Queue

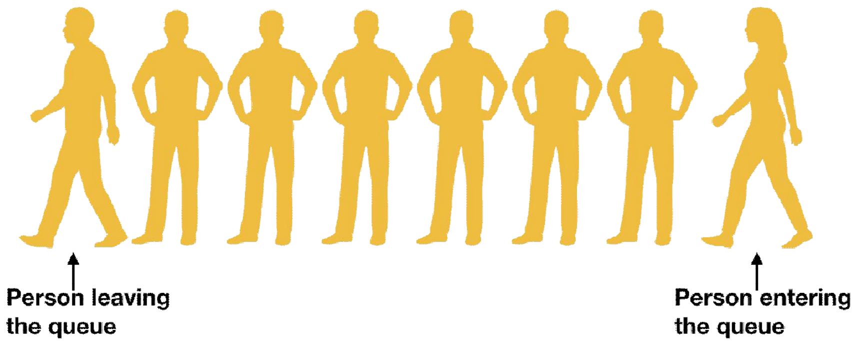
Introduction

- In general, Queue is a line of person waiting for their turn at some service counter like ticket window at cinema hall, at bus stand or at railway station etc. The person who comes first, he/she gets the service first.

Similarly in data structure,

- Queue is a linear list in which insertions can take place at one end of list, known as **rear** of the list, and deletions can take place at the other end of list, known as **front** of list.
- Nature of queue is First Come First Serve (FCFS) or First In First Out (FIFO)

Example – Printer task queue, keystroke queue etc.



Basic Operations

1.	CreateQueue –	Creates an empty queue
2.	Enqueue –	Inserts element into queue
3.	Dequeue –	Deletes element from queue
4.	IsEmpty –	Checks the emptiness of queue
5.	IsFull –	Checks the fullness of queue

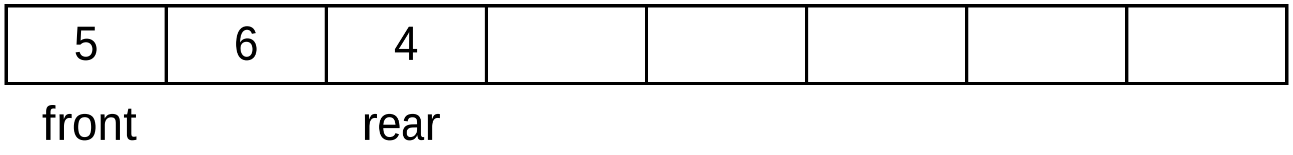
Array as Linear Queue

Array Representation of Queue

Queue is defined by its two pointers namely front and rear

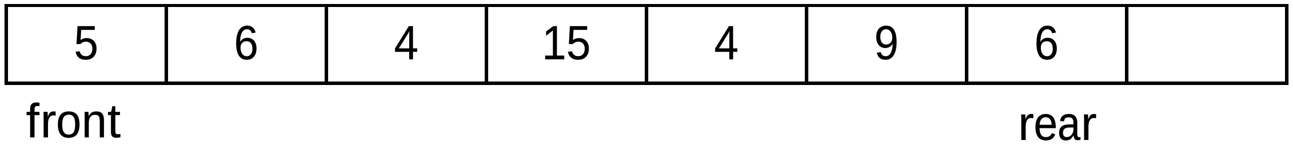
1. If queue is empty then front=-1 and rear=-1
2. If queue is full then rear = SIZE-1 and front=0 where, SIZE is the size of array used as queue

Representation (Cont….)



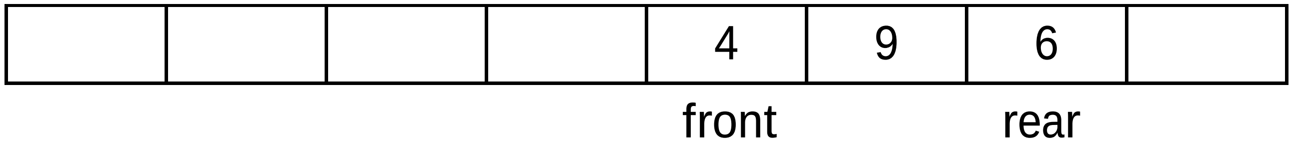
front=0 and rear=2

Insert (Enqueue) 15, 4,9,6



front=0 and rear=6

Delete (Dequeue) 4 times



front=4 and rear=6

Create queue

C code:

```
struct queue{
        int array[SIZE];
        int front, rear;
};
```

Create queue:

Algorithm Create_Queue(struct queue *Q): This algorithm creates an empty queue i.e. it initializes rear = -1 and front = -1.

