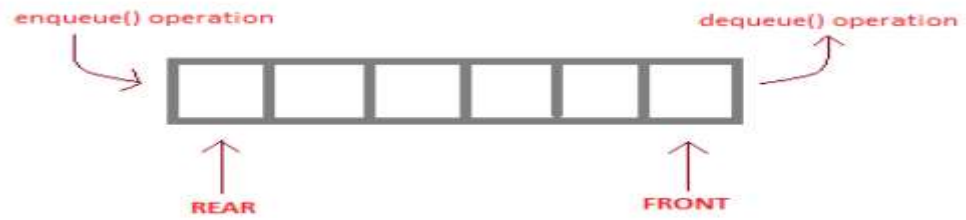
UNIT-IV
QUEUE

Queues: Introduction to queues: properties and operations, implementing queues using arrays and linked lists, Applications of queues in breadth-first search, scheduling, etc.
Deques: Introduction to deques (double-ended queues), Operations on deques and their applications.

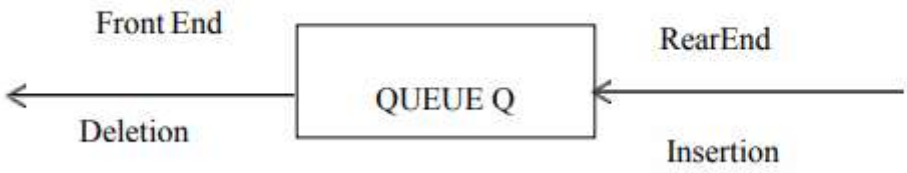
1. Introduction to queues

A queue is linear data structure and collection of elements.
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**.
The principle of queue is a **"FIFO" or "First-in-first-out"**.
It is similar to the ticket queue outside a cinema hall, where the first person entering the queueis the first person who gets the ticket.

Representation of queue



or



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

1.1Properties of queue

FIFO:

FIFO (First in First Out) data structure, which means that element inserted first will be removed first

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.

Enqueue:

Elements are added to the back of the queue. This operation is also known as "enqueue"

Dequeue:

Elements are removed from the front of the queue. This operation is also known as "dequeue"

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

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.

Linear data structure:

Queue follows a linear data structure, where elements are arranged in a sequence

Dynamic in size:

Queues can grow or shrinks as elements are added or removed

1.2Operations of queue

Fundamental operations performed on the queue are

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.
- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty

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.

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

peek()

This function helps to see the data at the front of the queue.

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full.

isempty()

As we are using single dimension array to implement queue, we just check for the front end . If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

2.Implementing queues using arrays

- Queues can be easily represented using linear arrays
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done,

Array Declaration of Queue:

```
#define ArraySize 5

int  Q [ ArraySize];

or

int Q [ 5 ];
```

Initial representation of Queue:

2.1Enqueue Operation

It is the process of inserting a new element at the Rear end of the Queue.

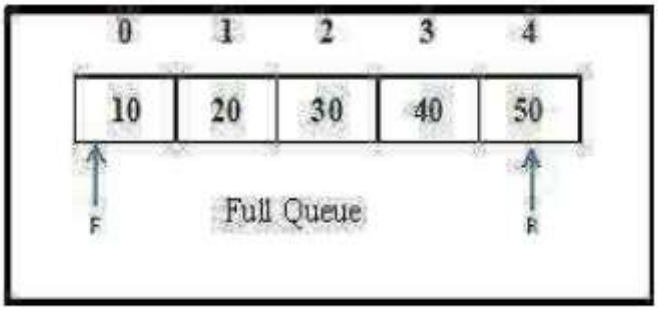
- It takes two parameters, Enqueue(X, Q).
- The elements X to be inserted at the Rear end of the Queue Q.
- Before inserting a new Element into the Queue, check for Full Queue.
- If the Queue is already Full, Insertion is not possible.
- Otherwise, Increment the Rear pointer by 1 and then insert the element X at the Rear end of the Queue.
- If the Queue is Empty, Increment both Front and Rear pointer by 1 and then insert the element X at the Rear end of the Queue.

Algorithm for enqueue operation

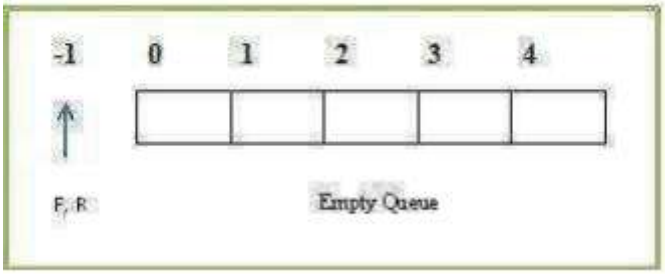
```
void EnQueue (int X , Queue Q)
 {
        if ( Rear = = Arraysize - 1)
         print ("Queue overflow  !!!!. Insertion not possible");
        else if (Rear = = - 1)
         {
        Front = Front + 1;
        Rear = Rear + 1;
        Q [Rear] = X;
         }
        else
        {
         Rear = Rear + 1;
        Q [Rear] = X;
         }
}
```

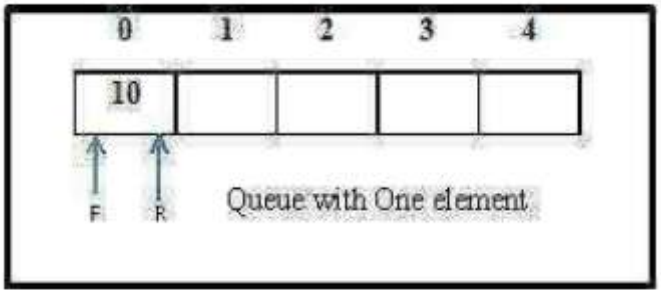
Example:

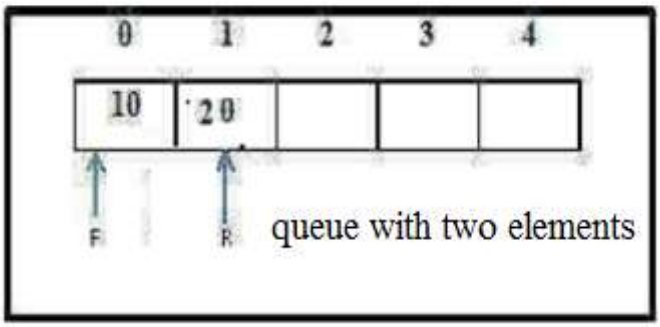
If queue is full



If queue is empty







2.2.DeQueue Operation

It is the process of deleting a element from the Front end of the Queue.

