h>
#include<conio.h>
#define max 20
int stack[max], tos;
void push();
void pop();
void display();
void main()
{
  int ch;
  clrscr();
  tos=-1;
  printf("Stack operation");
  while(ch!=4)
  {
    printf("\n1. PUSH \t 2. POP \t 3. DISPLAY 4. EXIT  ");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1: push();
                break;
        case 2: pop();
                break;
        case 3: display();
                break;
        case 4: exit(0);
        default: printf("Enter Correct choice");
    }
```

```
   }
   getch();
}

void push()
{
  int no;
  if(tos==max-1)
    printf("Stack overflow");
  else
  {
    tos++;
    printf("Enter element ");
    scanf("%d",&no);
    stack[tos]=no;
    printf("Node Inserted");
  }
}

void pop()
{
 if(tos==-1)
   printf("Stack underflow");
 else
 {
   printf("Element %d is pop",stack[tos]);
   tos--;
 }
}

void display()
{
 int i;
 if(tos==-1)
  printf("Stack is empty");
 else
  for(i=tos;i>=0;i--)
    printf("\nElement= %d",stack[i]);
}
```

Application of Stack

Application of stack are

1. Recursion
2. Checking validity of an arithmetic expression
3. Expression conversion & evaluation
4. Decimal to binary conversion
5. Reversing string.
6. Storing function call

Decimal to binary conversion

Stack is used to convert decimal number to binary.

C program

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int n, tos=-1;
  int stack[20];
  clrscr();

  printf("Enter decimal number: ");
  scanf("%d",&n);

  printf("Binary conversion of %d number is: ",n);

  while(n>0)
  {
    tos++;
    stack[tos]=n%2;
    n=n/2;
  }
  while(tos!=-1)
        printf("%d",stack[tos--]);
  getch();
}
```

Polish Notation

Polish notation is a way of expressing arithmetic expressions that avoids the use of brackets or parenthesis to define priorities for evaluation of operators. Polish notation was devised by the polish philosopher and mathematician Jan Lukasiewicz for use in symbolic logic.

In this notation the operators precede their operands, so it is also known as prefix expression.

Eg if infix expression is (3 + 5) * (7 – 2)
Then in polish notation / prefix from it can be written as *+35-7-2

The Reversed form “Reverse Polish Notation” has however been found more convenient form a computational point of view. In RPN the operands of an operator precedes it is also known as postfix notation. Both prefix & postfix i.e. polish & Reverse polish notation expression are parenthesis free. Above expression can be written in postfix form as 3 5 + 7 2 - *

In all forms the operands occur in the same order and just the operators have to be moved to keep the meaning correct.

Infix to postfix conversion by using stack.

Precedence/ priority of operators

1. ^ (Exponential) Highest
2. Multiplication & Division.
3. Addition & Subtraction.

Algorithm

1. Scan the infix expression from left to right
2. If scanned character is an operand print it to postfix expression
3. If scanned character is an operator then check
    a. If scanned operator has higher precedence than the operator on top of the stack then push it onto stack.
    b. If scanned operator is of lower or equal precedence than operator on top of the stack then pop stack top operator & print it to postfix expression & go to step 3.
4. If scanned symbol is a left parenthesis then push it onto a stack.
5. If scanned symbol is a right parenthesis then pop out all the operators of the stack to postfix expression until left parenthesis is found, then pop left parenthesis.
6. If you are at end of expression then go to step 7 otherwise scan next symbol & go to step 2
7. Pop stack operator until stack is empty & print it to postfix expression.
8. Stop.

Example

Infix Expression: A + B * C

Infix Expression	Stack	Postfix Expression
A + B * C $	$	
+ B * C $	$	A
B * C $	+ $	A
* C $	+ $	A B
C $	* + $	A B
$	* + $	A B C
$	+ $	A B C *
$	$	A B C * +

Postfix Expression: A B C * +

Infix Expression: (A + B) / (C – D)

Infix Expression	Stack	Postfix Expression
(A + B) / (C – D)$	$	
A + B) / (C – D)$	($	
+ B) / (C – D)$	($	A
B) / (C – D)$	+ ($	A
) / (C – D) $	+ ($	A B

/ (C – D)$	$	A B +
(C – D) $	/ $	A B +
C – D)$	(/ $	A B +
– D)$	(/ $	A B + C
D)$	- (/ $	A B +
)$	(/ $	A B + C D
$	/ $	A B + C D -
$	$	A B + C D - /

