

## CS3351 – DATA STRUCTURES

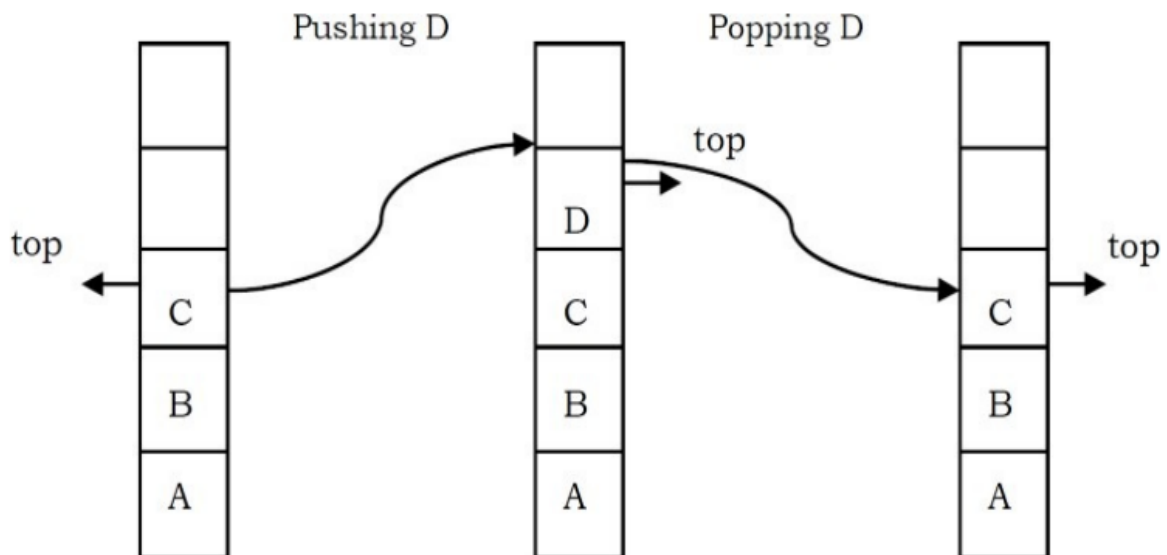
### UNIT II STACKS AND QUEUES

Stack ADT – Operations – Applications – Balancing Symbols – Evaluating arithmetic expressions  
-Infix to Postfix conversion – Function Calls – Queue ADT – Operations – Circular Queue –  
DeQueue – Applications of Queues.

### What is a Stack?

A stack is a simple data structure used for storing data (similar to Linked Lists). In a stack, the order in which the data arrives is important. A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and they are placed on the top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

**Definition:** A *stack* is an ordered list in which insertion and deletion are done at one end, called *top*. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.



**Stack ADT**

The following operations make a stack an ADT. For simplicity, assume the data is an integer type.

### Main stack operations

- Push (int data): Inserts *data* onto stack.
- int Pop(): Removes and returns the last inserted element from the stack.

### Auxiliary stack operations

- int Top(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in the stack.
- int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.

### Applications

Following are some of the applications in which stacks play an important role.

#### Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer to [Problems](#) section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML

#### Indirect applications

- Auxiliary data structure for other algorithms (Example: Tree traversal algorithms)
- Component of other data structures (Example: Simulating queues, refer [Queues](#) chapter)

### Implementation

There are many ways of implementing stack ADT; below are the commonly used methods.

- Array based implementation
- Linked lists implementation

**\*\*\*\*\*ARRAY BASED IMPLEMENTATION\*\*\*\*\***

//Implementation of Stack using Array

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

```
#include<stdlib.h>
```

```
#define SIZE 5
```

```
int stack[SIZE];
```

```
int top=-1,element;
```

```
void push();
```

```
void pop();
```

```
void peek();
```

```
void display();
```

```
int main()
```

```
{
```

```
    int choice;
```

```
    do
```

```
    {
```

```
        printf("enter choice 1.push 2.pop 3.peek 4.display\n");
```

```
        scanf("%d",&choice);
```

```
        switch(choice)
```

```
        {
```

```
        case 1:
```

```
            push();
```

```
            break;
```

```
        case 2:
```

```
            pop();
```

```
            break;
```

```
        case 3:
```

```
            peek();
```

```
            break;
```

```
        case 4:
```

```
            display();
```

```
            break;
```

```
        default:
```

```
            printf("no choice\n");
```

```
        }
```

```
    }
```

```
    while(choice!=0);
```

```
    return 0;
}
void push()
{
    if(top==SIZE-1)
    {
        printf("Stack is overflow\n");
    }
    else
    {
        printf("enter the element\n");
        scanf("%d",&element);
        top++;
        stack[top]=element;
    }
}
void pop()
{
    if(top== -1)
        printf("Stack is underflow\n");

    else
    {
        element=stack[top];
        top--;
        printf("%d\n",element);
    }
}
void peek()
{
    printf("The top element in the stack: %d\n",stack[top]);
}
void display()
{
    printf("The elements are\n");
    int i;
```

```
for(i=top; i>=0; i--)
{
    printf("%d\n",stack[i]);
}
}
```

\*\*\*\*\***LINKED LIST BASED IMPLEMENTATION**\*\*\*\*\*

//Implementation of Stack using LinkedList

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

```
#include<stdlib.h>
```

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node *next;
```

```
}*top,*newnode,*temp;
```

```
void push();
```

```
void pop();
```

```
void peek();
```

```
void display();
```

```
int main()
```

```
{
```

```
    int choice;
```

```
    do
```

```
    {
```

```
        printf("enter choice 1.push 2.pop 3.peek 4.display\n");
```

```
        scanf("%d",&choice);
```

```
        switch(choice)
```

```
        {
```

```
        case 1:
```

```
            push();
```

```
            break;
```

```
        case 2:
```

```
            pop();
```

```
            break;
```

```
        case 3:
```

```
            peek();
```

```
            break;
```

```
case 4:
    display();
    break;
default:
    printf("no choice\n");
}

}
while(choice!=0);

return 0;
}
void push()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("enter the data\n");
    scanf("%d",&newnode->data);
    newnode->next=top;
    top=newnode;
}
void pop()
{
    temp=top;
    if(top==NULL)
    {
        printf("Stack is empty");
    }
    else
    {
        printf("\n%d is deleted",top->data);
        top=top->next;
        free(temp);
    }
}
void peek()
{
    if(top==NULL)
        printf("Stack is empty");
```