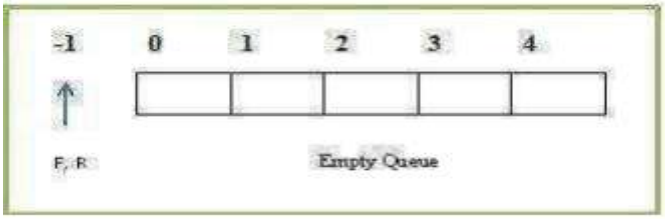
- It takes one parameter, DeQueue (Q). Always front element in the Queue will be deleted.
- Before deleting an Element from the Queue, check for Empty Queue.
- If the Queue is empty, deletion is not possible.
- If the Queue has only one element, then delete the element and represent the empty queue by updating Front = - 1 and Rear = - 1.
- If the Queue has many Elements, then delete the element in the Front and move the Front pointer to next element in the queue by incrementing Front pointer by 1.

Algorithm for enqueue operation

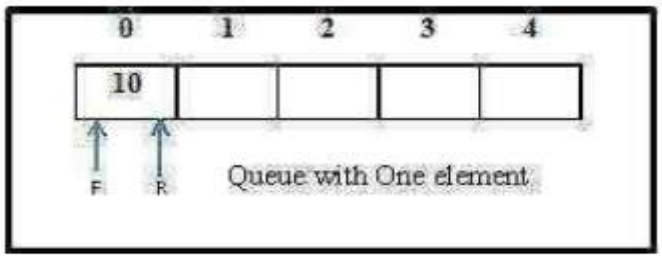
```
void DeQueue ( Queue Q )
{
        if ( Front = = - 1)
        print (" Empty Queue !. Deletion not possible " );
       else if( Front = = Rear )
        {
       X = Q [ Front ];
       Front = - 1;
       Rear = - 1;
        }
        else
       {
        X = Q [ Front ];
         Front = Front + 1 ;
        }
}
```

Example:

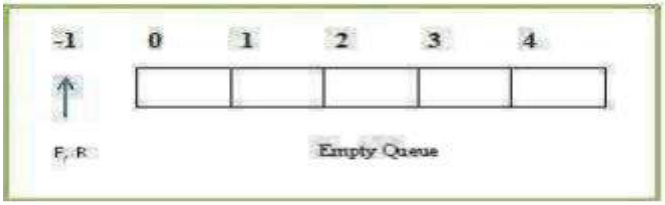
If queue is empty



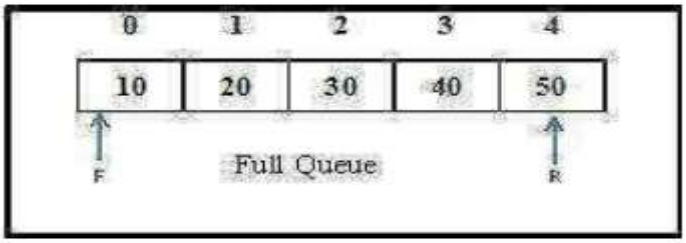
If queue is only one element



After dequeue



If queue contains more than one element



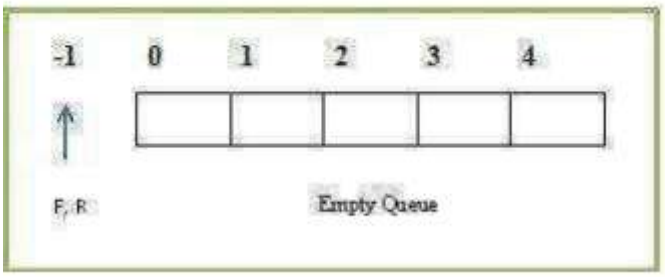
2.3Queue Empty Operation:

- Initially Queue is Empty.
- With Empty Queue, Front (F) and Rear (R) pointsto – 1.
- It is necessary to check for Empty Queue before deleting (DeQueue) an element from the Queue (Q).

Algorithm for isempty operation

```
int IsEmpty ( Queue Q )
{
        if( ( Front = = - 1) && ( Rear = = - 1 ) )
        return ( 1 );
}
```

Example



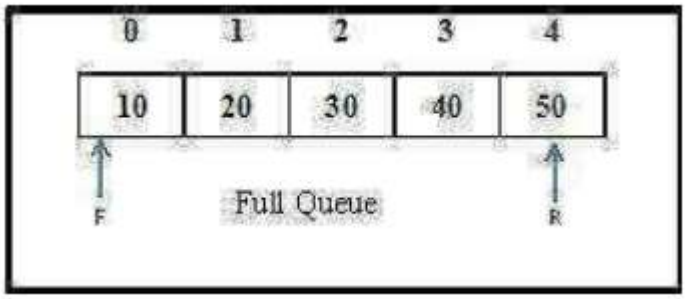
2.4.Queue Full Operation

- As we keep inserting the new elements at the Rear end of the Queue, the Queue becomes full. When the Queue is Full, Rear reaches its maximum Arraysize.
- For everyEnqueue Operation, we need to check for full Queue condition

Algorithm for isfull operation

```
int IsFull( Queue Q )
{
        if ( Rear = = ArraySize - 1 )
        return ( 1 );
}
```

Example:



2.5.Disaply operation

Algorithm for display operation

Step-1:START
Step-2: if front==rear then Write' Queue is empty'
Step-3: otherwise
3.1: for i=front to rear then
3.2: print 'queue[i]'
Step-4:STOP

