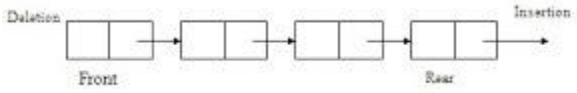
TYPES OF QUEUES

Different types of queues:

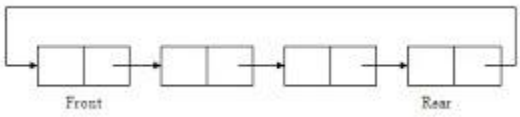
Queue can be of four types:

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended queue)

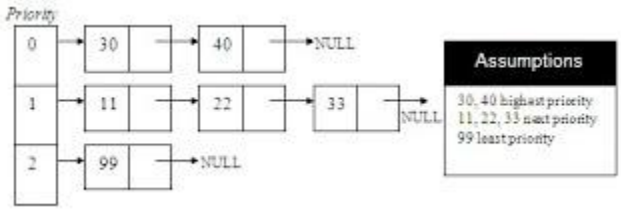
1. **Simple Queue:** In Simple queue Insertion occurs at the rear of the list, and deletion occurs at the front of the list.



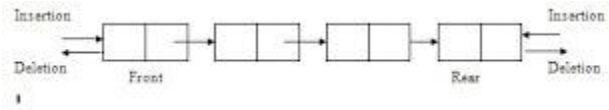
2. **Circular Queue :** A circular queue is a queue in which all nodes are treated as circular such that the first node follows the last node.



3. **Priority Queue:** A priority queue is a queue that contains items that have some preset priority. When an element has to be removed from a priority queue, the item with the highest priority is removed first



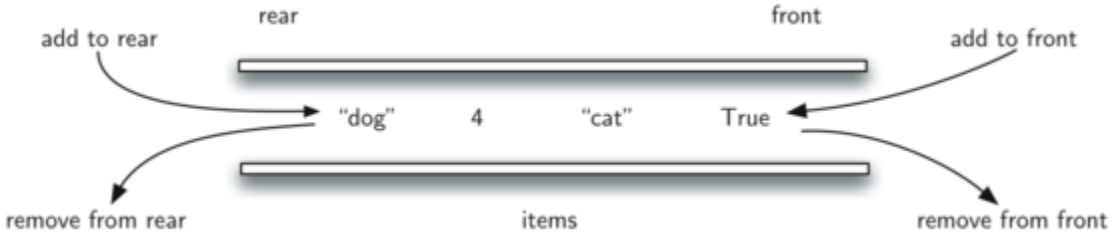
4. **Dequeue (Double Ended queue):** In dequeue(double ended queue) Insertion and Deletion occur at both the ends i.e. front and rear of the queue.



What Is a Deque?

A **deque**, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.



Double-Ended Queue

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

There are two variants of a double-ended queue they include

(a) Input restricted Dequeue – Insertions can be done only at one of the ends, while deletions can be done from both ends.

(b) Output restricted dequeue – Deletions can be done only at one ends while insertions can be done on both ends .

Algorithm for Insertion at rear end

```
Step -1: [Check for overflow]
    if(rear==MAX)
            Print("Queue is Overflow”);
       return;
Step-2: [Insert element]
    else
            rear=rear+1;
            q[rear]=no;
            [Set rear and front pointer]
            if rear=0
                    rear=1;
            if front=0
                    front=1;
Step-3: return
```

Implementation of Insertion at rear end

```
void add_item_rear()
{
    int num;
    printf("\n Enter Item to insert : ");
    scanf("%d",&num);
    if(rear==MAX)
    {
            printf("\n Queue is Overflow");
            return;
```

```
    }
    else
    {
            rear++;
            q[rear]=num;
            if(rear==0)
                    rear=1;
            if(front==0)
                    front=1;
    }
}
```

Algorithm for Insertion at font end

Step-1 : [Check for the front position]
    if(front<=1)
            Print (“Cannot add item at front end”);
    return;
Step-2 : [Insert at front]
    else
            front=front-1;
            q[front]=no;
Step-3 : Return

Implementation of Insertion at font end

```
void add_item_front()
{
    int num;
    printf("\n Enter item to insert:");
    scanf("%d",&num);
    if(front<=1)
    {
```

```
            printf("\n Cannot add item at front end");
            return;
        }
        else
        {
            front--;
            q[front]=num;
        }
    }
```

Algorithm for Deletion from front end

Step-1 [Check for front pointer]

if front=0

print(" Queue is Underflow”);

return;

Step-2 [Perform deletion]

else

no=q[front];

print(“Deleted element is”,no);

[Set front and rear pointer]

if front=rear

front=0;

rear=0;

else

front=front+1;