Postfix Expression: A B + C D - /

Infix to postfix conversion

```
#include<stdio.h>
#include<conio.h>

char stack[30];
int tos=-1;
void itop(char*);
void push(char);
char pop();
void main()
{
	char infix[30];
	clrscr();
	printf("\nEnter the Infix expression: ");
	gets(infix);
	itop(infix);
	getch();
}

void push(char sym)
{
	if(tos>=29)
	{
		printf("\nStack overflow..");
		return;
	}
	else
	{
		tos=tos+1;
		stack[tos]=sym;
	}
}
char pop()
{
	char i;
```

```
	if(tos==-1)
	{
		printf("\nStack underflow..");
		getch();
		return;
	}
	else
	{
		i=stack[tos];
		tos=tos-1;
	}
	return i;
}

int prec(char ch)
{
	if(ch=='^')
	{
		return 5;
	}
	else if(ch=='*' || ch=='/')
	{
		return 4;
	}
	else if(ch=='+' || ch=='-')
	{
		return 3;
	}
	else
	{
		return 2;
	}
}

void itop(char infix[])
{
	int length;
	static int index=0,pos=0;
	char symbol,temp;
	char postfix[50];

	length=strlen(infix);

	while(index<length)
	{
		symbol=infix[index];
		switch(symbol)
		{
			case '(':push(symbol);
```

```
                        break;
                case ')':temp=pop();
                        while(temp!='(')
                        {
                                postfix[pos]=temp;
                                pos++;
                                temp=pop();
                        }
                        break;
                case '+':
                case '-':
                case '*':
                case '/':
                        while(prec(stack[tos]) >= prec(symbol))
                        {
                                temp=pop();
                                postfix[pos]=temp;
                                pos++;
                        }
                        push(symbol);
                        break;
                default: postfix[pos++]=symbol;
                        break;
        }
        index++;
    }
    while(tos>=0)
    {
        temp=pop();
        postfix[pos++]=temp;
    }
    postfix[pos++]='\0';
    puts(postfix);
    return;
}
```

Evaluation of postfix expression

Algorithm

1. Scan postfix expression from left to right
2. If scanned character is operand then push it onto an operand stack.
3. If scanned character is an operator then pop two operands from stack apply that operator evaluates the result and push that result onto operand stack.
4. If you are at end of expression then go to step 5 otherwise go to step 1
5. Stop.

Example Evaluate example 5 6 2 + * 8 4 / -

Postfix Expression	Stack	Operation
5 6 2 + * 8 4 / - $	$	
6 2 + * 8 4 / - $	5 $	
2 + * 8 4 / - $	6 5 $	
+ * 8 4 / - $	2 6 5 $	
* 8 4 / - $	5 $	6 + 2 = 8
* 8 4 / - $	8 5 $	
8 4 / - $	$	5 * 8= 40
8 4 / - $	40 $	
4 / - $	8 40 $	
/ - $	4 8 40 $	
- $	40 $	8 / 4 = 2
- $	2 40 $	
$	$	40 – 2 = 38
$	38 $	

Evaluation of **5 6 2 + * 8 4 / -** is **38**

C program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

void main()
{
  int n, i, j, tos=-1, x=0, y=0;
  int stack[20];
  char str[20];
  clrscr();

  printf("Enter valid postfix expression: ");
  scanf("%s",&str);

  n=strlen(str);

  for(i=0; i<n; i++)
  {
    switch(str[i])
    {
        case '+': y=stack[tos--];
                  x=stack[tos--];
                  x=x+y;
```

```
                stack[++tos]=x;
                break;

        case '-': y=stack[tos--];
                x=stack[tos--];
                x=x-y;
                stack[++tos]=x;
                break;
        case '*': y=stack[tos--];
                x=stack[tos--];
                x=x*y;
                stack[++tos]=x;
                break;
        case '/': y=stack[tos--];
                x=stack[tos--];
                x=x/y;
                stack[++tos]=x;
                break;

        case '^': y=stack[tos--];
                x=stack[tos--];
                for(j=1;j<y;j++)
                        x=x*x;
                stack[++tos]=x;
                break;

