Queue

A queue is a linear list in which elements can be added at one end and elements can be removed only at other end. So the information in this list is processed in same order as it was received .Hence queue is called a FIFO structure.(First In First Out).

Ex: people waiting in a line at a bus stop.

The first person in queue is the first person to take bus. Whenever new person comes he joins at end of the queue.

Implementation of Queue

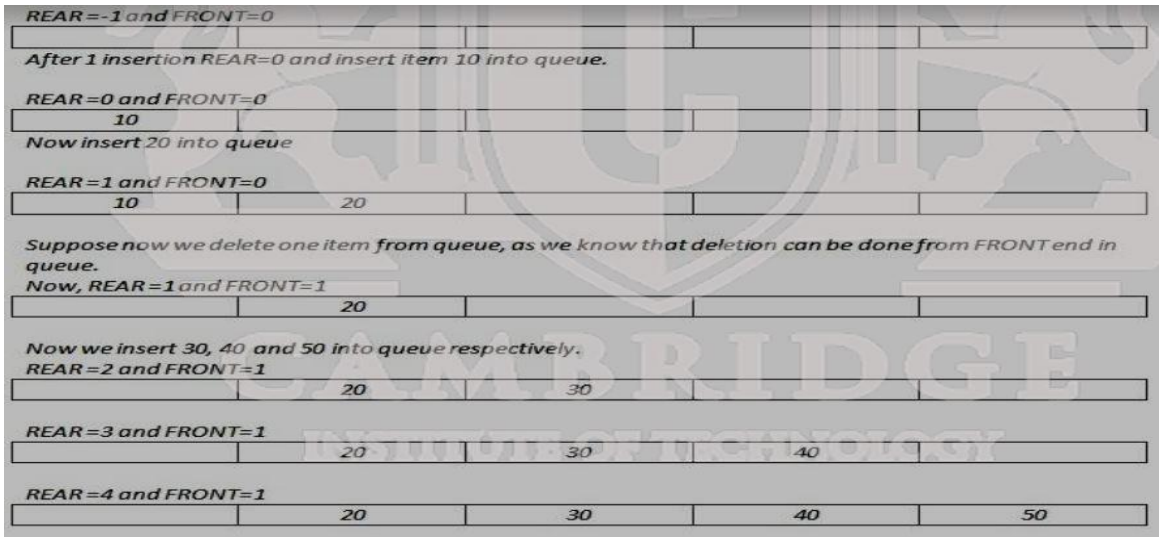
Queue can be implementing by two ways:

- Array implementation
- Linked list implementation

Array Representation of Queue

In Array implementation FRONT pointer initialized with 0 and REAR initialized with - 1.Consider the implementation: - If there are 5 items in a Queue,

Note: In case of empty queue, front is one position ahead of rear : FRONT = REAR + 1;.This is the queue underflow condition. The queue is full when REAR =N-1.This is the queue overflow condition.



The figure above ,the last case after insertion of three elements, the rear points to 4, and hence satisfies the overflow condition although the queue still has space to accommodate one more element .This problem can be overcome by making the rear pointer reset to the starting position in the queue and hence view the array as a circular representation. This is called a circular queue.

Implementation of queue using arrays

```
# include <conio.h>
# define MAX 5
int Q[MAX];
int front=0, rear=-1;
void insertQ() //Enqueue
{
int data;
if(rear == MAX-1)
{ printf("\n Linear Queue is full");
return; }
printf("\n Enter data: ");
scanf("%d", &data);
```

```
Q[++rear] = data;
printf("\n Data Inserted in the Queue ");
}
void deleteQ() // dequeue
{ if( front>rear) //OR front=rear +1
{
printf("\n\n Queue is Empty.."); return;
}
printf("\n Deleted element from Queue is %d", Q[front]);
front++;
}
void displayQ()
{ int i;
if(front >rear)
{ printf("\n\n\t Queue is Empty"); return; }
printf("\n Elements in Queue are: ");
for(i = front; i < rear; i++)
printf("%d\t", Q[i]);
}
```

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- 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.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue

Peek()

This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
return queue[front]
end procedure
```

Implementation of peek() function in C programming language

```
int peek()
{
   return queue[front];
}
```

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. In case we maintain the queue in a circular linked-list, the algorithm will differ.

Algorithm

