(Refer video DSA Lab 7 instructions and DSA Lab 8 instructions)

QUEUE CONCEPTS

Objectives:

In this lab, student will be able to:

- Understand queue as a data structure
- Design programs using queue concepts

I. SOLVED EXERCISE:

1) Implement a queue of integers. Include functions insertq, deleteq and displayq.

Description:

Whenever we perform an insertion the rear pointer is incremented by 1 and front remains the same. Whenever a deletion is performed the front is incremented by one. But in real life problem the front pointer will be in the same position the object which is in the queue shifts to the front. But when we implement a queue in computer the front moves, this is because if we shift the elements to the front the time complexity increases. To implement a linear queue we consider two pointers or markers front and rear. Initial value of front and rear is assumed to be -1

Algorithm: Queue Insert

Step 1. Check for overflow

```
If R== N-1
        Then write ('OVERFLOW')
        Return
```

Step2. else Increment rear pointer

```
R= R +1
```

Step3: Insert element

```
Q[R ] = val
```

Step4: If F = -1

```
then F = 0
```

Return.

Algorithm: Queue Delete

Step1: Check for underflow

If F = = -1

Then Write ('UNDERFLOW')

Return

Step2: Delete an element

val = Q[F]

Step3: Queue empty?

if F > R

then F=R =-1

else F= F +1

Step4. Return an element

Return(val)

Step5: Stop

Program:

File name: queue_fun.h

```
#define MAX 20
typedef struct {
   int x[MAX];
   int front;
   int rear;
} queue;

void insertq(queue *, int);
void displayq(queue);
int deleteq(queue *);
```

```
void insertq(queue * q,int x)
 {
   if(q->rear==MAX)
   {
      printf("\nOverflow\n");
   }
   else
   {
      q->x[++q->rear]=x;
      if(q->front==-1)
      {
         q->front=0;
      }
   }
 }

 int deleteq(queue * q)
 {
   int x;
   if(q->front==-1)
   {
      printf("\nUnderflow!!!\n");
   }
   else if(q->front==q->rear)
   {
      x=q->x[q->front];
      q->front=q->rear=-1;
      return x;
   }
   else
   {
      return q->x[q->front++];
```

```
    }
 }

  void displayq(queue q)
 {
    int i;
    if(q.front==-1&&q.rear==-1)
    {
       printf("\nQueue is Empty!!!");
    }
    else
    {
       printf("\nQueue is:\n");
       for(i=q.front;i<=q.rear;i++)
       {
          printf("%d\n",q.x[i]);
       }
    }
}
```

File name: queue.c

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

int main()
{
   queue q;
   q.front=-1;
   q.rear=-1;
   int ch,x,flag=1;
   while(flag)
```

```
    {
      printf("\n\n1. Insert Queue\n2. Delete Queue\n3. Display Queue\n4. Exit\n\n");
      printf("Enter your choice: ");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
           printf("\nEnter the Element:");
           scanf("%d",&x);
           insertq(&q,x);
           break;
        case 2:
           x=deleteq(&q);
           printf("\nRemoved %d from the Queue\n",x);
           break;
        case 3:
           displayq(q);
           break;
        case 4:
           flag=0;
           break;
        default:
           printf("\nWrong choice!!! Try Again.\n");
      }
    }
  return 0;
}
```

QUEUE APPLICATIONS

Objectives:

In this lab, student will be able to:

- Identify the need for queue data structure in a given problem.
- Develop c programs applying queue concepts

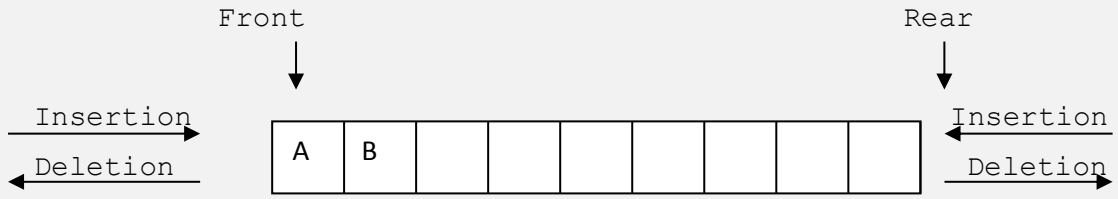
I. SOLVED EXERCISE:

1) Implement a dequeue of integers with following functions.

a) deleteLeft b) addLeft c) deleteRight d) addRight e) display

Description:

A queue that supports insertion and deletion at both the front and rear is called **double-ended queue or Dequeue.** A Dequeue is a linear list in which elements can be added or removed at either end but not in the middle.



The operations that can be performed on Dequeue are

1. Insert to the beginning
2. Insert at the end
3. Delete from the beginning
4. Delete from end

There are two variation of Dequeue namely, an input-restricted Dequeue and an Output restricted Dequeue – which are intermediate between a Dequeue and a queue. Specifically, an input-restricted Dequeue is a Dequeue which allows insertions at only one end of the list but allows deletions at both ends of the list; and an output-restricted Dequeue is a Dequeue which allows deletions at only one end of the list allows insertion at both ends of the list.

Algorithm: Insert End (rear)

Step1: if(front==MAX/2)

Write("Queue Full: Can't enter more elements at the end of queue");

return;

Step2: Else Insert the element and update the front pointer i.e queue[front]=token;

front=front+1;

Algorithm: Insert Start (front)

Step1: If (front==MAX/2)

Write("Queue Full: can't enter more elements at the start of queue");

return;

Step2: Insert the element by updating the rear pointer i.e

rear=rear-1;

queue[rear]=token;

Algorithm: Delete End (rear)

Step1: If(front==rear)

Write("Queue empty");

return 0;

Step2: Otherwise update the front and return the element i.e

front=front-1;

t=queue[front+1];

return t;

Algorithm: Delete Start (front)

Step1: If(front==rear)

Write("\nQueue empty");

return 0;

Step2: Update the rear pointer and return the element i.e

rear=rear+1;

t=queue[rear-1];

return t;

Program:

File name: deque_fun.h