        default : if(str[i]>=48 && str[i]<=57)
                {
                        x=str[i]-48;
                        stack[++tos]=x;
                }
    }
  }
  printf("Value of %s expression is : %d ",str, stack[tos]);
  getch();
}
```

Infix to prefix conversion

Algorithm

1. Scan the expression from right to left
2. If the scanned character is ‘)’ then push it onto stack.
3. If the scanned character is ‘(‘ then pop all the operator until ‘)’ is reached.
4. If scanned character is operand then add it to output expression.
5. If scanned character is an operator then
    a. If scanned operator is of higher or equal precedence than stack top operator then push it onto a stack.

b. If scanned operator is of lower precedence than stack top operator then pop stack top operator & then go to step 5.

6. Reverse output expression.

Example

Infix Expression: A + B * C

Infix Expression	Stack	Postfix Expression
$ A + B * C	$	
$ A + B *	$	C
$ A + B	* $	
$ A +	* $	C B
$ A	+ $	C B *
$	+ $	C B * A
$	$	C B * A +

Prefix Expression: + A * B C

Infix Expression: (A + B) / (C – D)

Infix Expression	Stack	Postfix Expression
$(A + B) / (C – D)	$	
$(A + B) / (C – D	) $	
$(A + B) / (C –	) $	D
$(A + B) / (C	-) $	D
$(A + B) / (	-) $	D C
$(A + B) /	$	D C -
$(A + B)	/ $	D C -
$(A + B	) / $	D C -
$(A +	) / $	D C - B
$(A	+) / $	D C - B
$(	+) / $	D C – B A
$	/ $	D C – B A +
$	$	D C – B A + /

Prefix Expression: / + A B – C D

Recursion:

A recursive function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call.

Recursion is like a top-down approach to problem solving; it divides the problem into pieces or selects one key step, postponing the rest. On the other hand, iteration is more of a bottom-up approach; it begins with what is known and from this constructs the solution step by step.

What is needed for implementing recursion?

- Decomposition into smaller problems of same type
- Necessity of stopping Condition
- It acts as a terminating condition. Without as explicitly defined stopping Condition, a recursive function would call itself indefinitely.
- It is the building block to the complete solution. In a sense, a recursive function determines its solution from the stopping Condition(s) it reaches.

The recursive algorithms will generally consist of an if statement with the following form:

```
return_data_type func_name(data_type varible1….)
{
        if (Stopping_Condition) then
                solve it directly
        else
                return func_name(variable1,…).
}
```

Eg.