QUEUE USING ARRAYS

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int queue[50],n,i,element,rear=-1,front=0;
void enqueue();
void dequeue();
void display();
void main()
{
        int ch;
        clrscr();
        printf("\nEnter the size of queue:");
        scanf("%d",&n);
        printf("\n*******MENU*********");

        do
        {
                printf("\n1.INSERT  2.DELETE  3.DISPLAY  4.EXIT");
                printf("\nEnter your choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: enqueue();
                                break;
```

```
                        case 2: dequeue();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(1);


                }
        }while(ch!=4);
        getch();
}

void enqueue()
{
        if(rear==n-1)
        {
                printf("\nQUEUE OVERFLOW");
        }
        else
        {
                printf("\nEnter the element to insert:");
                scanf("%d",&element);
                rear++;
                queue[rear] = element;
        }
}
void dequeue()
{
        if(front>rear)
        {
                printf("\nQUEUE UNNDERFLOW");
        }
        else
        {
                printf("\nDeleted Element is %d", queue[front]);
                front++;
        }
}
void display()
{
        if(front>rear)
        {
                printf("\nThere are no elements in the Queue to Display");
        }
```

```
        else
        {
                for(i=front;i<=rear;i++)
                {
                        printf("%d ",queue[i]);
                }
        }
}
```

3.Implementing Queue using Linked list:

- 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 alist.
- We use two pointers *front* and *rear* for our linked queue implementation.
- The linked queue looks as shown in figure:

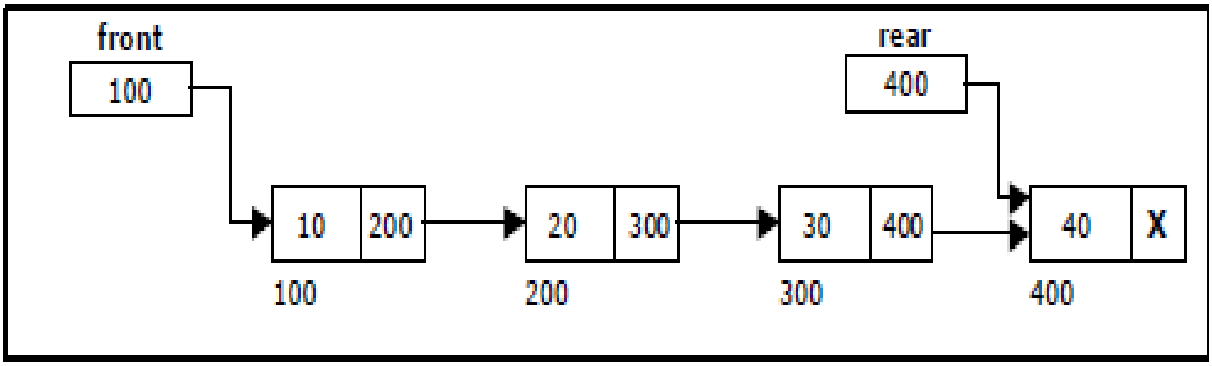


Figure : Linked Queue representation

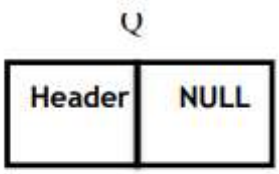
3.1.EnQueue Operation

- 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.

Algorithm for EnQueue Operation

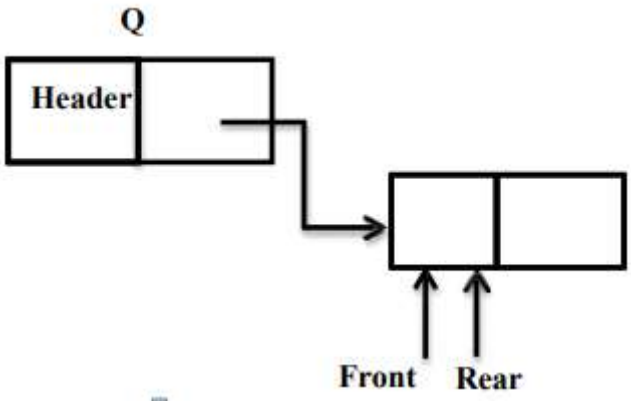
```
void EnQueue ( int X, Queue Q )
 {
        struct node *newnode;
        newnode = malloc (sizeof (struct node));
         if (Rear = = NULL)
         {
                newnode->data=X
                newnode->next=NULL;
                Q -> next = newnode;
                 Front = newnode;
                Rear = newnode;
        }
        else
        {
                newnode->data=X
                newnode->next=NULL;
                Rear->next = newnode;
                Rear = newnode;
}
}
```

Example:

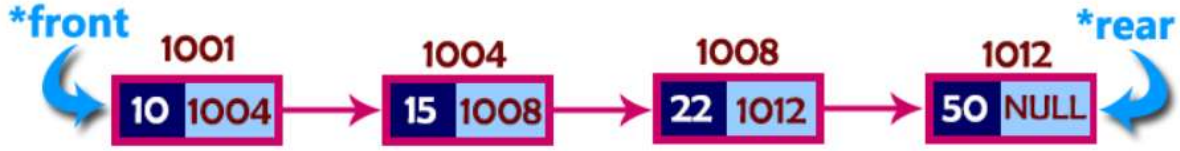


Empty Queue

Before Insertion



After inserting first element



After inserting more elements

3.2. 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 un changed but the data present in that address is now undefined.

Algortihm for DeQueue operation

```
void DeQueue ( Queue Q )
{
        struct node *temp;
        if ( Front = = NULL )
```

```
        Error ("Empty Queue!!! Deletion not possible." );
      else if (Front = = Rear)
       {
            temp = Front;
            Q -> next = NULL;
           Front = NULL;
           Rear = NULL;
            free ( temp );
      }
       else
       {
       temp = Front;
      Q -> next = temp -> next;
      Front =front->Next;
       free (temp);
      }
}
```

3.3display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

Step 1 - Check whether queue is **Empty** (**front** == **NULL**).

Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.

Step 4 - Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).

Step 5 - Finally! Display '**temp → data** ---> **NULL**'.

QUEUE USING LINKED LIST

