UNIT II

Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- Stack.
- Queue.

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

STACK:

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. Stacks are sometimes known as LIFO (last in, first out) lists.

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.

The three basic operations associated with stacks are:

- *Push*: is the term used to insert an element into a stack.
- *Pop*: is the term used to delete an element from a stack.
- *Peep*: is the term used to return the top most element of a stack.

"Push" is the term used to insert an element into a stack. "Pop" is the term used to delete an element from the stack.

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:

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.

When an element is added to a stack, the operation is performed by push(). Figure 2.1 shows the creation of a stack and addition of elements using push().

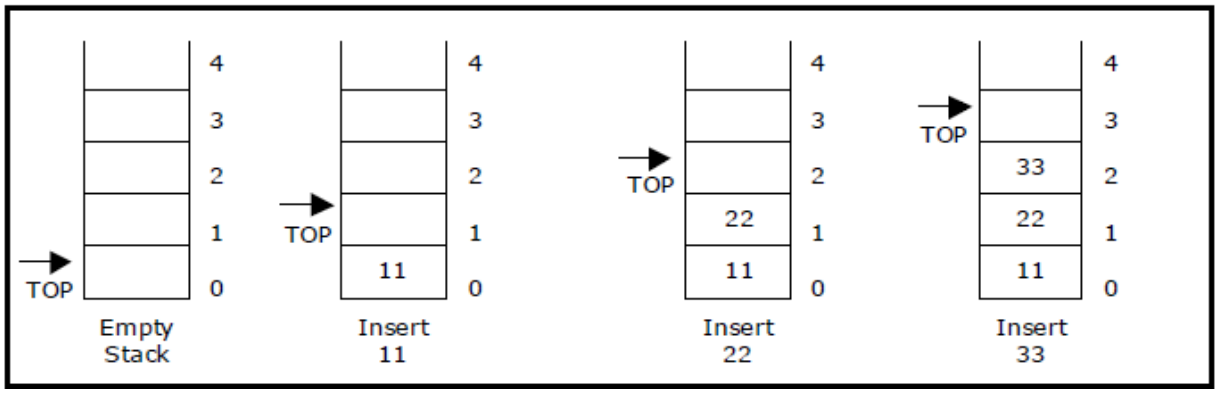
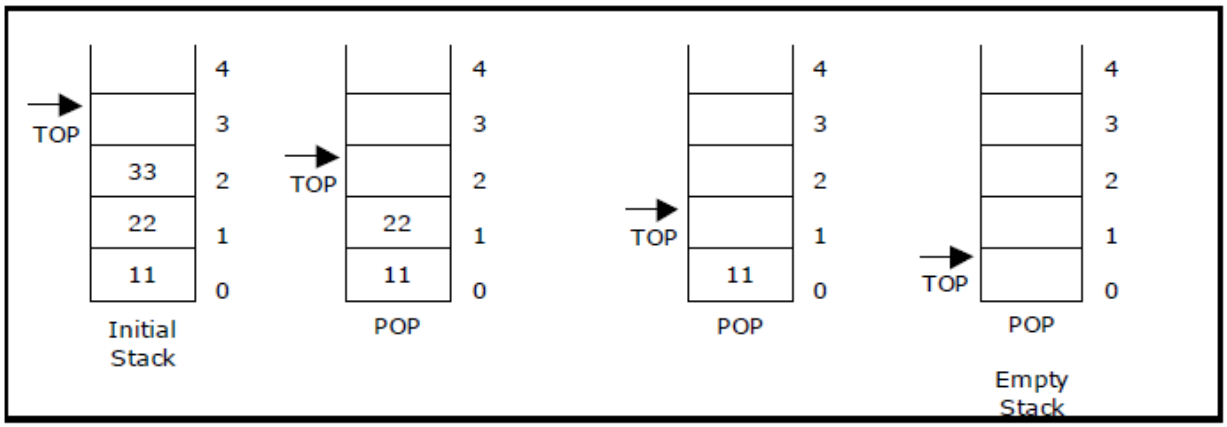


Figure:Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure shows a stack initially with three elements and shows the deletion of elements using pop().