```
else
    printf("\nThe top element is %d", top->data);
}
void display()
{
    temp=top;
    if(top==NULL)
    {
        printf("\nStack is empty");
    }
    else
    {
        while(temp!=NULL)
        {
            printf("\nElements in stack %d ",temp->data);
            temp=temp->next;
        }
    }
}
```

## Balancing of symbols

Stacks can be used to check whether the given expression has balanced symbols. This algorithm is especially useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [- then it is written to the stack. When a closing delimiter is encountered like ), }, or ]-the stack is popped.

The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line. A linear-time  $O(n)$  algorithm based on stack can be given as:

### Algorithm:

- a) Create a stack.
- b) while (end of input is not reached) {
  - 1) If the character read is not a symbol to be balanced, ignore it.
  - 2) If the character is an opening symbol like (, [, {, push it onto the stack
  - 3) If it is a closing symbol like ), ], }, then if the stack is empty report an error. Otherwise pop the stack.
  - 4) If the symbol popped is not the corresponding opening symbol, report an error.
- }
- c) At end of input, if the stack is not empty report an error

### Examples:

Example	Valid?	Description
(A+B)+(C-D)	Yes	The expression has a balanced symbol
((A+B)+(C-D)	No	One closing brace is missing
((A+B)+[C-D])	Yes	Opening and immediate closing braces correspond
((A+B)+[C-D])}	No	The last closing brace does not correspond with the first opening parenthesis

For tracing the algorithm let us assume that the input is: () ( () ()

Input Symbol, A[i]	Operation	Stack	Output
(	Push (	(	
)	Pop ( Test if ( and A[i] match? YES		
(	Push (	(	
(	Push (	((	
)	Pop ( Test if ( and A[i] match? YES	(	



[	Push [	((	
(	Push (	((	
)	Pop ( Test if( and A[i] match? YES	((	
]	Pop [ Test if [ and A[i] match? YES	(	
)	Pop ( Test if( and A[i] match? YES		
	Test if stack is Empty? YES		TRUE

### \*\*\*IMPLEMENTATION OF BALANCING OF SYMBOLS USING STACK\*\*

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 30
int top=-1;
int stack[MAX];
void push(char);
char pop();
int match(char a,char b);
int check(char []);
int main()
{
    char exp[MAX];
    int valid;
    printf("Enter an algebraic expression : ");
    gets(exp);
    valid=check(exp);
    if(valid==1)
        printf("Valid expression\n");
    else
```

```
printf("Invalid expression\n");

return 0;

}
int check(char exp[] )
{
    int i;
    char temp;
    for(i=0;i<strlen(exp);i++)
    {
        if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
            push(exp[i]);
        if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
            if(top== -1) /*stack empty*/
            {
                printf("Right parentheses are more than left parentheses\n");
                return 0;
            }
        else
        {
            temp=pop();
            if(!match(temp, exp[i]))
            {
                printf("Mismatched parentheses are : ");
                printf("%c and %c\n",temp,exp[i]);
                return 0;
            }
        }
    }
    if(top== -1)
    {
        printf("Balanced Parentheses\n");
        return 1;
    }
    else
    {
        printf("Left parentheses more than right parentheses\n");
```

```
        return 0;
    }
}
int match(char a,char b)
{
    if(a=='[' && b==']')
        return 1;
    if(a=='{' && b=='}')
        return 1;
    if(a=='(' && b==')')
        return 1;
    return 0;
}
void push(char item)
{
    if(top==(MAX-1))
    {
        printf("Stack Overflow\n");
        return;
    }
    top=top+1;
    stack[top]=item;
}
char pop()
{
    if(top==-1)
    {
        printf("Stack Underflow\n");
        exit(1);
    }
    return(stack[top--]);
}
```

## Evaluating the Arithmetic Expression

There are 3 types of Expressions

- Infix Expression
- Postfix Expression
- Prefix Expression

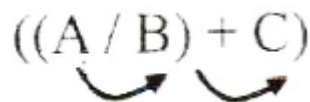
### INFIX:

The arithmetic operator appears between the two operands to which it is being applied.

$$A / B + C$$

### POSTFIX:

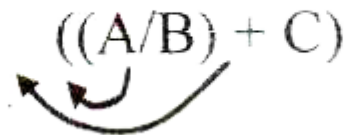
The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation.

$$((A / B) + C)$$


$$AB/C +$$

### PREFIX:

The arithmetic operator is placed before the two operands to which it applies. Also called polish notation.

$$((A/B) + C)$$


$$+/ABC$$

## Evaluating Arithmetic Expressions

1. Convert the given infix expression to Postfix expression
2. Evaluate the postfix expression using stack.

### Algorithm to convert Infix Expression to Postfix Expression:

Read the infix expression one character at a time until it encounters the delimiter “#”

Step 1: If the character is an operand, place it on the output.

Step 2: If the character is an operator, push it onto the stack. If the stack operator has a higher or equal priority than input operator then pop that operator from the stack and place it onto the output.