```
#include<stdlib.h>
#include<stdio.h>
struct node
{
        int data;
        struct node *link;
};
struct node *front=NULL,*rear=NULL,*temp,*new_node;
void enqueue();
void dequeue();
void display();
void main()
{
        int choice;
        clrscr();
        printf("\n          MAIN MENU          ");
        printf("\n1.Enqueue 2.Dequeue 3.Display 4.Exit\n");
        printf("*********************************************");
        while(1)
        {
                printf("\nEnter your choice:");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: enqueue();
                                        break;
```

```
                        case 2: dequeue();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(1);
                                break;
                        defaulf:    printf("Invalid option");
                                break;
                }
        }
}
void enqueue()
{
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data:");
        scanf("%d",&new_node->data);
        new_node->link = NULL;
        if(front == NULL)
        {
                front=rear=new_node;
        }
        else
        {
                rear->link = new_node;
                rear = new_node;
        }
}
```

```
void dequeue()
{
	if(front==NULL)
	{
		printf("\nThere are no elements in the QUEUE.");
	}
	else
	{
		printf("\nDeleted Element is %d", front->data);
		temp = front;
		front = temp->link;
		free(temp);
		}
}
void display()
{
	if(front==NULL)
	{
		printf("There are no elements in the QUEUE");
	}
	else
	{	printf("\nElements in the QUEUE are:\n");
		temp = front;
		do
		{
			printf("%d ",temp->data);
			temp=temp->link;
```