Fugure:Pop operations on stack

Source code for stack operations, using array:

```
 # include <stdio.h>
 # include <conio.h>
 # include <stdlib.h>
 # define MAX 6 intstack[MAX];

int top = 0;

int menu()
 {
        intch;
        clrscr();
        printf("\n … Stack operations using ARRAY... ");
        printf("\n ----------**********----------\n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
```

```
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        returnch;
}

void display()
 {
        int i;
        if(top == 0)
        {
                printf("\n\nStack empty..");
        }
        else
        {
                printf("\n\nElements in stack:");
                for(i = 0; i < top; i++)
                        printf("\t%d", stack[i]);
        }
        }

void pop()
{
        if(top == 0)
        {
                printf("\n\nStack Underflow..");

                return;
        }
        else
                printf("\n\npopped element is: %d ", stack[--top]);
}

void push()
{

        int data;
        if(top == MAX)

        {
                printf("\n\nStack Overflow..");
                return;
        }
        else
        {
                printf("\n\nEnter data: ");
                scanf("%d", &data);
                stack[top] = data;
                top = top + 1;
                printf("\n\nData Pushed into the stack");

        }
}
void main()
{
        intch;
```

```
        do
        {
                ch = menu();
                switch(ch)
                {
                case 1:
                        push();
                        break;
                case 2:
                        pop();
                        break;
                case 3:
                        display();
                        break;
                case 4: exit(0);
                }
                getch();
        } while(1);
}
```

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using*top* pointer.

Source code for stack operations, using linked list:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct stack
{
        int data;
        struct stack *next;
};
struct stack *top=NULL;
struct stack *push(struct stack *,int);
struct stack *pop(struct stack *);
struct stack *display(struct stack *);
void main()
{
        intvalue,option;
        clrscr();
        do
        {
                printf("\n ******** MAIN MENU ********");
                printf("\n\t 1. PUSH ");
```

```
                printf("\n\t 2. POP ");
                printf("\n\t 3. DISPLAY ");
                printf("\n\t 4. EXIT ");
                printf("\n \n Enter your option :");
                scanf("%d",&option);
                switch(option)
                {
                case 1:
                        printf("\n Enter the value to be pushed on the stack ");
                        scanf("%d",&value);
                        top=push(top,value);
                        break;
                case 2:
                        top=pop(top);
                        break;
                case 3:
                        top=display(top);
                        break;
                }
        }while(option!=4);
        getch();
}
struct stack *push(struct stack *top,int value)
{
        struct stack *ptr;
        ptr=(struct stack*)malloc(sizeof(struct stack));
        ptr->data=value;
        if(top==NULL)
        {
                ptr->next=NULL;
                top=ptr;
        }
        else
        {
                ptr->next=top;
                top=ptr;
        }
        return top;
}
struct stack *pop(struct stack *top)
{
        struct stack  *ptr;
```

```
        ptr=top;
        if(top==NULL)
                printf("\n Stack UNDERFLOW");
        else
        {
                top=top->next;
                printf("\n The value deleted is : %d",ptr->data);
                free(ptr);
        }
        return top;
}
struct stack *display(struct stack *top)
{
        struct stack *ptr;
        ptr=top;
        if(top==NULL)
                printf("\n Stack is Empty ");
        else
        {
                while(ptr!=NULL)
                {
                        printf("\t %d",ptr->data);
                        ptr=ptr->next;
                }
        }
        return top;
}
```

Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

Polish Notation:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a

mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.
Example: (A + B) * (C - D)
Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).
Example: * + A B – C D
Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.
Example: A B + C D - *
The three important features of postfix expression are:
1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.
We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Converting expressions using Stack:
Let us convert the expressions from one type to another. These can be done as follows:
1. Infix to postfix
2. Infix to prefix

Conversion from infix to postfix:
Procedure to convert from infix expression to postfix expression is as follows:
1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
b) If the scanned symbol is an operand, then place directly in the postfix expression (output).
C)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.
d) 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:
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.	

Example 2:.

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.	

Program to convert an infix to postfix expression:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
#define MAX 100
char stack[MAX];
int top=-1;
void push(char stack[],char);
char pop(char stack[]);
voidInfixtoPostfix(char source[],char target[]);
intgetPriority(char);
void main()
{
        char infix[100],postfix[100];
        clrscr();
        printf("\n Enter any infix expression : ");
        gets(infix);
        strcpy(postfix,"");
        InfixtoPostfix(infix,postfix);
        printf("\n The corresponding postfix expression is : ");
        puts(postfix);
        getch();
}
void InfixtoPostfix(char source[],char target[])
{
        int i=0,j=0;
        strcpy(target,"");
        while(source[i]!='\0')
        {
                if(source[i]=='(')
                {
                        push(stack,source[i]);
                        i++;
                }
                else if(source[i]==')')
                {
                        while((top!=-1) && (stack[top]!='('))
```

```
                {
                        target[j]=pop(stack);
                        j++;
                }
                if(top==-1)
                {
                        printf("\n Incorrect Expression ");
                        return;
                }
                pop(stack); // Remove left parenthesis
                i++;
        }
        else if(isdigit(source[i]) || isalpha(source[i]))
        {
                target[j]=source[i];
                j++;
                i++;
        }
        else if(source[i]=='+' || source[i]=='-'|| source[i]=='*'|| source[i]=='/'|| source[i]=='%')
        {
        while((top!=-1) && (stack[top]!='(') && (getPriority(stack[top]) >= getPriority(source[i])))
           {
                target[j]=pop(stack);
                j++;
           }
        push(stack,source[i]);
        i++;
        }
        else
        {
                printf("\n Incorrect element in expression ");
                exit(1);
        }
    }
    while((top!=-1) && (stack[top]!='('))
    {
        target[j]=pop(stack);
        j++;
    }
    target[j]='\0';
}
intgetPriority(char op)
```

```
{
        if(op=='/' || op=='*' ||op=='%')
                return 1;
        else if(op=='+' || op=='-')
                return 0;
        else
                return -1;
}
void push(char stack[],char value)
{
        if(top==MAX-1)
                printf("\n stack overflow");
        else
        {
                top=top+1;
                stack[top]=value;
        }
}
char pop(char stack[])
{
        char value=' ';
        if(top==-1)
                printf("\n Stack Underflow");
        else
        {
                value=stack[top];
                top=top-1;
        }
        return value;
}
```