Step 3: If the character is left parenthesis, push it onto the stack

Step 4: If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

For better understanding let us trace out an example:  $A * B - (C + D) + E$

Input Character	Operation on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and append to postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End of input	Pop till empty		AB*CD+-E+

**\*\*\*\*\*IMPLEMENTATION OF INFIX TO POSTFIX CONVERSION\*\*\*\*\***

```
#include<stdio.h>
#include<ctype.h>
char stack[100];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}
char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}
int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}
int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
    scanf("%s",exp);
    printf("\n");
    e = exp;
    while(*e != '\0')
    {
        if(isalnum(*e))
```

```
    printf("%c ",*e);
else if(*e == '(')
    push(*e);
else if(*e == ')')
{
    while((x = pop()) != '(')
        printf("%c ", x);
}
else
{
    while(priority(stack[top]) >= priority(*e))
        printf("%c ",pop());
    push(*e);
}
e++;
}
while(top != -1)
{
    printf("%c ",pop());
}return 0;
}
```

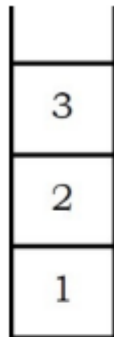
### **Evaluating the Postfix Expression**

#### **Algorithm to evaluate the obtained Postfix Expression**

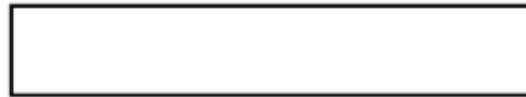
- 1 Scan the Postfix string from left to right.
- 2 Initialize an empty stack.
- 3 Repeat steps 4 and 5 till all the characters are scanned.
- 4 If the scanned character is an operand, push it onto the stack.
- 5 If the scanned character is an operator, and if the operator is a unary operator, then pop an element from the stack. If the operator is a binary operator, then pop two elements from the stack. After popping the elements, apply the operator to those popped elements. Let the result of this operation be retVal onto the stack.
- 6 After all characters are scanned, we will have only one element in the stack.
- 7 Return top of the stack as result.

**Example:** Let us see how the above-mentioned algorithm works using an example. Assume that the postfix string is 123\*+5-.

Initially the stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. They will be pushed into the stack in that order.



Stack



Expression

The next character scanned is "\*", which is an operator. Thus, we pop the top two elements from the stack and perform the "\*" operation with the two operands. The second operand will be the first element that is popped.

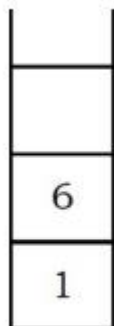


Stack



Expression

The value of the expression (2\*3) that has been evaluated (6) is pushed into the stack.



Stack



Expression

The next character scanned is "+", which is an operator. Thus, we pop the top two elements from



the stack and perform the “+” operation with the two operands. The second operand will be the first element that is popped.



Stack

$$1 + 6 = 7$$

Expression

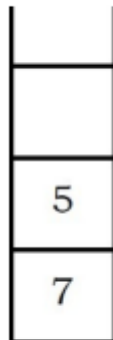
The value of the expression (1+6) that has been evaluated (7) is pushed into the stack.



Stack

Expression

The next character scanned is “5”, which is added to the stack.



Stack

Expression

The next character scanned is “-”, which is an operator. Thus, we pop the top two elements from the stack and perform the “-” operation with the two operands. The second operand will be the first element that is popped.



Stack

$$7 - 5 = 2$$

Expression

The value of the expression (7-5) that has been evaluated (2) is pushed into the stack.



Stack

Expression

Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned. End result:

- Postfix String : 123\*+5-
- Result : 2

#### \*\*\*\*\*IMPLEMENTATION OF EVALUATING POSTFIX EXPRESSION\*\*\*\*\*

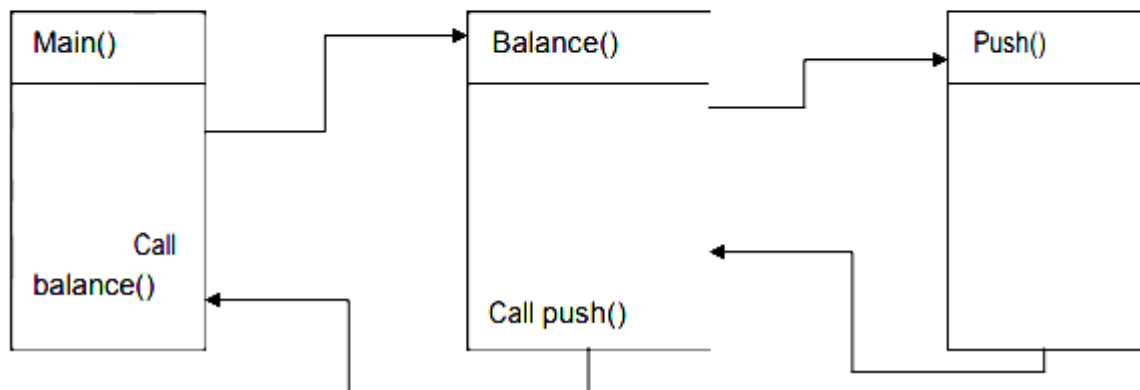
```
#include<stdio.h>
int stack[20];
int top = -1;
void push(int x)
{
    stack[++top] = x;
```

```
}
int pop()
{
    return stack[top--];
}
int main()
{
    char exp[20];
    char *e;
    int n1,n2,n3,num;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isdigit(*e))
        {
            num = *e - 48;
            push(num);
        }
        else
        {
            n1 = pop();
            n2 = pop();
            switch(*e)
            {
                case '+':
                {
                    n3 = n1 + n2;
                    break;
                }
                case '-':
                {
                    n3 = n2 - n1;
                    break;
                }
                case '*':
                {
```

```
        n3 = n1 * n2;
        break;
    }
    case '/':
    {
        n3 = n2 / n1;
        break;
    }
    case '^':
    {
        n3 = n2 ^ n1;
        break;
    }
    }
    push(n3);
}
e++;
}
printf("\nThe result of expression %s = %d\n\n",exp,pop());
return 0;
}
```

### Function Calls

When a call is made to a new function all the variables local to the calling routine need to be saved, otherwise the new function will overwrite the calling routine variables. Similarly the current location address in the routine must be saved so that the new function knows where to go after it is completed.