```
#define MAX 30
typedef struct dequeue
{
   int data[MAX];
   int rear,front;
}dequeue;

void initialize(dequeue *p);
int empty(dequeue *p);
int full(dequeue *p);
void enqueueR(dequeue *p,int x);
void enqueueF(dequeue *p,int x);
int dequeueF(dequeue *p);
int dequeueR(dequeue *p);
void print(dequeue *p);

void initialize(dequeue *P)
{
   P->rear=-1;
   P->front=-1;
}

int empty(dequeue *P)
{
   if(P->rear==-1)
      return(1);
   return(0);
```

```
}

int full(dequeue *P)
{
   if((P->rear+1)%MAX==P->front)
      return(1);

   return(0);
}
void enqueueR(dequeue *P,int x)
{
   if(empty(P))
   {
      P->rear=0;
      P->front=0;
      P->data[0]=x;
   }
   else
   {
      P->rear=(P->rear+1)%MAX;
      P->data[P->rear]=x;
   }
}

void enqueueF(dequeue *P,int x)
{
   if(empty(P))
   {
      P->rear=0;
      P->front=0;
      P->data[0]=x;
   }
   else
```

```
    {
        P->front=(P->front-1+MAX)%MAX;
        P->data[P->front]=x;
    }
}

int dequeueF(dequeue *P)
{
    int x;
    x=P->data[P->front];
    if(P->rear==P->front)    /*delete the last element */
        initialize(P);
    else
        P->front=(P->front+1)%MAX;
    return(x);
}

int dequeueR(dequeue *P)
{
    int x;
    x=P->data[P->rear];
    if(P->rear==P->front)
        initialize(P);
    else
        P->rear=(P->rear-1+MAX)%MAX;
    return(x);
}

void print(dequeue *P)
{
    if(empty(P))
    {
        printf("\nQueue is empty!!");
```

```
        exit(0);
    }
    int i;
    i=P->front;
    while(i!=P->rear)
    {
        printf("\n%d",P->data[i]);
        i=(i+1)%MAX;
    }
    printf("\n%d\n",P->data[P->rear]);
}
```

File name: dequeuer.c

```
#include<stdio.h>
#include<process.h>
#include "dequeue_fun.h"
int main()
{
    int i,x,op,n;
    dequeue q;
    initialize(&q);
    do
    {

printf("\n1.Create\n2.Insert(rear)\n3.Insert(front)\n4.Delete(rear)\n5.Delete(front)");
        printf("\n6.Print\n7.Exit\n\nEnter your choice:");
        scanf("%d",&op);
        switch(op)
        {
            case 1: printf("\nEnter number of elements:");
                    scanf("%d",&n);
                    initialize(&q);
```

```
        printf("\nEnter the data:");

        for(i=0;i<n;i++)
        {
           scanf("%d",&x);
           if(full(&q))
           {
              printf("\nQueue is full!!");
              exit(0);
           }
           enqueueR(&q,x);
        }
        break;

case 2: printf("\nEnter element to be inserted:");
        scanf("%d",&x);

        if(full(&q))

        {
           printf("\nQueue is full!!");
           exit(0);
        }
        enqueueR(&q,x);
        break;

case 3: printf("\nEnter the element to be inserted:");
        scanf("%d",&x);
        if(full(&q))
        {
           printf("\nQueue is full!!");
           exit(0);
        }
```

```
                enqueueF(&q,x);
                break;

        case 4: if(empty(&q))
                {
                   printf("\nQueue is empty!!");
                   exit(0);
                }
                x=dequeueR(&q);
                printf("\nElement deleted is %d\n",x);
                break;

        case 5: if(empty(&q))
                {
                   printf("\nQueue is empty!!");
                   exit(0);
                }
                x=dequeueF(&q);
                printf("\nElement deleted is %d\n",x);
                break;

        case 6: print(&q);
                break;
        default: break;
      }
   }while(op!=7);
   return 0;
}
```

Questions for Lab 5

1) Implement a circular queue of integers. Include functions insertcq, deletecq and displaycq.

2) Implement two circular queues of integers in a single array where first queue will run from 0 to N/2 and second queue will run from N/2+1 to N-1 where N is the size of the array.

3) Implement a queue with two stacks without transferring the elements of the second stack back to stack one. (use stack1 as an input stack and stack2 as an output stack).

Questions for Lab6

1) Implement an ascending priority queue.

   Note: An ascending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. If apq is an ascending priority queue, the operation pqinsert(apq,x) inserts element x into apq and pqmindelete(apq) removes the minimum element from apq and returns its value.

2) Implement a queue of strings using an output restricted dequeue (no deleteRight).
   Note: An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only, where as An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.

3) Write a program to check whether given string is a palindrome using a dequeue.