```
int fact(int n)
{
        if(n<=1)
                return 1;
        else
                return(n*fact(n-1));
}
```

When we find out the solution of a problem using recursion, in that stack is used. When new function is called then previous function is push into stack. Whenever function execution is completed then it pop out from stack and continue execution.

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. It is a classical recursion problem. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the pole and placing it on top of another pole i.e. a disk can only be moved if it is the uppermost disk on a pole.

3. No disk may be placed on top of a smaller disk.
4. Only one auxiliary pole should be use.

C Program

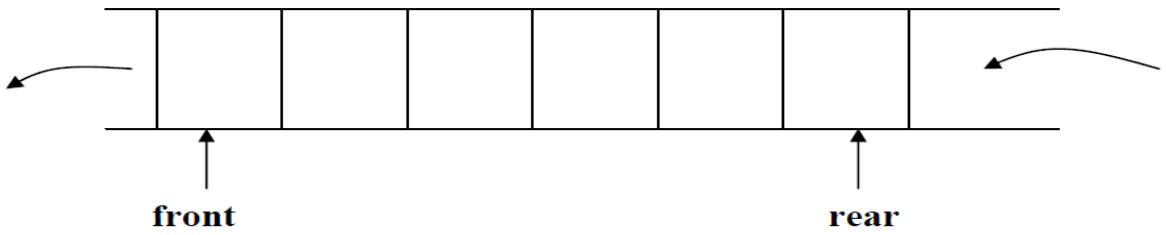
```
#include<stdio.h>
#include<conio.h>
void Tower(int,char,char,char);
int` step=0;
void main()
{
        int num;
        clrscr();
        printf("\nplease,enter no. of disk: ");
        scanf("%d",&num);
        printf("\nStart Tower of Hanoi..");
        Tower(num,'A','B','C');
        printf("\nProgram exits here...");
        getch();
}
void Tower(int n,char s,char d,char a)
{       if(n==1)
        {
                printf("\nStep %d:disk move from %c to %c",++step,s,d);
        }
        else
        {
                Tower(n-1,s,a,d);
                printf("\nStep %d:disk move from %c to %c",++step,s,d);
                Tower(n-1,a,d,s);
        }
}
```

Queue

A queue is simply a waiting line that grows by adding elements to its end and shrinks by removing elements from the front. A formal definition of queue as a data structure: It is a list from which items may be deleted at one end (front) and into which items may be inserted at the other end (rear). It is also referred to as a first-in-first-out (FIFO) data structure.

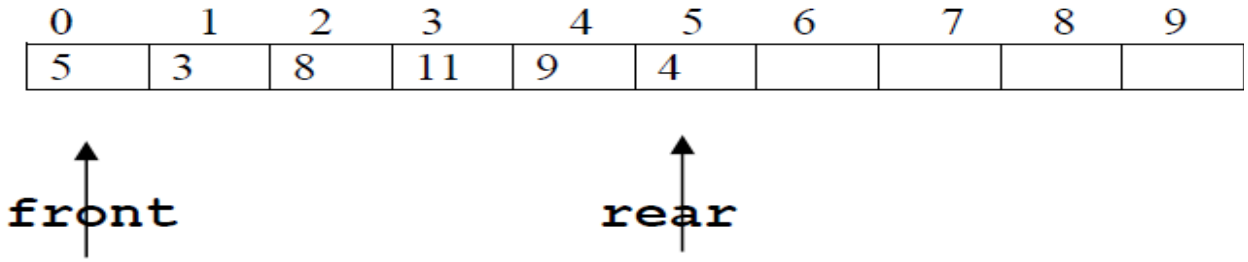
first in first out (FIFO)

Waiting lines in supermarkets, banks, food counters are common examples of queues. Queue has many applications in computer systems: − Handling jobs in a single processor computer − print spooling − transmitting information packets in computer networks.