```
begin procedure isfull
   if rear equals to MAXSIZE
      return true
```

```
  else
    return false
  endif
end procedure
```

Implementation of isfull() function in C programming language

```
bool isfull()
{
  if(rear == MAXSIZE - 1)
    return true;
  else
    return false;
}
```

isempty()

Algorithm

```
begin procedure isempty
  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif
end procedure
```

If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

C programming code

```
bool isempty()
{
  if(front < 0 || front > rear)
    return true;
  else
```

return false;

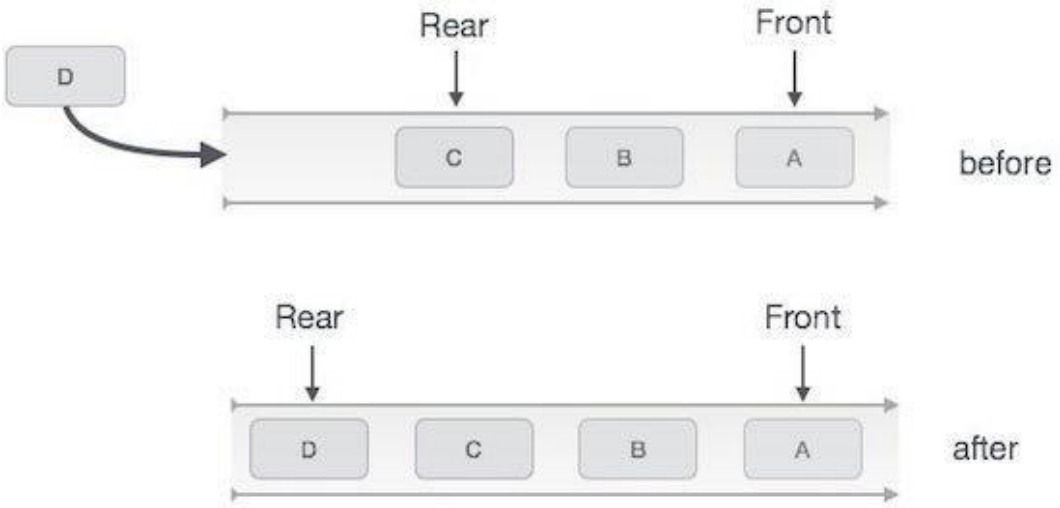
}

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue,

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success.



Queue Enqueue

Algorithm for enqueue operation

procedure enqueue(data)

if queue is full

```
    return overflow
  endif
rear ← rear + 1
  queue[rear] ← data
  return true
end procedure
```

Implementation of enqueue() in C programming language

```
int enqueue(int data)
  if(isfull())
    return 0;
  rear = rear + 1;
  queue[rear] = data;
  return 1;
end procedure
```

Dequeue Operation

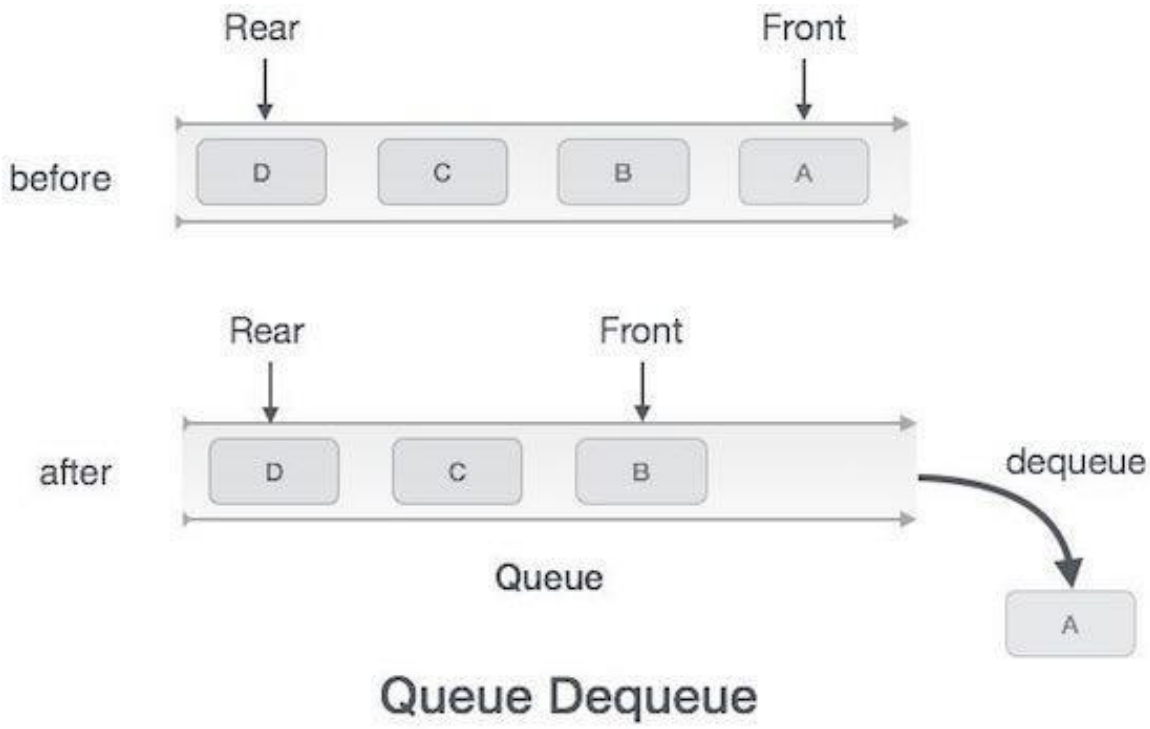
Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

- Step 1 − Check if the queue is empty.
- Step 2 − If the queue is empty, produce underflow error and exit.
- Step 3 − If the queue is not empty, access the data where front is pointing.
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.

Algorithm for dequeue operation