### Recursive Function to Find Factorial

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

### What is a Queue?

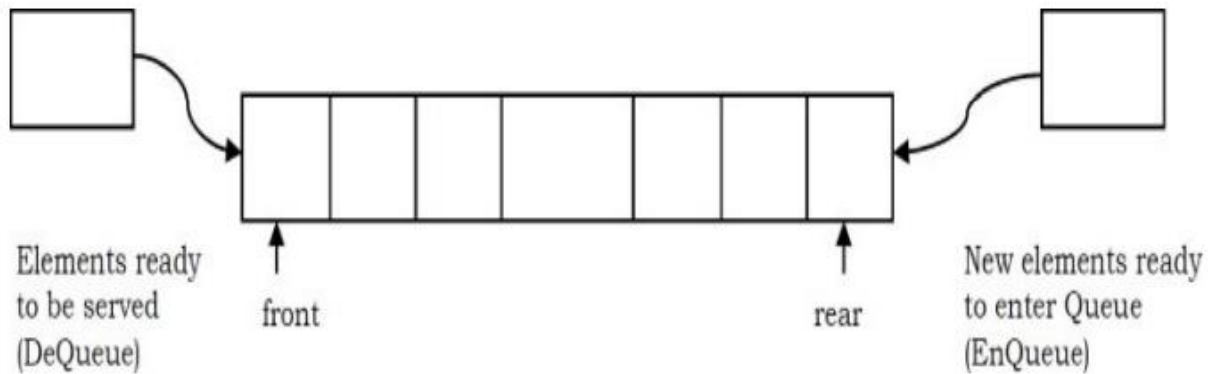
A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

**Definition:** A *queue* is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.

Similar to *Stacks*, special names are given to the two changes that can be made to a queue. When an element is inserted in a queue, the concept is called *EnQueue*, and when an element is removed from the queue, the concept is called *DeQueue*.

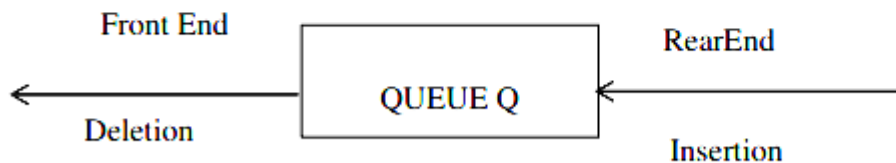
*DeQueueing* an empty queue is called *underflow* and *EnQueueing* an element in a full queue is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of

the queue.



### QUEUES:

- Queue is a Linear Data Structure that follows First in First out (FIFO) principle.
- Insertion of element is done at one end of the Queue called “**Rear**” end of the Queue.
- Deletion of element is done at other end of the Queue called “**Front**” end of the Queue.
- Example: - Waiting line in the ticket counter.



**Queue Model**

**Front Pointer:-**

It always points to the first element inserted in the Queue.

**Rear Pointer:-**

It always points to the last element inserted in the Queue.

**For Empty Queue:-**
$$\text{Front (F)} = - 1$$
$$\text{Rear (R)} = - 1$$
**Queue ADT**

The following operations make a queue an ADT. Insertions and deletions in the queue must follow the FIFO scheme. For simplicity we assume the elements are integers.

**Main Queue Operations**

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

**Auxiliary Queue Operations**

- int Front(): Returns the element at the front without removing it
- int QueueSize(): Returns the number of elements stored in the queue
- int IsEmptyQueueQ: Indicates whether no elements are stored in the queue or not

**(i) EnQueue operation:-**

- It is the process of inserting a new element at the rear end of the Queue.
- For every EnQueue operation
  - Check for Full Queue
  - If the Queue is full, Insertion is not possible.
  - Otherwise, increment the rear end by 1 and then insert the element in the rear end of the Queue.

**(ii) DeQueue Operation:-**

- It is the process of deleting the element from the front end of the queue.
- For every DeQueue operation
  - Check for Empty queue
  - If the Queue is Empty, Deletion is not possible.
  - Otherwise, delete the first element inserted into the queue and then increment the front by 1.

**Exceptional Conditions of Queue**

- Queue Overflow
- Queue Underflow

**(i) Queue Overflow:**

- An Attempt to insert an element X at the Rear end of the Queue when the Queue is full is said to be Queue overflow.
- For every Enqueue operation, we need to check this condition.

**(ii) Queue Underflow:**

- An Attempt to delete an element from the Front end of the Queue when the Queue is empty is said to be Queue underflow.
- For every DeQueue operation, we need to check this condition.

**Applications**

Following are some of the applications that use queues.

**Direct Applications**

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.



- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

### Indirect Applications

- Auxiliary data structure for algorithms
- Component of other data structures

### Implementation

There are many ways (similar to Stacks) of implementing queue operations and some of the commonly used methods are listed below.