Array Implementation

The array to implement the queue would need two variables (indices) called *front* and *rear* to point to the first and the last elements of the queue



Initially:

rear = -1; front = -1;

For each insert operation *rear* is incremented by one, and for each delete operation, *front* is incremented by one. While the insert and delete operations are easy to implement, there is a big disadvantage in this set up. The size of the array needs to be huge, as the number of slots would go on increasing as long as there are items to be added to the list

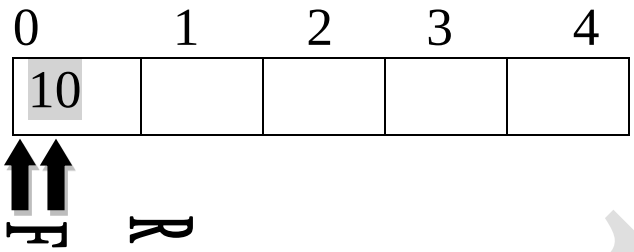
Operation performed on a queue

1. Insertion
2. Deletion

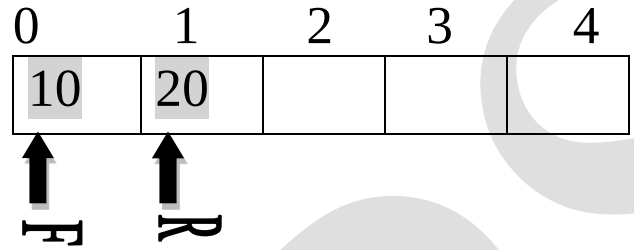
INSERTION:

Initially Front=Rear=-1

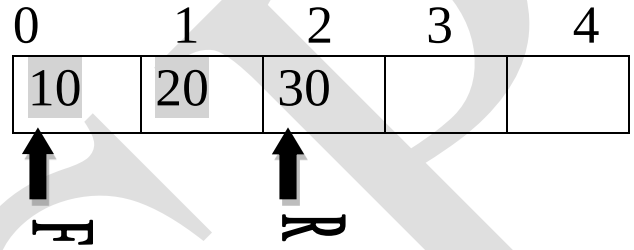
Insert



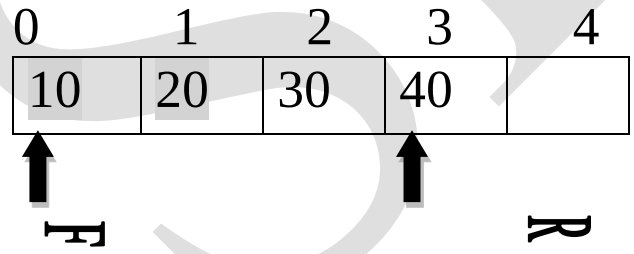
Insert



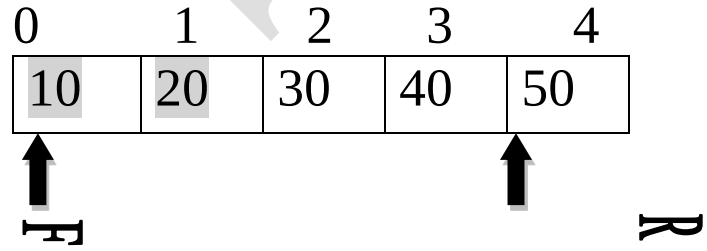
Insert



Insert



Insert



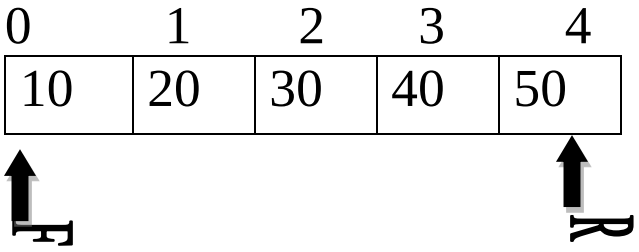
Insert

Overflow :

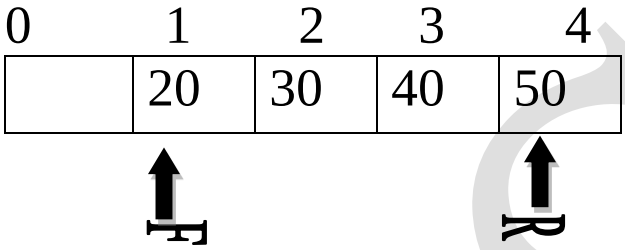
When rear =Imax-1…… (Error “overflow”)

DELETION:

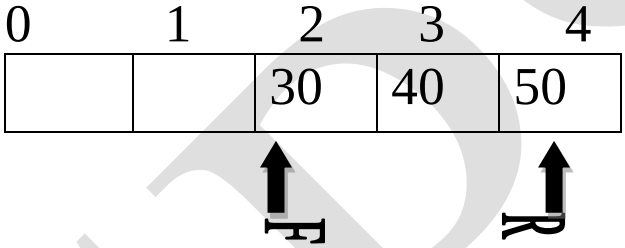
Initially



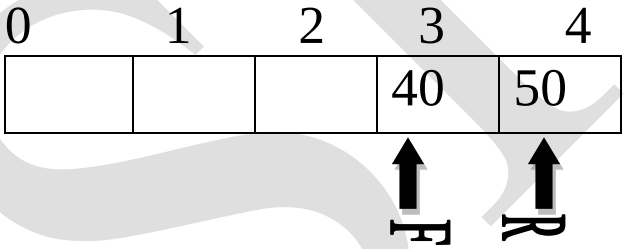
Delete:



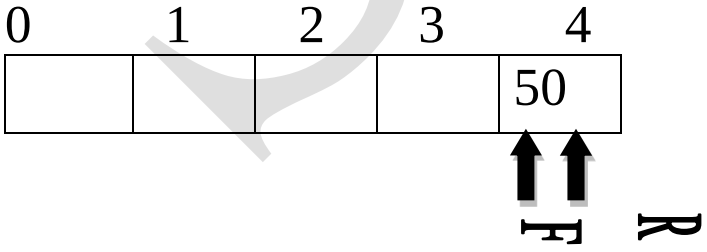
Delete:



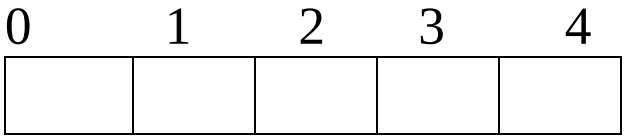
Delete:



Delete:



Delete:



Underflow:

Delete

When Front = =-1…… (Error underflow)

ALGORITHM:

FOR INSERTION

1. If Rear=max-1
   then print "Queue is overflow"
   goto step 5
2. Read data
3. if (Front= -1)
   then Front=Front+1
   & Rear=Rear+1;
   else Rear=Rear+1
4. Queue[Rear]=data
5. Stop

FOR DELETION:

1. if (Front =-1)
   then print "Queue is underflow"
   goto step 3
2. if (Front=Rear)
   then Queue[Front]=NULL;
   Front=Rear=-1;
   else Q[Front]=NULL;
   Front= Front+1;
3. Stop

C Program (Using Array)

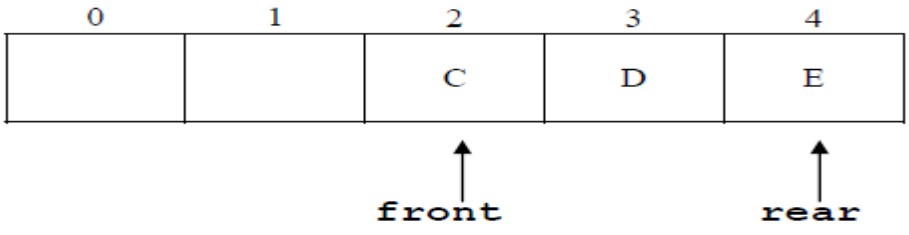
```
#include<stdio.h>
#include<stdlib.h>
#define max 20