Conversion from infix to prefix:

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression A + B - C into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

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

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	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.	

Program to convert an infix to prefix expression:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
#define MAX 100
char stack[MAX];
int top=-1;
void push(char stack[],char);
char pop(char stack[]);
void InfixtoPostfix(char source[],char target[]);
intgetPriority(char);
void main()
{
        char infix[100],postfix[100];
        clrscr();
        printf("\n Enter any infix expression : ");
        gets(infix);
        strrev(infix);
        strcpy(postfix,"");
        InfixtoPostfix(infix,postfix);
        strrev(postfix);
        printf("\n The corresponding prefix expression is : ");
        puts(postfix);
        getch();
}
void InfixtoPostfix(char source[],char target[])
{
        int i=0,j=0;
        strcpy(target,"");
        while(source[i]!='\0')
        {
                if(source[i]==')')
                {
                        push(stack,source[i]);
                        i++;
                }
                else if(source[i]=='(')
                {
                        while((top!=-1) && (stack[top]!=')'))
```

```
                {
                        target[j]=pop(stack);
                        j++;
                }
                if(top==-1)
                {
                        printf("\n Incorrect Expression ");
                        return;
                }
                pop(stack); // Remove left parenthesis
                i++;
        }
        else if(isdigit(source[i]) || isalpha(source[i]))
        {
                target[j]=source[i];
                j++;
                i++;
        }
        else if(source[i]=='+' || source[i]=='-'|| source[i]=='*'|| source[i]=='/'|| source[i]=='%')
        {
           while((top!=-1) && (stack[top]!='(') && (getPriority(stack[top])
>getPriority(source[i])))
           {
                target[j]=pop(stack);
                j++;
           }
           push(stack,source[i]);
           i++;
        }
        else
        {
                printf("\n Incorrect element in expression ");
                exit(1);
        }
    }
    while((top!=-1) && (stack[top]!='('))
    {
        target[j]=pop(stack);
        j++;
    }
    target[j]='\0';
}
```