```
            }while(temp != NULL);
        }
}
```

4.Applications of queues

4.1.Applications of queues in breadth-first search

What Is the Breadth-First Search Algorithm?

BFS is a graph traversal approach in which you start at a source node and layer by layer through the graph, analyzing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the next-level neighbor nodes.

Breadth-First Search uses a queue data structure to store the node and mark it as "visited" until it marks all the neighboring vertices directly related to it.

The queue operates on the First In First Out (FIFO) principle, so the node's neighbors will be viewed in the order in which it inserts them in the node, starting with the node that was inserted first.

How Does the BFS Algorithm Work?

Breadth-First Search uses a queue data structure technique to store the vertices. And the queue follows the First In First Out (FIFO) principle, which means that the neighbors of the node will be displayed, beginning with the node that was put first.

The transverse of the BFS algorithm is approaching the nodes in two ways.

- Visited node
- Not visited node

Breadth First Search Algorithm
Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices of a graph in a breadth ward motion, starting from a given source vertex. It uses a queue data structure to keep track of the vertices that are yet to be explored.

The algorithm works as follows:

Step 1: Begin by choosing a graph that you want to navigate.

Step 2: Select a starting vertex from which you want to traverse the graph.

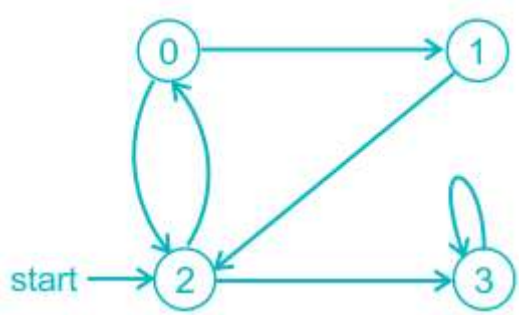
Step 3: Choose two data structures to use for traversing the graph: a visited array of size equal to the number of vertices in the graph, and a queue data structure.

Step 4: Start with the chosen vertex and add it to the visited array. Enqueue all adjacent vertices of the starting vertex into the queue.

Step 5: Remove the first vertex from the queue using the FIFO (first-in-first-out) concept, add it to the visited array, and enqueue all of its unvisited adjacent vertices into the queue.

Step 6: Repeat step 5 until the queue is empty and there are no unvisited vertices left in the graph.

Example:



Let us understand the algorithm of breadth first search with the help of an example.

Consider that we will begin the traversal of the following graph from vertex 2

- At vertex 0, we examine all of its adjacent vertices to determine where to traverse next.
- In this case, vertex 2 is an adjacent vertex of 0.
- It is important to mark visited vertices to avoid revisiting them and getting stuck in a non-terminating process.
- Multiple BFS traversals can be performed on a graph.
- For the given graph, there are two different BFS traversals: 2,3,0,1 and 2,0,3,1

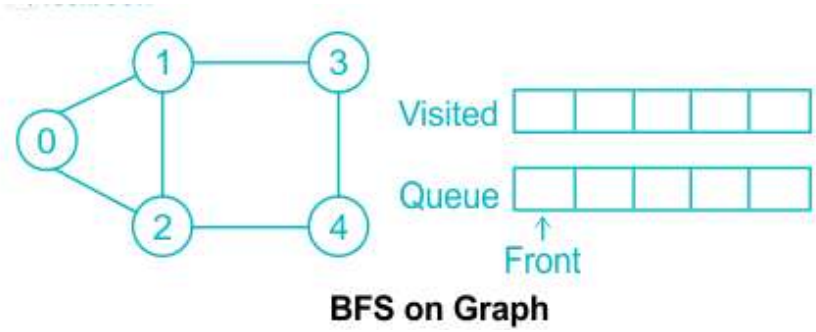
You can implement the BFS traversal by following the method described below:

1. Create a queue and add the starting vertex to it.
2. Create a visited array and mark the starting vertex as visited.
3. Repeat the following steps until the queue is empty:

- Remove the first vertex from the queue.
- Mark the vertex as visited.
- Add all the unvisited neighbors of the vertex to the queue.

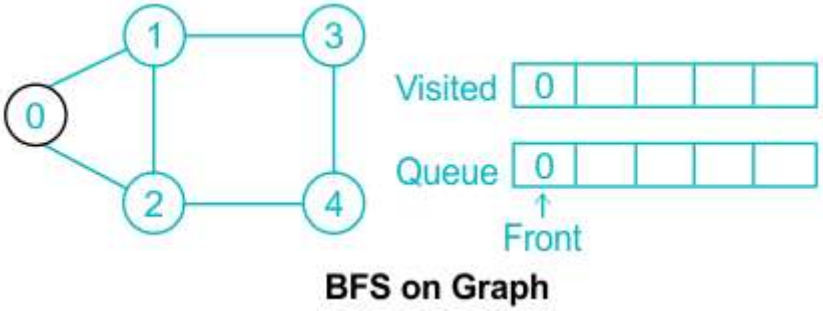
Example:

Step 1: Initially the queue and visited arrays are empty.



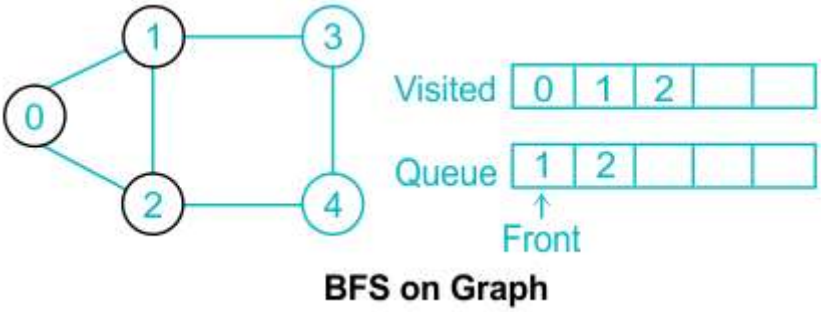
BFS on Graph

Step 2: Push node 0 into the queue and mark it visited



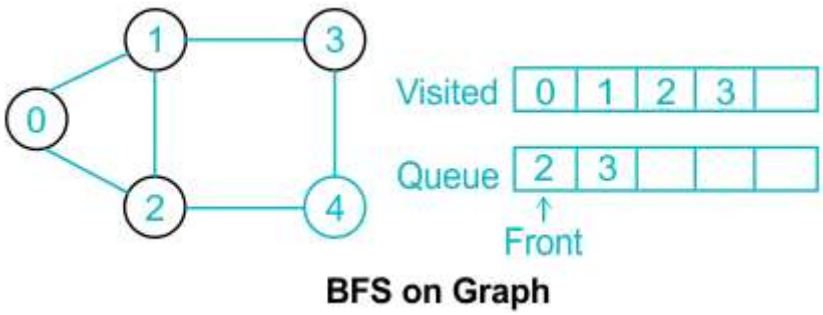
BFS on Graph

Step 3: Remove node 0 from the front of the queue and visit the unvisited neighbours and push them into the queue.



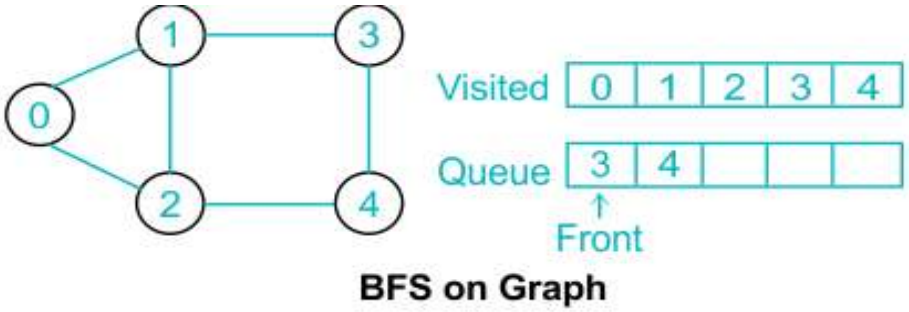
BFS on Graph

Step 4: Remove node 1 from the front of the queue and visit the unvisited neighbours and push them into the queue.



BFS on Graph

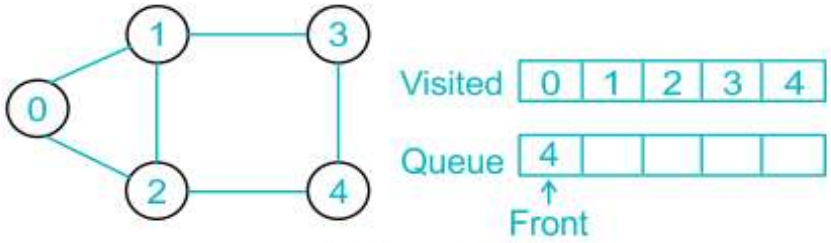
Step 5: Remove node 2 from the front of the queue and visit the unvisited neighbours and push them into the queue



BFS on Graph