int front=-1,rear=-1;
int queue[max];

void insert();
void delet();
void display();

void main()
{
        int ch=0;
        while(ch!=4)
        {
```

```
                printf("\n****MAIN MENU****");
                printf("\n1.INSERT");
                printf("\n2.DELETE");
                printf("\n3.DISPLAY");
                printf("\n4.EXIT");
                printf("\nEnter Your Choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: insert();
                                break;
                        case 2: delet();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(0);
                        default:printf("\nEnter Proper Choice..");
                }
        }
        printf("\nProgram exits here..");
}

void insert()
{
   int data;
        if(rear==max-1)
        {
                printf("\nQueue Overflow.....");
        }
        else
        {
                printf("\nEnter integer value: ");
                scanf("%d",&data);
                if(front<0)
                {
                        front=0;
                        rear=0;
                }
                else
      {
         rear++;
      }
      queue[rear]=data;
                printf("\nOne value inserted");
        }
}

void delet()
{
```

```
        if(front==-1)
        {
                printf("\nQueue Underflow....");
        }
        else
        {
           printf("\nElement %d is deleted..",queue[front]);
           if(front==rear)
      {
        front=-1;
        rear=-1;
      }
      else
      {
        front++;
      }
        }
}

void display()
{
        int i;
        if(rear==-1)
        {
                printf("\nQueue is an empty");
        }
        else
        {
                printf("\nQueue elements are...\n");
                for(i=front;i<=rear;i++)
                {
                        printf("%d ",queue[i]);
                }
        }
}
```

Problems with this representation:

Although there is space in the following queue, we may not be able to add a new item. An attempt will cause an overflow.

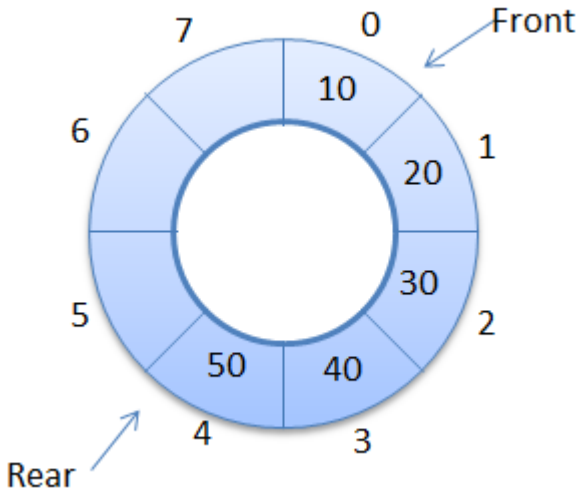


Circular Queue

Let us now imagine that the above array is wrapped around a cylinder, such that the first and last elements of the array are next to each other. When the queue gets apparently full, it can continue to store elements in the empty spaces in the beginning of the array. This makes efficient use of the array slots. It is also referred to as a circular array. This enables us to utilize the unavailable slots, provided the indices *front* and *rear* are handled carefully.

Circular queue using Array :-

In circular queue rear gets to the end of the array, it is wrapped around to the beginning. Now, the array can be thought of as a circle. The first position follows the last element.



ALGORITHM:-

INSERT:

Step 1: START
Step 2: if (Front = (Rear+1)%max)
Then PRINT "queue overflow"
& go to step 7.
Step 3: Read element (E);
Step 4: if(Front = -1)
Front++;
Step 5: Rear = (Rear+1)% max
Step 6: Queue[Rear]=E;
Step 7: STOP

DELETE:

Step 1: START
Step 2: if (Front = -1)
PRINT "Queue underflow"
& go to step 4.
Step 3: if (Front = Rear)
Front = Rear =-1
else
Front = (Front + 1) % max;

Step 4: STOP

C Program