- Array based implementation
- Linked lists based implementation

### \*\*\*\*\*ARRAY IMPLEMENTATION OF QUEUE \*\*\*\*\*

```
#include<stdio.h>
#include<stdlib.h>
#define max 5
int rear=-1,front=-1;
int queue[max];
void insert();
void delete();
void display();
void peek();
int main()
{
    int ch,num;
    while(1)
    {
        printf("\n1. Insert");
        printf("\n2. Delete");
        printf("\n3. Display");
        printf("\n4. Peek");
        printf("\n. EXIT");
        printf("\nEnter What you want :");
        scanf("%d",&ch);
```

```
    if(ch==1)
        insert();
    else if(ch==2)
        delete();
    else if(ch==3)
        display();
    else if(ch==4)
        peek();
    else if(ch==5)
        exit(1);
    else
        printf("\nInvalid Choice!!");
}
return 0;
}
void insert()
{
    int num;
    if(rear==max-1)
    {
        printf("\nQueue is Full !\n");
    }
    else if(front == -1 && rear == -1)
    {
        front=rear=NULL;
        printf("\nEnter a number for insert :");
        scanf("%d",&num);
        queue[rear]=num;
    }
    else
    {
        rear++;
        printf("\nEnter new number insert :");
        scanf("%d",&num);
        queue[rear]=num;
    }
}
void delete()
```

```
{
    int num;
    if(front== -1 && rear== -1)
    {
        printf("\nQueue is Empty !\n");
    }
    else if(front==rear)
    {
        front=rear=-1;
    }
    else
    {
        printf("The deleted element from queue is %d",queue[front]);
        front++;
    }
}

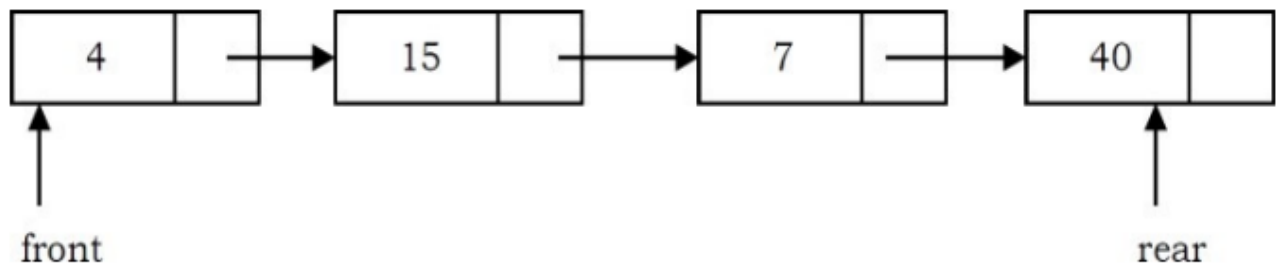
void display()
{
    int i;
    if(front== -1 && rear == -1)
    {
        printf("\nQueue Is Empty ! Nothing To Display !!");
    }
    else
    {
        for(i=front; i<rear+1; i++)
        {
            printf("%d\t",queue[i]);
        }
    }
}

void peek()
{
    if(front== -1 && rear == -1)
    {
        printf("\nQueue Is Empty ! Nothing To Display !!");
    }
    else
```

```
{
    printf("%d",queue[front]);
}
```

### Linked List Implementation

Another way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting an element at the end of the list. *DeQueue* operation is implemented by deleting an element from the beginning of the list.

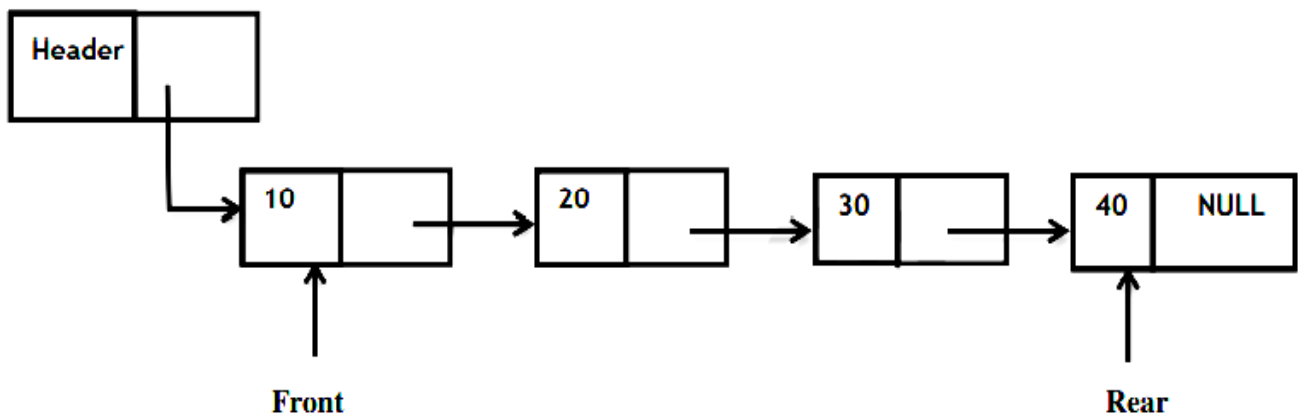


- Queue is implemented using SLL (Singly Linked List ) node.
- Enqueue operation is performed at the end of the Linked list and DeQueue operation is performed at the front of the Linked list.
- With Linked List implementation, for Empty queue

**Front = NULL & Rear = NULL**

Linked List representation of Queue with 4 elements

Q



### Declaration for Linked List Implementation of Queue ADT

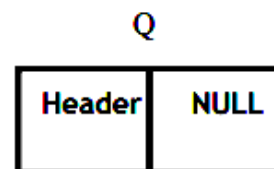
```
struct node;
typedef struct node * Queue;
typedef struct node * position;
int IsEmpty (Queue Q);
Queue CreateQueue (void);
void MakeEmpty (Queue Q);
void Enqueue (int X, Queue Q);
void Dequeue (Queue Q);
struct node
{
    int data ;
    position next;
} * Front = NULL, *Rear = NULL;
```

#### (i) Queue Empty Operation:

- Initially Queue is Empty.
- With Linked List implementation, Empty Queue is represented as S -> next = NULL.
- It is necessary to check for Empty Queue before deleting the front element in the Queue.

### ROUTINE TO CHECK WHETHER THE QUEUE IS EMPTY

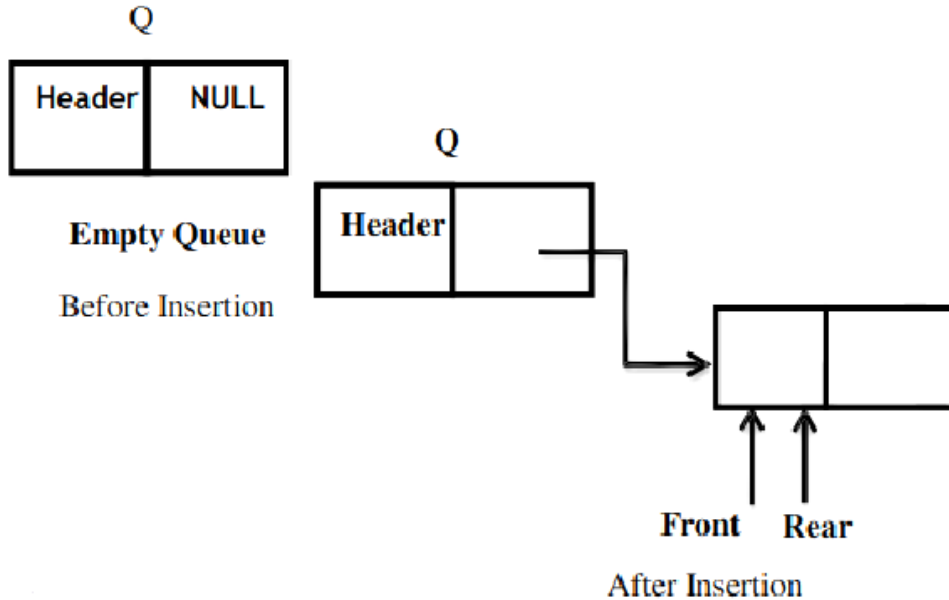
```
int IsEmpty (Queue Q)
{
    if ( Q->Next == NULL)
        return (1);
}
```