Step 6: Remove node 3 from the front of the queue and visit the unvisited neighbours and push them into the queue.

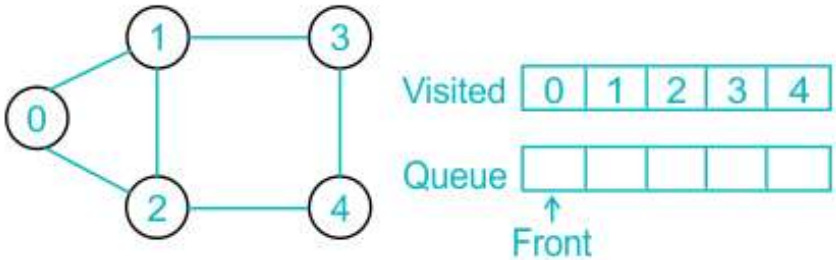
As we can see that every neighbour of node 3 is visited, so move to the next node that is in the front of the queue.



BFS on Graph

Step 7: Remove node 4 from the front of the queue and visit the unvisited neighbours and push them into the queue.

As we can see that every neighbour of node 4 is visited, so move to the next node that is in the front of the queue.



BFS on Graph

Now, the queue becomes empty, So, terminate these process of iteration.

4.2. Applications of queues in CPU Scheduling

This is one of the most common applications of queues where a single resource is shared among multiple consumers, or asked to perform multiple tasks.

Imagine you requested a task first before your friend, but your friend got the output first? Wouldn't you call it a corrupt system? (quite literally!)

CPU page replacement algorithm doesn't let that happen. It lets the operating system store all the processes in the memory in the form of a queue.

It makes sure that the tasks are performed on the basis of the sequence of requests. When we need to replace a page, the oldest page is selected for removal, underlining the First-In-First-Out (FIFO) principle.

Algorithm for the said function:

Start traversing the pages.

If a frame holds fewer pages than its full allocation capacity-

- Insert pages into the set one by one until the size of the set reaches capacity or all page requests are processed
- Update the pages in the queue to perform First Come-First Serve
- Increment page fault

Else

If the current page is present in the set, do nothing.

Else

- Remove the first page from the queue.
- Replace it with the current page in the string and store the current page in the queue.
- Increment page faults.

Return page faults.

II.Deque:

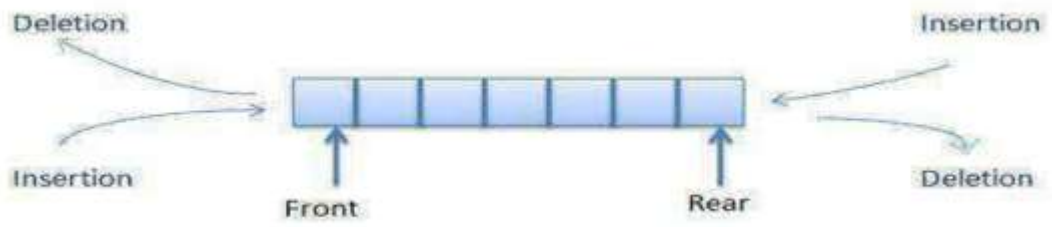
1.Introduction to deque

In the queues 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***.

Below figure shows the representation of a deque.



Example:

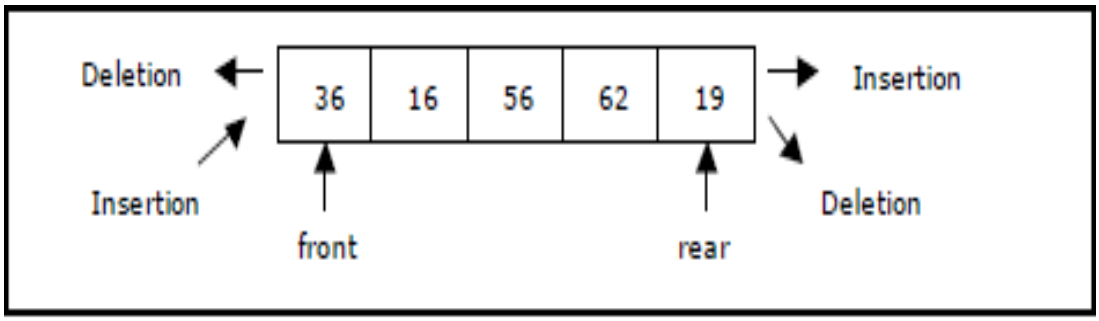


Figure Representation of a deque.

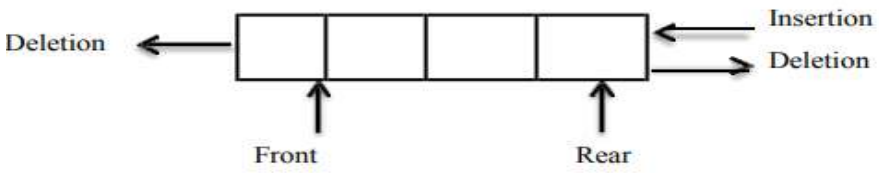
There are two variations of deque.

They are:

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

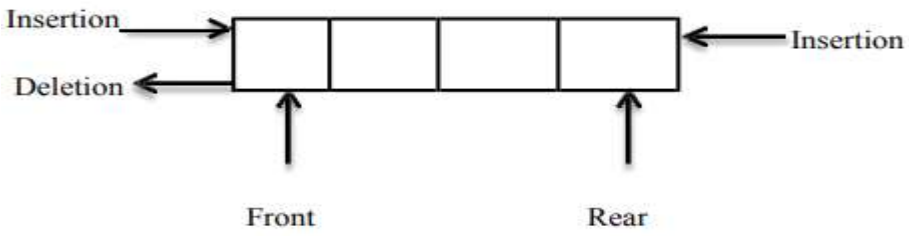
1.Input restricted deque (IRD)

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



2.Output restricted deque (ORD)

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



2.Operations of deque

deque provides four operations.

- 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.

Below Figure shows the basic operations on a deque.

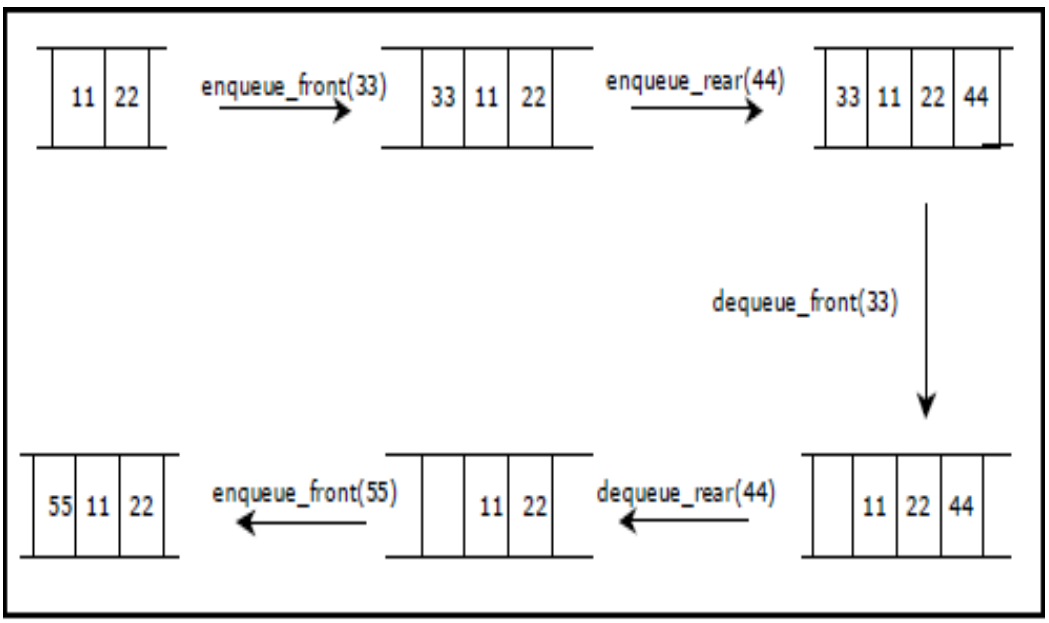


Figure ` `. Basic operations on deque

Insertion at the rear end

Step 1 : Check for the overflow condition.

Step 2 : If it is true, display that the queue is full

Step 3 : Otherwise, If the rear and front pointers are at the initial values (-1). Increment both the pointers. Goto step 5.

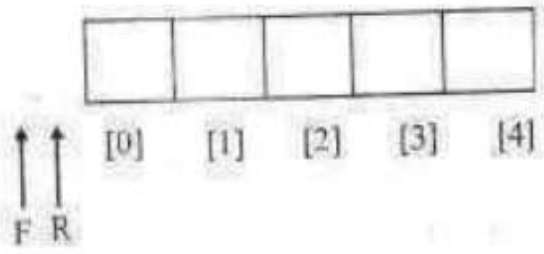
Step 4 : Increment the rear pointer

Step 5 : Assign the value to Q[rear]

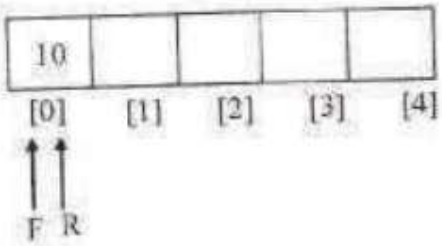
```
void Insert_Rear (int X, DEQUE DQ)
{
        if( Rear = = Arraysize - 1)
        printf("que is overflow!!!! Insertion not possible");
        else if( Rear = = -1)
        {
        Front = Front + 1;
        Rear = Rear + 1;
        DQ[ Rear ] = X;
          }
        else
        {
        Rear = Rear + 1;
        DQ[ Rear ] = X;
          }
}
```

Example:

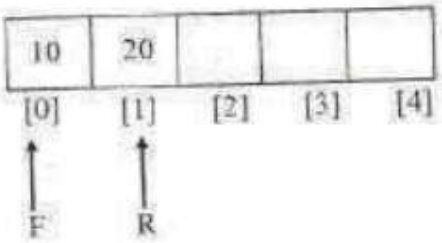
Insertion at the rear end



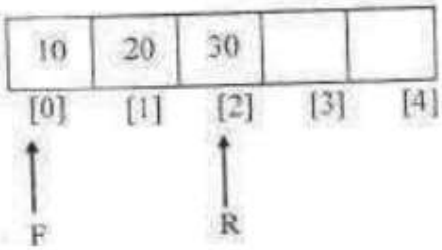
Insert_rear (10)



Insert_rear (20)



Insert_rear (30)



Insertion at front end

Step 1 : Check the front pointer, if it is in the first position (0) then display an error message that the value cannot be inserted at the front end.

Step 2 : Otherwise, decrement the front pointer

Step 3 : Assign the value to Q[front]