```
#include<stdio.h>
#include<stdlib.h>
#define max 3

int front=-1,rear=-1;
int queue[max];

void insert();
void delet();
void display();

void main()
{
        int ch=0;
        while(ch!=4)
        {
                printf("\n****MAIN MENU****");
                printf("\n1.INSERT");
                printf("\n2.DELETE");
                printf("\n3.DISPLAY");
                printf("\n4.EXIT");
                printf("\nEnter Your Choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: insert();
                                break;
                        case 2: delet();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(0);
                        default:printf("\nEnter Proper Choice..");
                }
        }
        printf("\nProgram exits here..");
}

void insert()
{
   int data;
        if(front==(rear+1)%max)
        {
```

```
                printf("\nQueue Overflow.....");
        }
        else
        {
                printf("\nEnter integer value: ");
                scanf("%d",&data);
                if(front==-1)
                {
                        front=0;
                        rear=0;
                }
                else
    {
       rear=(rear+1)%max;
    }
    queue[rear]=data;
                printf("\nOne value inserted");
        }
}

void delet()
{
        if(front==-1)
        {
                printf("\nQueue Underflow....");
        }
        else
        {
           printf("\nElement %d is deleted..",queue[front]);
           if(front==rear)
    {
       front=-1;
       rear=-1;
    }
    else
    {
       front=(front+1)%max;
    }
        }
}

void display()
{
        int f,r;
        if(rear==-1)
        {
                printf("\nQueue is an empty");
        }
        else
```

```
        {
                f=front;
                r=rear;

                printf("\nQueue elements are...\n");
                while(f!=r)
      {
                        printf("%d ",queue[f]);
        f=(f+1)%max;
                }
                printf("%d",queue[f]);
        }
}
```

Priority Queue

A queue in which we can insert items at any position depending on some priority is known as priority queue.

There are two types of priority queue

1. Ascending priority queue
2. Descending priority queue

1. **Ascending priority queue:**
   It is a collection of items in which items can be inserted arbitrarily but only smallest element can be removed

2. **Descending priority queue**
   It is a collection of items in which items can be inserted arbitrarily but only largest element can be removed

C Program

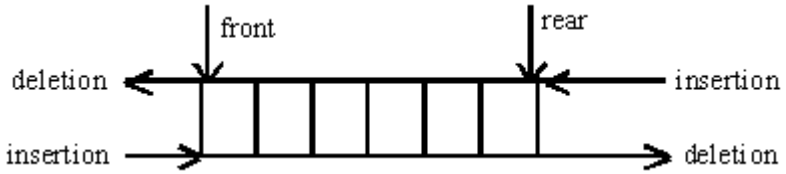
```
#include<stdio.h>
#include<stdlib.h>
#define max 5

int front=-1,rear=-1;
int queue[max];

void insert();
void delet();
void display();

void main()
{
        int ch=0;
        while(ch!=4)
```

```
        {
                printf("\n****MAIN MENU****");
                printf("\n1.INSERT");
                printf("\n2.DELETE");
                printf("\n3.DISPLAY");
                printf("\n4.EXIT");
                printf("\nEnter Your Choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: insert();
                                break;
                        case 2: delet();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(0);
                        default:printf("\nEnter Proper Choice..");
                }
        }
        printf("\nProgram exits here..");
}

void insert()
{
   int data,j;
        if(rear==max-1)
        {
                printf("\nQueue Overflow.....");
        }
        else
        {
                printf("\nEnter integer value: ");
                scanf("%d",&data);
                j=rear;
                while(j>=0&&data<queue[j])
      {
         queue[j+1]=queue[j];
         j--;
      }
                if(front<0)
                {
                        front=0;
                        rear=0;
                }
                else
      {
         rear++;
      }
```

```
        queue[j+1]=data;
                printf("\nOne value inserted");
           }
}

void delet()
{
          if(front==-1)
          {
                printf("\nQueue Underflow....");
          }
          else
          {
             printf("\nElement %d is deleted..",queue[front]);
             if(front==rear)
        {
          front=-1;
          rear=-1;
        }
        else
        {
          front++;
        }
          }
}