```
procedure dequeue
  if queue is empty
    return underflow
  end if
  data = queue[front]
```

```
   front ← front + 1
   return true
end procedure
```



Queue Dequeue

Implementation of dequeue() in C programming language

```
int dequeue()
{
   if(isempty())
      return 0;
int data = queue[front];
   front = front + 1;
   return data;
}
```

Circular Queue

In a normal Queue Data Structure, elements can be inserted until queue becomes full. But once if queue becomes full, no more elements can be inserted until all the elements are deleted from the queue. For example consider the queue below... After inserting all the elements into the queue.

Now consider the following situation after deleting three elements from the queue...

This situation also says that Queue is Full and the new element cannot be inserted because, 'rear' is still at last position. In above situation, even though we have empty positions in the queue they cannot be used to insert new element. This is the major drawback in normal queue data structure. This is overcome in circular queue data structure.

What's a Circular Queue?

A circular queue is linear data structure 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.

In any queue it is necessary that:

- Before insertion, fullness of Queue must be checked (for overflow).
- Before deletion, emptiness of Queue must be checked (for underflow).

Operation of Circular Queue:

- Insertion
- Deletion
- Display the queue content

Algorithm for Circular Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

Step 1: Include all the header files which are used in the program and define a

constant 'SIZE' with specific value.

Step 2: Declare all user defined functions used in circular queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (int cQueue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = 1, rear = -1)

Step 5: Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

Array Implementation of Circular Queue

```
#define MAX 4
int CQ[MAX], n;
int r = -1;
int f = 0,ct=0;
void enqueue() //function to insert an element to queue
{
int key;
if (ct == n )
{
printf("Queue Overflow\n");
return;
}
printf("\nenter the element for adding in queue : ");
r = (r+1)%n;
scanf("%d", &key);
CQ[r]=key;
ct++;
}
void dequeue() //function to remove an element from queue
```