```
void Insert_Front ( int X, DEQUE DQ )
 {
        if( Front = = 0 )
        Error("Element present in Front!!!!! Insertion not possible");
       else if(Front = = -1)
```

```
        {
        Front = Front + 1;
        Rear = Rear + 1;
        DQ[Front] = X;
          }
       else
       {
       Front = Front - 1;
        DQ[Front] = X;
  }
 }
```

Example:

1.Check the position of front.

2.If front < 1, reinitialize front = n-1 (last index)

3.Else, decrease front by 1.

4. Add the new key 5 into array[front].

Deletion from Front End :

Step 1 : Check for the underflow condition. If it is true display that the queue is empty.

Step 2 : Otherwise, delete the element at the front position, by asigning X as Q[front]

Step 3 : If the rear and front pointer points to the same position (ie) only one value is present, then reinitialize both the pointers.

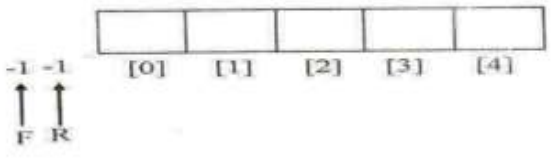
Step 4 : Otherwise, Increment the front pointer

```
void Delete_Front(DEQUE DQ)
{
        if(Front = = - 1)
        Error("Empty queue!!!! Deletion not possible");
        else if( Front = = Rear )
        {
        X = DQ[ Front];
                Front = - 1;
        Rear = - 1;
          }
        else
        {
         X = DQ [ Front ];
        Front = Front + 1;
         }
}
```

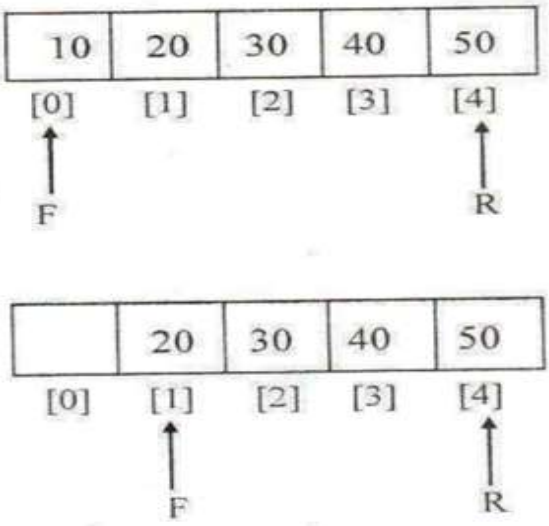
Example:

Deletion from front end

Dequeue_front ()



Q - is empty



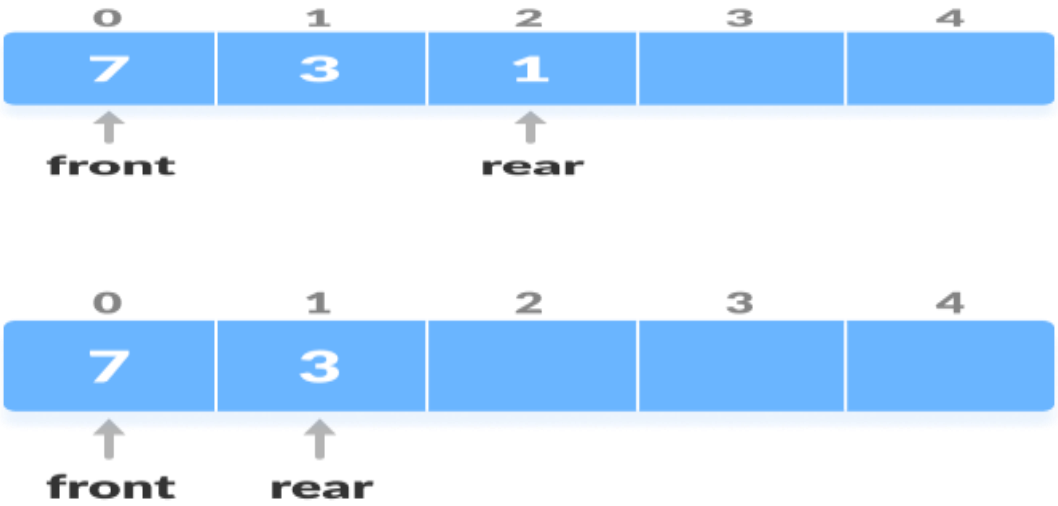
Delete from the Rear

This operation deletes an element from the rear.

1. Check if the deque is empty.
2. If the deque is empty (i.e. front = -1), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1, else follow the steps below.
4. If rear is at the front (i.e. rear = 0), set go to the front rear = n - 1.
5. Else, rear = rear - 1