void display()
{
          int i;
          if(rear==-1)
          {
                printf("\nQueue is an empty");
          }
          else
          {
                printf("\nQueue elements are...\n");
                for(i=front;i<=rear;i++)
                {
                        printf("%d ",queue[i]);
                }
          }
}
```

Deque (Double Ended Queue)

Deque is a data structure in which elements may be added to or deleted from the front or the rear. Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: ins_Front, ins_Rear, del_front, del_Rear. Deque can be behave like a queue by using only ins_Rear and del_front , and behaves like a stack by using only ins_Rear and del_Rear. .
The Deque is represented as follows.



Deque can be represented in two ways they are

1) Input restricted Deque
2) Output restricted Deque

The output restricted Deque allows deletions from only one end and input restricted Deque allow insertions at only one end.

1) Input restricted Deque
   ins_Rear(), del_front(), del_Rear()
2) Output restricted Deque
   ins_Front(), ins_Rear(), del_front()

Algorithm for Deque :

Insert at front

step1. Start

step2. Check the queue is full or not as if (front==rear+1)

step3. If false update the pointer front as front= front-1

step4. Insert the element at pointer front as Q[front] = element

step5. Stop

Insert at rear

step1. Start

step2. Check the queue is full or not as if (rear == max-1) if yes queue is full

step3. If false update the pointer rear as rear= rear+1

step4. Insert the element at pointer rear, enter element and assign value

step5. Stop

Delete from front

step1. Start

step2. Check the queue is empty or not as if (front == -1) if yes queue is empty

step3. If (front== rear) reset pointer front and rear as front=rear=-1

step4. If false delete element at position front as element = Q[front] and update pointer front as front = front+1

step5. Stop

Delete from rear

step1. Start

step2. Check the queue is empty or not as if (front == -1) if yes queue is empty

step3. If (front== rear) reset pointer front and rear as front= rear= -1

step4. If false delete element at position rear as element = Q[rear]

step5. Update pointer rear as rear = rear-1

step6. Stop

C program (using array).

```
#include<stdio.h>
#define max 20

int front=-1,rear=-1;
int queue[max];

void ins_front();
void ins_rear();
void del_front();
void del_rear();
void display();

void main()
{
        int ch=0;
        while(ch!=6)
        {
                printf("\n****MAIN MENU****");
                printf("\n1.Insert at front");
                printf("\n2.insert at rear");
                printf("\n3.Delete from front");
                printf("\n4.Delete from rear");
                printf("\n5.DISPLAY");
                printf("\n6.EXIT");
```

```
                printf("\nEnter Your Choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: ins_front();
                                break;
                        case 2: ins_rear();
                                break;
                        case 3: del_front();
                                break;
                        case 4: del_rear();
                                break;
                        case 5: display();
                                break;
                        case 6: exit(0);
                        default:printf("\nEnter Proper Choice..");
                }
        }
}

void ins_front()
{
 if(front==(rear+1)%max)
    printf("Queue is full");
 else
 {
  if(front==-1)
  {
   front=0;
   rear=0;
  }
  else
  {
   if(front==0)
        front=max-1;
    else
        front=front-1;
  }
  printf("\nEnter element");
  scanf("%d",&queue[front]);
 }
}

void ins_rear()
{
 if(front==(rear+1)%max)
   printf("Queue is full");
 else
 {
```

```
    if(front==-1)
       front=0;
    rear=(rear+1)%max;
    printf("\nEnter element");
    scanf("%d",&queue[rear]);
  }
}

void del_front()
{
  if(front==-1)
    printf("Queue is empty");
  else
  {
    printf("\nElement %d is deleted",queue[front]);
    if(front==rear)
    {
      front=-1;
      rear=-1;
    }
    else
      front=(front+1)%max;
  }
}

void del_rear()
{
  if(front==-1)
    printf("Queue is empty");
  else
  {
    printf("\nElement %d is deleted",queue[rear]);
    if(front==rear)
    {
      front=-1;
      rear=-1;
    }
    else
    {
      if(rear==0)
            rear=max-1;
      else
            rear=rear-1;
    }
  }
}

void display()
{
```

```
  int f,r;
  f=front;
  r=rear;
  if(front==-1)
    printf("Queue is empty");
  else
  {
    printf("Queue elements are");
    while(f!=r)
    {
      printf("%d ",queue[f]);
      f=(f+1)%max;
    }
    printf("%d ", queue[f]);
  }
}
```

Applications of Queue

- Queue is useful in CPU scheduling, Disk Scheduling. When multiple processes require CPU at the same time, various CPU scheduling algorithms are used which are implemented using Queue data structure.
- When data is transferred asynchronously between two processes. Queue is used for synchronization. Examples: IO Buffers, pipes, file IO, etc.
- In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate. Spooling also lets you place a number of print jobs on a queue instead of waiting for each one to finish before specifying the next one.
- Breadth First search in a Graph. It is an algorithm for traversing or searching graph data structures. It starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
- In real life, Call Centre phone systems will use Queues, to hold people calling them in an order, until a service representative is free.