```
{
if (ct == 0)
{
printf("Queue Underflow\n");
return ;
}
printf("Element deleted from queue is : %d\n", CQ[f]);
f=(f+1)%n;
ct--;
}
void display()
{
int i,k=f;
if (ct == 0)
{
printf("Queue is empty\n");
return;
}
printf("contents of Queue are :\n");
for (i = 0; i < ct; i++)
{
printf("%d\t", CQ[k]);
k=(k+1)%n;
}}
```

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following

steps to insert an element into the circular queue...

Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.

Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

Step 1: Check whether queue is EMPTY. (front == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.

Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value

and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

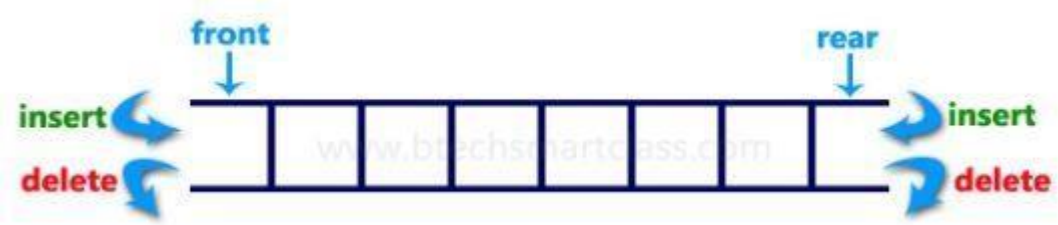
Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i <= SIZE - 1' becomes FALSE.

Step 6: Set i to 0.

Step 7: Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Double ended queue (dequeue)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

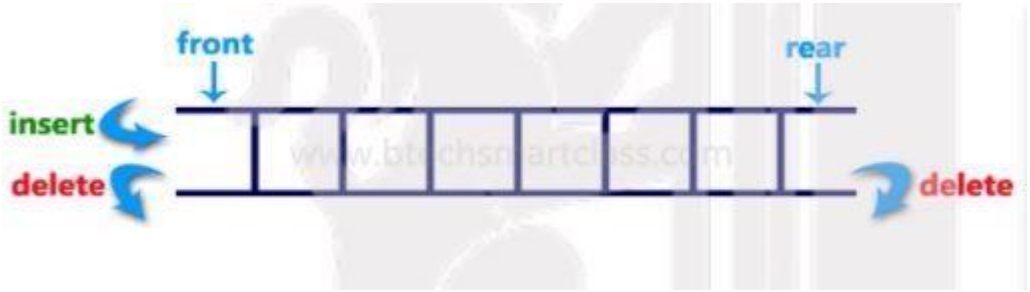


Double Ended Queue can be represented in TWO ways, those are as follows...

- ✓ **Input Restricted Double Ended Queue**
- ✓ **Output Restricted Double Ended Queue**

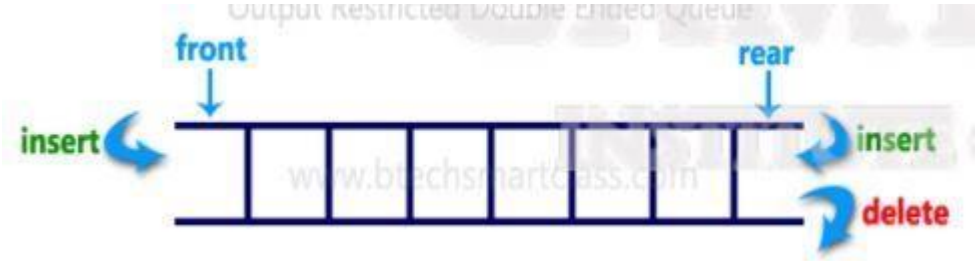
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Deque is a variation of queue data structure, pronounced “dequeue”, which stands for double- ended queue. In a deque values can be inserted at either the front or the back, A collection of peas in a straw is a good example.. Queues and deques are used in a number of ways in computer applications. A printer, for example, can only print one job at a time. During the time it is printing there may be many different requests for other output to be printed. To handle this printer will maintain a queue of pending print tasks. Since you want the results to be produced in the order that they are received, a queue is the appropriate data structure.

For a deque the defining property is that elements can only be added or removed from the end points. It is not possible to add or remove values from the middle of the collection.

Operations on deque

- Insertfront
- Deletefront

- Insertrear
- Deleterear

Priority Queue

Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a one difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.

getHighestPriority(): Returns the highest priority item.

deleteHighestPriority(): Removes the highest priority item.

- ✓ **insert()** operation can be implemented by adding an item at end of array in O(1) time.
- ✓ **getHighestPriority()** operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.
- ✓ **deleteHighestPriority()** operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

There are two types of priority queues they are as follows...

- ✓ **Max Priority Queue**

- ✓ **Min Priority Queue**

Max Priority Queue

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

The following are the operations performed in a Max priority queue...

- ✓ isEmpty() - Check whether queue is Empty.
- ✓ insert() - Inserts a new value into the queue.
- ✓ findMax() - Find maximum value in the queue.
- ✓ remove() - Delete maximum value from the queue.

Min Priority Queue Representations

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue.

The following operations are performed in Min Priority Queue...

- ✓ isEmpty() - Check whether queue is Empty.
- ✓ insert() - Inserts a new value into the queue.
- ✓ findMin() - Find minimum value in the queue.

Applications of Queue:

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