**Empty Queue**

#### (ii) EnQueue Operation

- It is the process of inserting a new element at the Rear end of the Queue.
- It takes two parameters, EnQueue ( int X , Queue Q ). The elements X to be inserted into the Queue Q.
- Using malloc ( ) function allocate memory for the newnode to be inserted into the Queue.
- If the Queue is Empty, the newnode to be inserted will become first and last node in the list. Hence Front and Rear points to the newnode.
- Otherwise insert the newnode in the Rear -> next and update the Rear pointer.



### (iii) DeQueue Operation

It is the process of deleting the front element from the Queue.

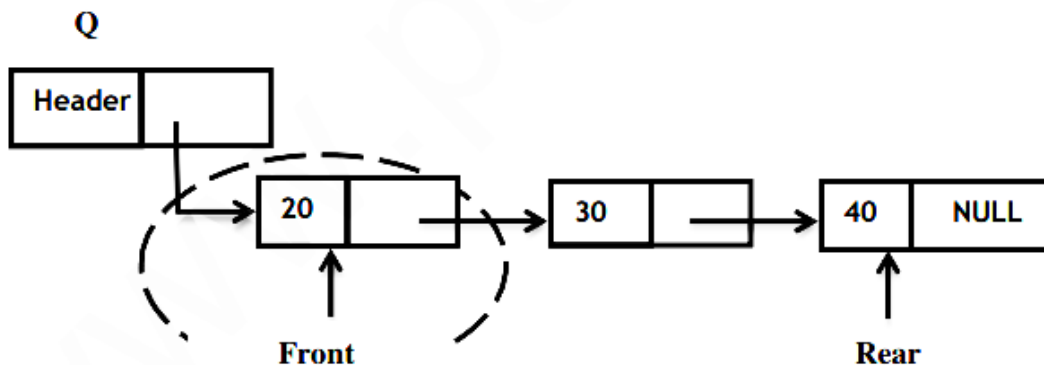
It takes one parameter, Dequeue ( Queue Q ). Always element in the front (i.e) element pointed by Q -> next is deleted always.

Element to be deleted is made "temp".

If the Queue is Empty, then deletion is not possible.

If the Queue has only one element, then the element is deleted and Front and Rear pointer is made NULL to represent Empty Queue.

Otherwise, Front element is deleted and the Front pointer is made to point to next node in the list. The free ( ) function informs the compiler that the address that temp is pointing to, is unchanged but the data present in that address is now undefined.



```
#include <stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
}*top,*newnode,*temp, *front=NULL, *rear=NULL;
void enqueue();
void dequeue();
void peek();
void display();
int main()
{
    int choice;
    do
    {
        printf("enter choice 1.enqueue 2.dequeue 3.peek 4.display\n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                enqueue();
                break;
            case 2:
                dequeue();
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            default:
                printf("no choice\n");
        }
    }
    while(choice!=0);
```

```
    return 0;
}
void enqueue()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("enter the data\n");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    if(front==NULL && rear==NULL)
    {
        front=rear=newnode;
    }
    else
    {
        rear->next=newnode;
        rear=newnode;
    }
}
void dequeue()
{
    temp=front;
    if(front==NULL && rear==NULL)
    {
        printf("\nQueue is empty");
    }
    else
    {
        printf("\n%d is deleted",front->data);
        front=front->next;
        free(temp);
    }
}
void peek()
{
    if(front==NULL && rear==NULL)
    {
        printf("\nQueue is empty");
    }
}
```



```
}  
else  
{  
    printf("\nThe front element is %d", front->data);  
}  
}  
void display()  
{  
    if(front==NULL && rear==NULL)  
    {  
        printf("\nQueue is empty");  
    }  
    else  
    {  
        temp=front;  
        while(temp!=NULL)  
        {  
            printf("\nElements in queue %d ",temp->data);  
            temp=temp->next;  
        }  
    }  
}
```

### **Applications of Queue**

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
4. Batch processing in operating system.
5. Job scheduling Algorithms like Round Robin Algorithm uses Queue.

### **Drawbacks of Queue (Linear Queue)**

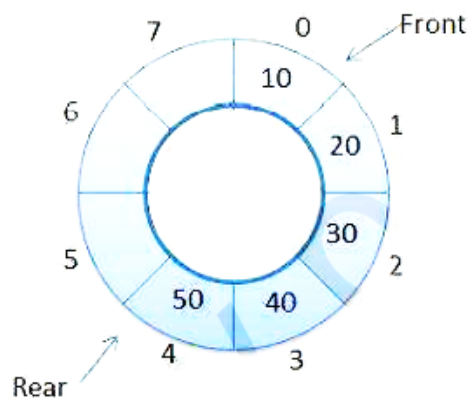
- With the array implementation of Queue, the element can be deleted logically only by moving  $Front = Front + 1$ .
- Here the Queue space is not utilized fully.

To overcome the drawback of this linear Queue, we use Circular Queue.

## CIRCULAR QUEUE

In Circular Queue, the insertion of a new element is performed at the very first location of the queue if the last location of the queue is full, in which the first element comes just after the last element.