Step-3 : Return

Implementation of Deletion from front end

```
void delete_item_front()
{
    int num;
```

```
    if(front==0)
    {
        printf("\n Queue is Underflow\n");
        return;
    }
    else
    {
        num=q[front];
        printf("\n Deleted item is %d\n",num);
        if(front==rear)
        {
            front=0;
            rear=0;
        }
        else
        {
            front++;
        }
    }
}
```

Algorithm for Deletion from rear end

Step-1 : [Check for the rear pointer]

if rear=0

print(“Cannot delete value at rear end”);

return;

Step-2: [perform deletion]

else

no=q[rear];

[Check for the front and rear pointer]

if front= rear

```
            front=0;
            rear=0;
        else
            rear=rear-1;
            print(“Deleted element is”,no);
    Step-3 : Return
```

Implementation of Deletion from rear end

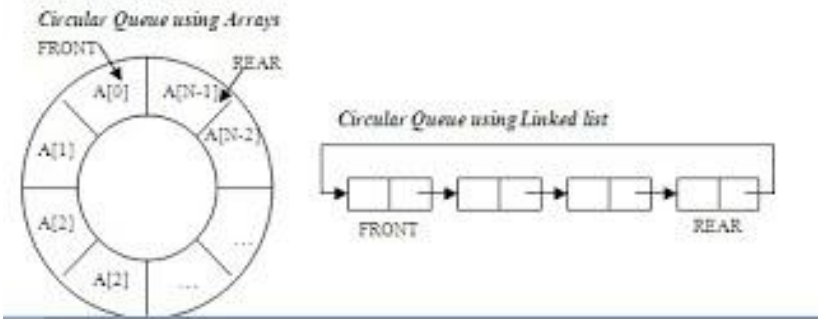
```
    void delete_item_rear()
    {
        int num;
        if(rear==0)
        {
            printf("\n Cannot delete item at rear end\n");
            return;
        }
        else
        {
            num=q[rear];
            if(front==rear)
            {
                front=0;
                rear=0;
            }
            else
            {
                rear--;
                printf("\n Deleted item is %d\n",num);
            }
        }
    }
```

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- **Deque()** creates a new deque that is empty. It needs no parameters and returns an empty deque.
- **addFront(item)** adds a new item to the front of the deque. It needs the item and returns nothing.
- **addRear(item)** adds a new item to the rear of the deque. It needs the item and returns nothing.
- **removeFront()** removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- **removeRear()** removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- **isEmpty()** tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- **size()** returns the number of items in the deque. It needs no parameters and returns an integer.

Circular queues:

A circular queue is a queue in which all nodes are treated as circular such that the first node follows the last node.



In a linked list circular queue, the rear node always has the reference of the front node. Even if the front node is removed the rear node has the reference of the new front node.

The common operations on circular queue are:

- **qstore(:** adding elements at the end of a circular queue.
- **qdelete:** removing elements from the front of the circular queue.
- **isfull:** isfull finds out whether the circular queue is full
- **isempty:** isempty finds out whether the circular queue is empty
- **create:** to create an empty circular queue

Applications of circular queue:

Adding large integers: Large integers can be added very easily using circular queues. Here the rightmost digit is placed in the front node and leftmost digit is placed in the rear node.

Memory management: The unused memory locations in the case of ordinary queues can be utilized in circular queues.

Computer controlled traffic system: In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.

Priority Queue

A Priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

1. An element with higher priority is processed before an element with a lower priority.
2. Two elements with the same priority are processed on a first come first served basis

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating system to execute the highest priority process first.

Implementation

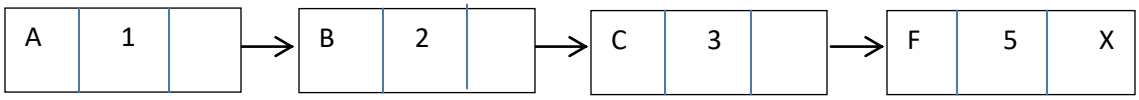
There are two ways to implement a priority queue. We can either use a sorted list to store the elements so that when an element has to be taken out , the queue will not have to be searched for the element with the highest priority or we can use an unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed. While a sorted list takes O(n) time to insert an element in the list, it takes only O(1) time to delete an element. On the contrary, an unsorted list will take o(1) time to insert an element and O(n) time to delete an element from the list.

Practically, both these techniques are in efficient and usually a blend of these two approaches is adopted that takes roughly o(log n) time or less.

Linked List Representation of Priority Queue

When a priority Queue is implemented using a linked List, then every node of the list will have three parts (a) the Information or data part (b) the priority

number of the element, and (c) the address of the next element. If you are using a sorted linked list, then the element with the higher priority will precede with the lower priority.



Lower priority number means higher priority. For example, if there are two elements A and B, and A has priority number a and B has a priority number 5, then a will be processed before B as it has higher priority. Element with a higher priority comes before the element with a lower priority. However when two elements have the same priority the elements are arranged and processed on FCFS principle.

Insertion

When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element.