1. Q -> front =-1
2. Q -> rear = -1

Enqueue

This algorithm inserts an element into queue Q. Item is the element which is to be inserted.

```
Algorithm Enqueue(struct queue *Q, int Item) –
1.  If (Q->front == -1 or Q->rear == -1)   // Check for emptiness
2.          Q->front = Q->rear = 0
3.  Else If ( Q-> rear == SIZE-1)          // Check for fullness
4.          If(Q->front == 0)   // Queue is full
5.                  Print "Queue Overflow"
6.                  Exit
7.          Temp = Q->front  // Temp is an integer variable pointing to front
8.          Set i=0
9.          While(Temp<= Q->rear)
10.                 Q->array[i] = Q->array[Temp]
11.                 Temp = Temp + 1
12.                 i=i+1
13.         Q->rear = Q->rear – Q->front + 1 // Update rear
14.         Q->front = 0
15. Else
16.         Q->rear = Q->rear + 1
17. Q->array[Q->rear] = Item
```

Complexity = O(n)

Dequeue

This algorithm deletes an element from queue Q. Item is the element which is returned after deletion.

```
Algorithm Dequeue(struct queue *Q)
1.  If (Q->front == -1 or Q->rear == -1)   //Check for emptiness
2.          Display "Queue is empty"
3.          Exit
4.  Item  = Q->array[Q->front]
5.  If ( Q-> rear == Q->front)              // Only one element
6.          Q->front = -1
7.          Q->rear = -1
8.  Else
9.          Q->front = Q->front + 1
10. Return Item
```

Complexity = O(1)

Limitation of Linear Queue

If last position of queue is occupied then it is not possible to enqueue any more element even though some positions are vacant towards the front end of queue.

Solutions:

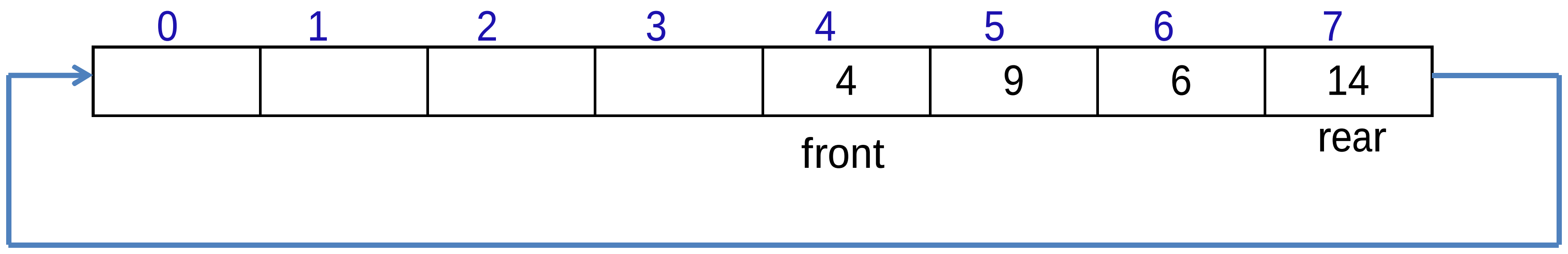
1. Shift the elements towards beginning (toward left) of queue and adjust the front and rear accordingly. But, if linear queue is very long then it would be very time consuming to shift the elements towards vacant positions. Hence Enqueue operation takes O(n) time.
2. **Make circular queue**

Circular Queue

During enqueue of an element in queue,

if rear reaches to (SIZE – 1) considering queue starting index is 0 then,

Make rear = 0



Creation of circular queue is same as linear queue.

Enqueue operation in Circular Queue

This algorithm inserts an element into queue Circular Queue. Item is the element which is to be inserted.

```
Algorithm EnCqueue( struct queue *CQ, int Item) –
1.  If (CQ->front == -1 or CQ->rear == -1)            //Check for emptiness
2.          CQ->front = CQ->rear = 0
3.  Else If ( CQ-> rear == SIZE-1)                     //1st condition to check for fullness
4.          If(CQ->front == 0)
5.                  Print “Queue Overflow”
6.                  Exit
7.          CQ->rear = 0        // Start inserting from beginning
8.  Else If(CQ->rear = = (CQ->front – 1))    //2nd condition to Check fullness
9.          Print “Queue Overflow”
10.         Exit
11. Else
12.         CQ->rear = CQ->rear + 1
13. CQ->array[CQ->rear] = Item
```

Complexity = O(1)

DeCqueue

This algorithm deletes an element from queue CQ. Item is the element which is returned after deletion.

```
Algorithm DeCqueue(struct queue *CQ) –
1.  If (CQ->front == -1 or CQ->rear == -1)          //Check for emptiness
2.          Display "Queue is empty"
3.          Exit
4.  Item  = CQ->array[CQ->front]
5.  If ( CQ-> rear == CQ->front)                    // Only one element
6.          CQ->front = -1
7.          CQ->rear = -1
8.  Else if (CQ->front == SIZE -1)         // if front points to last element
9.          CQ->front = 0
10. Else
11.         CQ->front = CQ->front + 1
12. Return Item
```