- A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue.
- Circular queues have a fixed size.
- Circular queue follows FIFO principle.
- Queue items are added at the rear end and the items are deleted at front end of the circular queue
- Here the Queue space is utilized fully by inserting the element at the Front end if the rear end is full.



### Operations on Circular Queue

Fundamental operations performed on the Circular Queue are

- Circular Queue Enqueue
- Circular Queue Dequeue

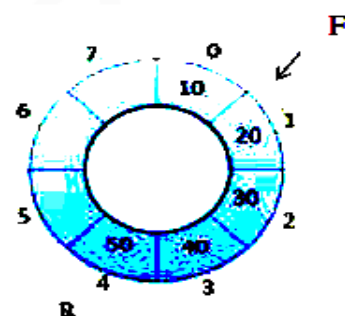
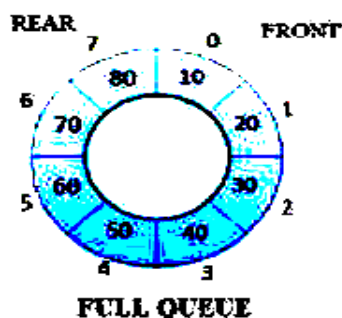
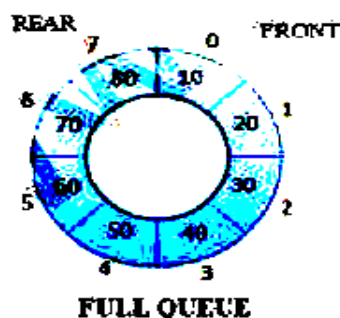
### Formula to be used in Circular Queue

For Enqueue       $\text{Rear} = (\text{Rear} + 1) \% \text{ArraySize}$

For Dequeue       $\text{Front} = (\text{Front} + 1) \% \text{ArraySize}$

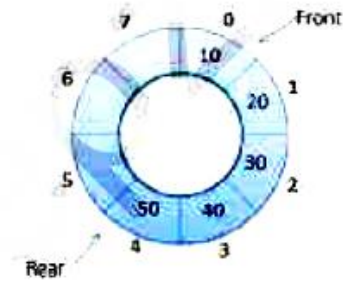
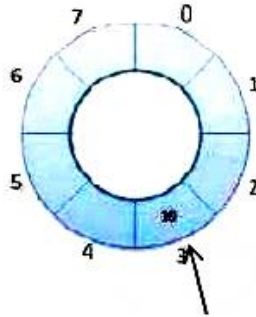
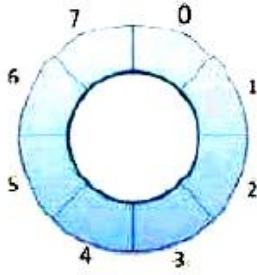
#### (i) Circular Queue Enqueue Operation

- ☐ It is same as Linear Queue EnQueue Operation (i.e) Inserting the element at the Rear end.
- ☐ First check for full Queue.
- ☐ If the circular queue is full, then insertion is not possible.
- ☐ Otherwise check for the rear end.
- ☐ If the Rear end is full, the elements start getting inserted from the Front end.



#### Circular Queue DeQueue Operation

- ☐ It is same as Linear Queue DeQueue operation (i.e) deleting the front element.
- ☐ First check for Empty Queue.
- ☐ If the Circular Queue is empty, then deletion is not possible.
- ☐ If the Circular Queue has only one element, then the element is deleted and Front and Rear pointer is initialized to -1 to represent Empty Queue.
- ☐ Otherwise, Front element is deleted and the Front pointer is made to point to next element in the Circular Queue.



F = -1, R = -1

F, R

### \*\*\*\*\*IMPLEMENTATION OF CIRCULAR QUEUE\*\*\*\*\*

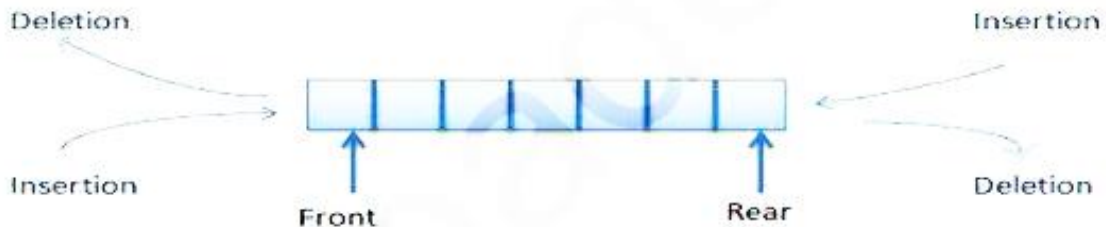
```
#include<stdio.h>
#define max 5
int queue[max];
int front=-1;
int rear=-1;
void enqueue(int element)
{
    if(front==-1 && rear==-1)
    {
        front=0;
        rear=0;
        queue[rear]=element;
    }
    else if((rear+1)%max==front)
    {
        printf("Queue is overflow..");
    }
    else
    {
        rear=(rear+1)%max;
        queue[rear]=element;
    }
}
void dequeue()
{
    if((front==-1) && (rear==-1))
    {
```

```
    printf("\nQueue is underflow..");
}
else if(front==rear)
{
    front=rear=-1;
}
else
{
    printf("\nThe dequeued element is %d", queue[front]);
    front=(front+1)%max;
}
}
void display()
{
    int i=front;
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
    else
    {
        printf("\nElements in a Queue are :");
        while(i!=rear)
        {
            printf("%d ", queue[i]);
            i=(i+1)%max;
        }
        printf("%d", queue[i]);
    }
}
int main()
{
    int choice=1,x;
    while(1)
    {
        printf("\n Press 1: Insert an element");
        printf("\nPress 2: Delete an element");
        printf("\nPress 3: Display the element");
```

```
printf("\nEnter your choice");  
scanf("%d", &choice);  
switch(choice)  
{  
case 1:  
  
    printf("Enter the element which is to be inserted");  
    scanf("%d", &x);  
    enqueue(x);  
    break;  
case 2:  
    dequeue();  
    break;  
case 3:  
    display();  
}  
}  
return 0;  
}
```