```
void Delete_Rear(DEQUE DQ)
{
        if( Rear = = - 1)
        Error("Empty queue!!!! Deletion not possible");
        else if( Front = = Rear )
        {
        X = DQ[ Rear ];
        Front = - 1;
```

```
        Rear = - 1;
       }
        else
       {
       X = DQ[ Rear ];
        Rear = Rear - 1;
         }
}
```




```
// Deque implementation in C
#include <stdio.h>
#define MAX 10
void addFront(int *, int, int *, int *);
void addRear(int *, int, int *, int *);
int delFront(int *, int *, int *);
int delRear(int *, int *, int *);
void display(int *);
int count(int *);
```

```
int main()
 {
            int arr[MAX];
            int front, rear, i, n;
            front = rear = -1;
           for (i = 0; i < MAX; i++)
            arr[i] = 0;
            addRear(arr, 5, &front, &rear);
           addFront(arr, 12, &front, &rear);
            addRear(arr, 11, &front, &rear);
            addFront(arr, 5, &front, &rear);
           addRear(arr, 6, &front, &rear);
            addFront(arr, 8, &front, &rear);
            printf("\nElements in a deque: ");
          display(arr);
            i = delFront(arr, &front, &rear);
            printf("\nremoved item: %d", i);
            printf("\nElements in a deque after deletion: ");
           display(arr);
            addRear(arr, 16, &front, &rear);
            addRear(arr, 7, &front, &rear);
            printf("\nElements in a deque after addition: ");
            display(arr);
            i = delRear(arr, &front, &rear);
            printf("\nremoved item: %d", i);
            printf("\nElements in a deque after deletion: ");
            display(arr);
```

```
        n = count(arr);
        printf("\nTotal number of elements in deque: %d", n);
}
void addFront(int *arr, int item, int *pfront, int *prear)
 {
        int i, k, c;
        if (*pfront == 0 && *prear == MAX - 1)
        {
                printf("\nDeque is full.\n");
                return;
        }
        if (*pfront == -1)
        {
                *pfront = *prear = 0;
                arr[*pfront] = item;
                return;
        }
        if (*prear != MAX - 1)
        {
          c = count(arr);
          k = *prear + 1;
                for (i = 1; i <= c; i++)
        {
                        arr[k] = arr[k - 1];
                        k--;
                }
                arr[k] = item;
```

```
                *pfront = k;
                 (*prear)++;
          }
        else
        {
                (*pfront)--;
                 arr[*pfront] = item;
          }
}

void addRear(int *arr, int item, int *pfront, int *prear)
 {
          int i, k;
          if (*pfront == 0 && *prear == MAX - 1)
          {
                printf("\nDeque is full.\n");
                return;
          }
          if (*pfront == -1)
        {
                 *prear = *pfront = 0;
           arr[*prear] = item;
           return;
          }
          if (*prear == MAX - 1)
        {
                k = *pfront - 1;
```

```
                for (i = *pfront - 1; i < *prear; i++)
        {
                        k = i;
                        if (k == MAX - 1)
                                arr[k] = 0;
                        else
                                arr[k] = arr[i + 1];
                }
                (*prear)--;
                (*pfront)--;
          }
        (*prear)++;
        arr[*prear] = item;
}
int delFront(int *arr, int *pfront, int *prear)
 {
          int item;
          if (*pfront == -1)
        {
                printf("\nDeque is empty.\n");
                  return 0;
          }
          item = arr[*pfront];
          arr[*pfront] = 0;

          if (*pfront == *prear)
                *pfront = *prear = -1;
```

```
            else
                    (*pfront)++;
  return item;
}
int delRear(int *arr, int *pfront, int *prear)
 {
            int item;
            if (*pfront == -1)
           {
                    printf("\nDeque is empty.\n");
                   return 0;
            }
            item = arr[*prear];
            arr[*prear] = 0;
            (*prear)--;
           if (*prear == -1)
                   *pfront = -1;
  return item;
}
void display(int *arr)
 {
            int i;
            printf("\n front:  ");
            for (i = 0; i < MAX; i++)
           printf("  %d", arr[i]);
            printf("  :rear");
}
```

```
int count(int *arr)
{
        int c = 0, i;
        for (i = 0; i < MAX; i++)
        {
                if (arr[i] != 0)
                c++;
        }
       return c;
}
```

OUPUT:

Elements in a deque:

front: 9 5 12 5 11 6 :rear

removed item: 9

Elements in a deque after deletion:

front: 0 5 12 5 11 6 :rear

Deque is full.

Elements in a deque after addition:

front: 5 12 5 11 6 16 :rear

removed item: 16

Elements in a deque after deletion:

front: 5 12 5 11 6 0 :rear

Total number of elements in deque: 5

Note: in the above execution 0 indicates empty cell