```
intgetPriority(char op)
{
        if(op=='/' || op=='*' ||op=='%')
                return 1;
        else if(op=='+' || op=='-')
                return 0;
        else
                return -1;
}
void push(char stack[],char value)
{
        if(top==MAX-1)
                printf("\n stack overflow");
        else
        {
                top=top+1;
                stack[top]=value;
        }
}
char pop(char stack[])
{
        char value=' ';
        if(top==-1)
                printf("\n Stack Underflow");
        else
        {
                value=stack[top];
                top=top-1;
        }
        return value;
}
```

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; this is our obvious advantage.

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

Program to evaluate a postfix expression:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 100
float stack[MAX];
int top=-1;
void push(float stack[],float value);
float pop(float stack[]);
floatevaluatePostfixExp(char exp[]);
void main()
{
        float value;
        charexp[100];
        clrscr();
        printf("\n Enter any postfix expression : ");
        gets(exp);
        value=evaluatePostfixExp(exp);
        printf("\n Value of the postfix expression = %.2f",value);
        getch();
}
```

```
floatevaluatePostfixExp(char exp[])
{
	int i=0;
	float op1,op2,value;
	while(exp[i]!='\0')
	{
		if(isdigit(exp[i]))
			push(stack,(float)(exp[i]-'0'));
		else
		{
			op2=pop(stack);
			op1=pop(stack);
			switch(exp[i])
			{
				case '+' :
					value=op1+op2;
					break;
				case '-' :
					value=op1-op2;
					break;
				case '*' :
					value=op1*op2;
					break;
				case '/' :
					value=op1/op2;
					break;
				case '%' :
					value=(int)op1%(int)op2;
					break;
			}
			push(stack,value);
		}
		i++;
	}
	return (pop(stack));
}
void push(float stack[],float value)
{
	if(top==MAX-1)
		printf("\n Stack overflow");
	else
	{
```

```
                top=top+1;
                stack[top]=value;
        }
}
float pop(float stack[])
{
        float value=-1;
        if(top==-1)
                printf("\n Stack underflow");
        else
        {
                value=stack[top];
                top=top-1;
        }
        return value;
}
```

Queue:

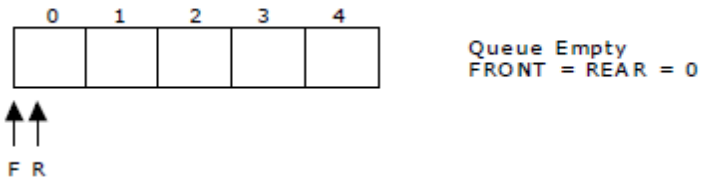
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

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. We shall use the following operations on queues:

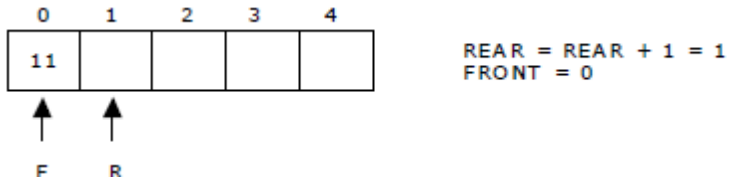
· *enqueue*: which inserts an element at the end of the queue.
· *dequeue*: which deletes an element at the start of the queue.

Representation of Queue:

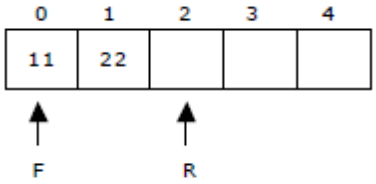
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



Now, insert 11 to the queue. Then queue status will be:

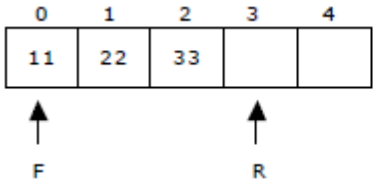


Next, insert 22 to the queue. Then the queue status is:



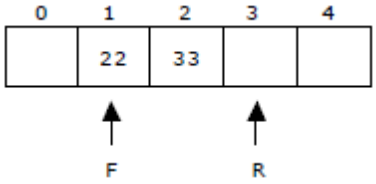
REAR = REAR + 1 = 2
FRONT = 0

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



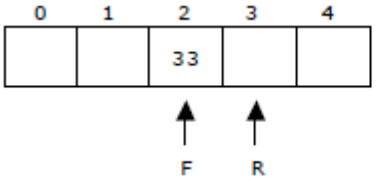
REAR = REAR + 1 = 3
FRONT = 0

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



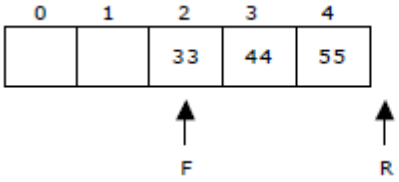
REAR = 3
FRONT = FRONT + 1 = 1

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:



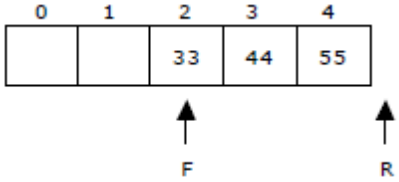
REAR = 3
FRONT = FRONT + 1 = 2

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



REAR = 5
FRONT = 2

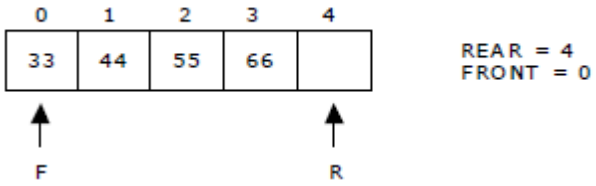
Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REAR = 5
FRONT = 2

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come 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.**

Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insert_element (): inserts an element at the end of queue Q.
2. delete_element (): deletes the first element of Q.
3. display(): displays the elements in the queue.

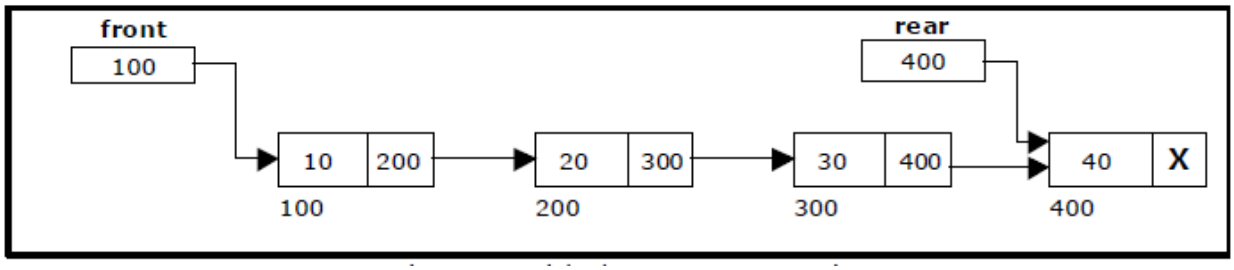
```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int queue[MAX];
int front=-1,rear=-1;
void insert_element();
int delete_element();
void display();
void main()
{
        intoption,value;
        clrscr();
        do
        {
                printf("\n ****** MAIN MENU *******");
                printf("\n\t 1. Insert an Element  ");
                printf("\n\t 2. Delete an Element  ");
                printf("\n\t 3. Display the queue");
                printf("\n\t 4. Exit ");
                printf("\n \n Enter your option :");
                scanf("%d",&option);
                switch(option)
                {
```

```
                case 1:
                        insert_element();
                        break;
                case 2:
                        value=delete_element();
                        if(value!=-1)
                                printf("\n %d data element deleted from the queue ",value);
                        break;
                case 3:
                        display();
                        break;
                }
        }while(option!=4);
        getch();
}
voidinsert_element()
{
        intnum;
        printf("\n Enter the number to be inserted in the queue : ");
        scanf("%d",&num);
        if(rear==MAX-1)
                printf("\n OVERFLOW ");
        else if(front==-1 && rear==-1)
                front=rear=0;
        else
                rear++;
        queue[rear]=num;
}
intdelete_element()
{
        int value;
        if(front==-1 || front>rear)
        {
                printf("\n UNDERFLOW");
                return -1;
        }
        else
        {
                value=queue[front];
                front++;
                if(front>rear)
                        front=rear=-1;
```

```
                return value;
        }
}
void display()
{
        int i;
        if(front==-1 || front>rear)
                printf("\n Queue is Empty");
        else
        {
                for(i=0;i<=rear;i++)
                        printf("\t %d",queue[i]);
        }
}
```

Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.
The linked queue looks as shown below.




```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct node
{
        int data;
        struct node *next;
};
struct queue
{
        struct node *front;
        struct node *rear;
};
struct queue *q;
```

```
voidcreate_queue(struct queue*);
struct queue *insert_element(struct queue*,int);
struct queue *delete_element(struct queue*);
struct queue *display(struct queue*);
void main()
{
	intval,option;
	create_queue(q);
	clrscr();
	do
	{
		printf("\n ****** MAIN MENU *******");
		printf("\n\t 1. Insert an Element  ");
		printf("\n\t 2. Delete an Element  ");
		printf("\n\t 3. Display the queue");
		printf("\n\t 4. Exit ");
		printf("\n \n Enter your option :");
		scanf("%d",&option);
		switch(option)
		{
		case 1:
			printf("\n Enter the number to insert in the queue : ");
			scanf("%d",&val);
			q=insert_element(q,val);
			break;
		case 2:
			q=delete_element(q);
			break;
		case 3:
			q=display(q);
			break;
		}
	}while(option!=4);
	getch();
}
voidcreate_queue(struct queue *q)
{
	q->rear=NULL;
	q->front=NULL;
}
struct queue *insert_element(struct queue *q,intval)
{
```

```
        struct node *ptr;
        ptr=(struct node*)malloc(sizeof(struct node));
        ptr->data=val;
        if(q->front==NULL)
        {
                q->front=ptr;
                q->rear=ptr;
                q->front->next=q->rear->next=NULL;
        }
        else
        {
                q->rear->next=ptr;
                q->rear=ptr;
                q->rear->next=NULL;
        }
        return q;
}
struct queue *display(struct queue *q)
{
        struct node *ptr;
        ptr=q->front;
        if(ptr==NULL)
                printf("\n Queue is  empty ");
        else
        {
                while(ptr!=q->rear)
                {
                        printf("\t %d",ptr->data);
                        ptr=ptr->next;
                }
                printf("\t %d",ptr->data);
        }
        return q;
}
struct queue *delete_element(struct queue *q)
{
        struct node *ptr;
        ptr=q->front;
        if(q->front==NULL)
                printf("\n Underflow");
        else
        {
```

```
            q->front=q->front->next;
            printf("\n The value being deleted is : %d ",ptr->data);
            free(ptr);
        }
        return q;
}
```

Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

Circular Queue:

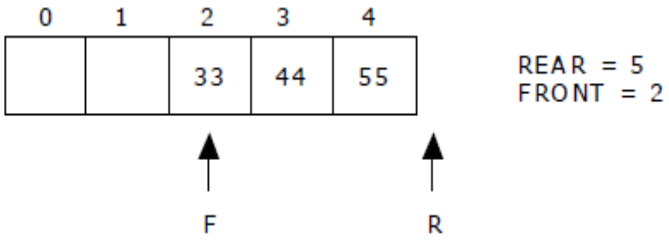
A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

There are two problems associated with linear queue. They are:

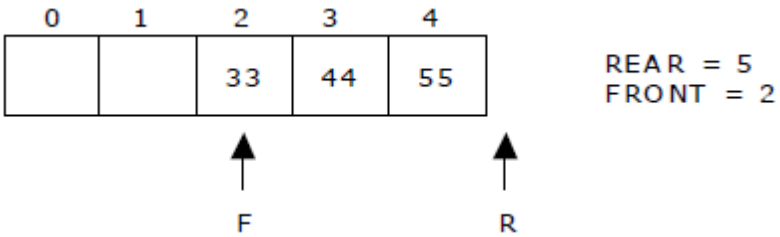
· Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.

· Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:

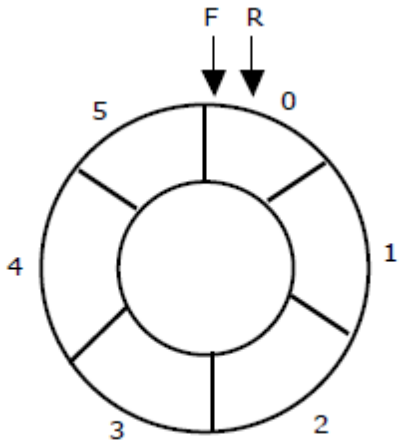


Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.** In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

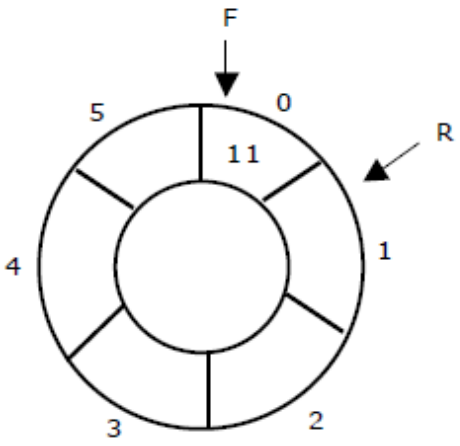
Representation of Circular Queue:
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

Now, insert 11 to the circular queue. Then circular queue status will be:

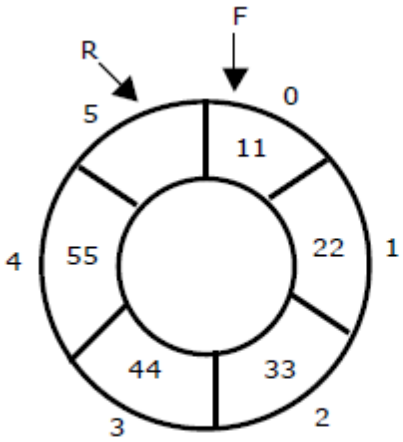


Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue

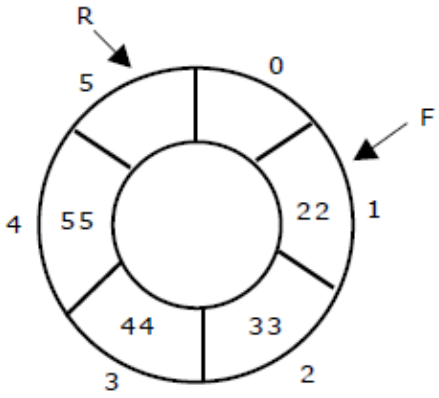
status is:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

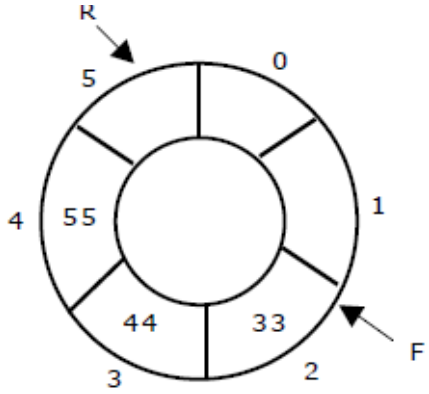
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4

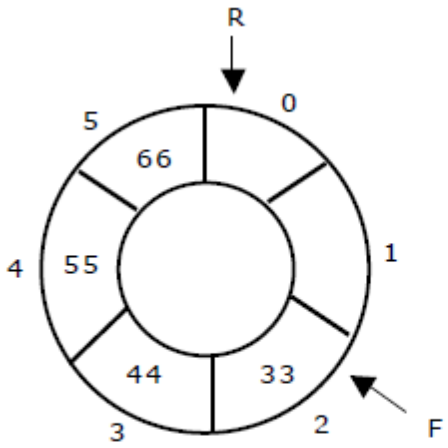
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

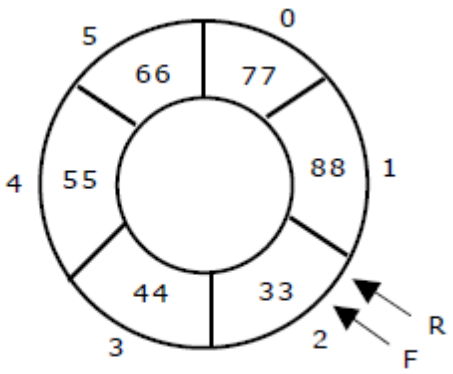
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

Source code for Circular Queue operations, using array:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int queue[MAX];
int front=-1,rear=-1;
void insert_element();
int delete_element();
void display();
void main()
{
        intoption,value;
        clrscr();
        do
        {
                printf("\n ****** MAIN MENU *******");
                printf("\n\t 1. Insert an Element  ");
                printf("\n\t 2. Delete an Element  ");
                printf("\n\t 3. Display the queue");
                printf("\n\t 4. Exit ");
                printf("\n \n Enter your option :");
                scanf("%d",&option);
                switch(option)
                {
                case 1:
                        insert_element();
```

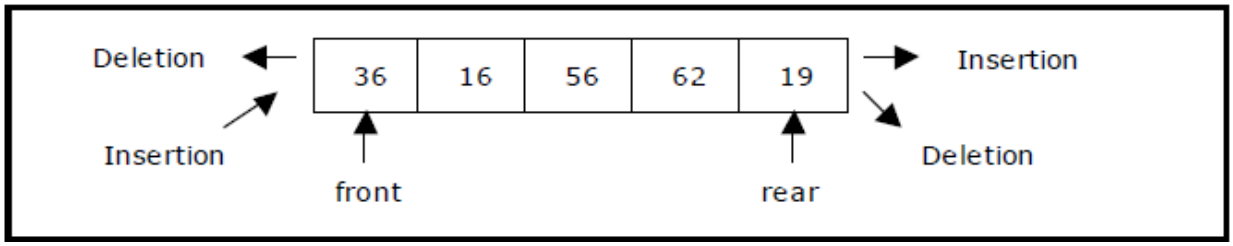
```
                    break;
            case 2:
                    value=delete_element();
                    if(value!=-1)
                            printf("\n %d data element deleted from the queue ",value);
                    break;
            case 3:
                    display();
                    break;
            }
    }while(option!=4);
    getch();
}
Void insert_element()
{
    intnum;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d",&num);
    if(front==-1 && rear ==MAX-1)
            printf("\n OVERFLOW ");
    else if(front==-1 && rear==-1)
    {
            front=rear=0;
            queue[rear]=num;
    }
    else if(rear==MAX-1 && front!=0)
    {
            rear=0;
            queue[rear]=num;
    }
    else
    {
            rear++;
            queue[rear]=num;
    }
}
Int delete_element()
{
    int value;
    if(front==-1 || rear==-1)
    {
            printf("\n UNDERFLOW");
```