## DOUBLE-ENDED QUEUE (DEQUE)

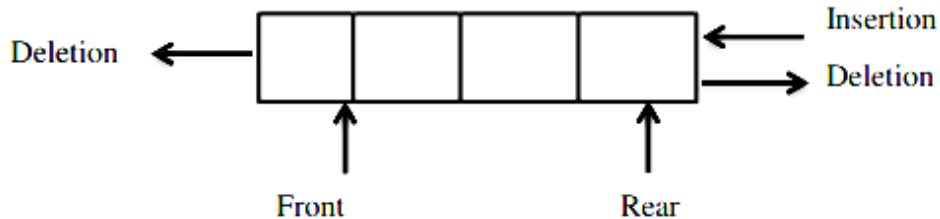
In DEQUE, insertion and deletion operations are performed at both ends of the Queue.



## Exceptional Condition of DEQUE

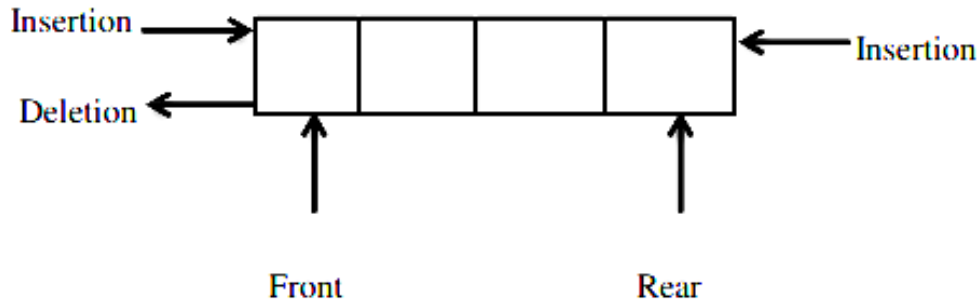
### (i) Input Restricted DEQUE

Here insertion is allowed at **one end** and deletion is allowed at **both ends**.



### (ii) Output Restricted DEQUE

Here insertion is allowed at **both ends** and deletion is allowed at **one end**.



## Operations on DEQUE

Four cases for inserting and deleting the elements in DEQUE are

1. Insertion At Rear End [ same as Linear Queue ]
2. Insertion At Front End
3. Deletion At Front End [ same as Linear Queue ]
4. Deletion At Rear End

\*\*\*\*\*IMPLEMENTATION OF DEQUE USING CIRCULAR ARRAY\*\*\*\*\*

```
#include<stdio.h>
#define N 5
int deque[N];
```

```
int front=-1;
int rear=-1;
void enqueue_front(int element)
{
    if((front==0 && rear==N-1) || (front==rear+1))
    {
        printf("Queue is overflow..");
    }
    else if(front==-1 && rear==-1)
    {
        front=rear=0;
        deque[front]=element;
    }
    else if(front==0)
    {
        front=N-1;
        deque[front]=element;
    }
    else
    {
        front--;
        deque[front]=element;
    }
}
void enqueue_rear(int element)
{
    if((front==0 && rear==N-1) || (front==rear+1))
    {
        printf("Queue is overflow..");
    }
    else if(front==-1 && rear==-1)
    {
        front=rear=0;
        deque[rear]=element;
    }
    else if(rear==N-1)
    {
        rear=0;
```



```
    deque[rear]=element;
}
else
{
    rear++;
    deque[rear]=element;
}
}
void display()
{
    int i=front;
    if(front== -1 && rear== -1)
    {
        printf("\n Queue is empty..");
    }
    else
    {
        printf("\nElements in a Queue are :");
        while(i!=rear)
        {
            printf("%d ", deque[i]);
            i=(i+1)%N;
        }
        printf("%d", deque[rear]);
    }
}
void getfront()
{
    if(front== -1 && rear== -1)
    {
        printf("\n Queue is empty..");
    }
    else
    {
        printf("The front element is %d", deque[front]);
    }
}
void getrear()
```

```
{
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
    else
    {
        printf("The rear element is %d",deque[rear]);
    }
}

void deque_front()
{
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
    else if(front==rear)
    {
        front=rear=-1;
        printf("The dequeued element is %d",deque[front]);
    }
    else if(front==N-1)
    {
        printf("The dequeued element is %d",deque[front]);
        front=0;
    }
    else
    {
        printf("The dequeued element is %d",deque[front]);
        front++;
    }
}

void deque_rear()
{
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
}
```

```
else if(front==rear)
{
    front=rear=-1;
    printf("The dequeued element is %d",deque[rear]);
}
else if(rear==0)
{
    printf("The dequeued element is %d",deque[rear]);
    rear=N-1;
}
else
{
    printf("The dequeued element is %d",deque[rear]);
    rear--;
}
}
int main()
{
    int choice,x;
    while(1)
    {
        printf("\nPress 1: Enqueue front element");
        printf("\nPress 2: Enqueue rear element");
        printf("\nPress 3: Dequeue front element");
        printf("\nPress 4: Dequeue rear element");
        printf("\nPress 5: Display the element");
        printf("\nPress 6: Get the front element");
        printf("\nPress 7: Get the rear element");
        printf("\nEnter your choice");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter the element which is to be inserted");
                scanf("%d",&x);
                enqueue_front(x);
                break;
            case 2:
```

```
printf("Enter the element which is to be inserted");
scanf("%d",&x);
enqueue_rear(x);
break;
case 3:
    deque_front();
    break;
case 4:
    deque_rear();
    break;
case 5:
    display();
    break;
case 6:
    getfront();
    break;
case 7:
    getrear();
    break;
case 8:
    printf("Invalid choice");
}
}
return 0;
}
```

## **Applications of Queues**

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queue are used to maintain the play list in media players inorder to add and remove the songs from the play-list.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queues are used in operating systems for handling interrupts.