```
                return -1;
        }
        value=queue[front];
        if(front==rear)
                front=rear=-1;
        else
        {
                if(front==MAX-1)
                        front=0;
                else
                        front++;
        }
                return value;
}
void display()
{
        int i;
        if(front==-1 || rear==-1)
                printf("\n Queue is Empty");
        else
        {
                if(front<rear)
                {
                        for(i=front;i<=rear;i++)
                                printf("\t %d",queue[i]);
                }
                else
                {
                                for(i=front;i<MAX;i++)
                                        printf("\t %d",queue[i]);
                                for(i=0;i<=rear;i++)
                                        printf("\t %d",queue[i]);

                }
        }
}
```

Deque:

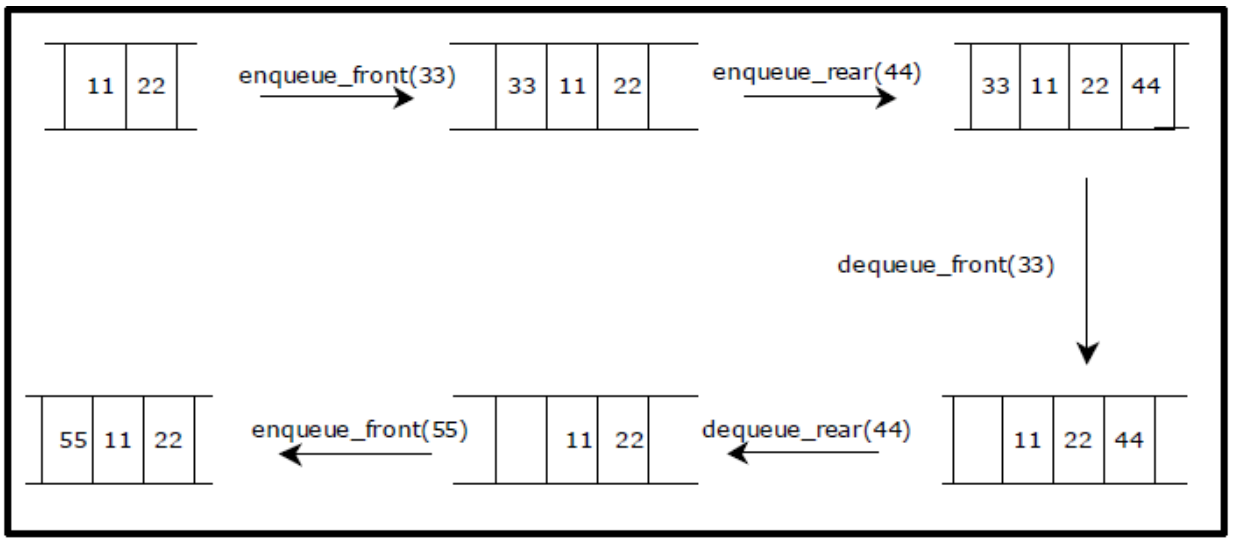
In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. The following figure shows the representation of a deque.



Representation of a deque

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

· enqueue_front: insert an element at front.
· dequeue_front: delete an element at front.
· enqueue_rear: insert element at rear.
· dequeue_rear: delete element at rear.



Basic operations on deque

There are two variations of deque. They are:

· Input restricted deque (IRD)
· Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.