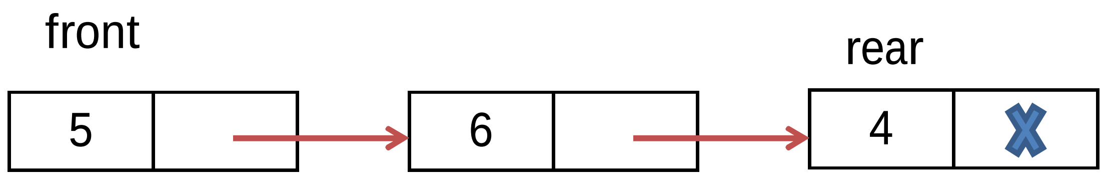
Complexity = O(1)

Limitation of array Implementation

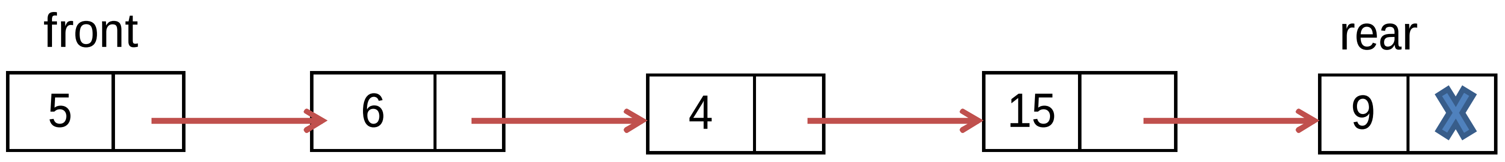
- Size of queue must be known in advance. In real scenario, it is not possible.

Linked List as Linear Queue

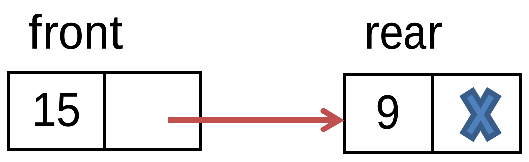
Representation



Insert (Enqueue) 15, 9



Delete (Dequeue) 3 times



Create queue

C code:

```
struct Node{
        int INFO;
        struct Node *NEXT;
};
struct Queue{
        struct Node *front;
         struct Node *rear;
}Q;
```

Create queue:

Algorithm Create_EmptyQueue(struct Queue ***Q):** This algorithm creates a queue with rear = NULL and front = NULL. Where front and rear are the pointers, pointing to first and last item of queue.

```
1. Q -> front = Q -> rear = NULL        // NULL means queue is empty
```

Enqueue

This algorithm inserts an element into queue Q. Item is the element which is to be inserted.

```
Algorithm EnLqueue( struct Queue * Q, int Item) –
1.  Node * New_node = Allocate memory
2.  If(New_node ==NULL) // Only when RAM is full
3.          Print "Queue overflow"
4.          Exit
5.  New_node >INFO = Item
6.  New_node >NEXT = NULL
7.  If (Q->front == NULL or Q->rear == NULL)        //Check for emptiness
8.          Q->front = Q->rear = New_node
9.  Else
10.         (Q->rear)->NEXT = New_node
11.         Q->rear = New_node
```

Complexity = O(1)

Dequeue

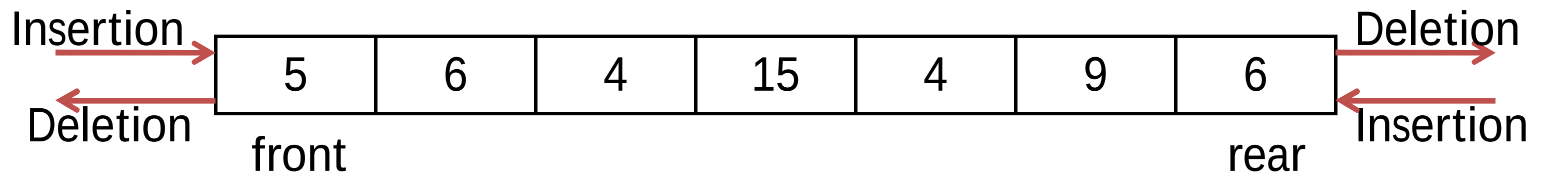
This algorithm deletes an element from queue Q. Item is the element which is returned after deletion.

```
Algorithm DeLqueue(struct Queue * Q) –
1.  Node *Temp
2.  If (Q->front == NULL or Q->rear == NULL)      // Check for emptiness
3.          Display “Queue is empty”
4.          Exit
5.  Item  = (Q-> front) ->INFO
6.  Temp = Q->front
7.  If ( Q-> rear == Q->front)                     // Only one element
8.          Q->front = NULL
9.          Q->rear = NULL
10. Else
11.         Q->front = (Q->front)-> NEXT
12. Free Temp
13. Return Item
```

Complexity = O(1)

Deque (Double-Ended Queue)

- Linear queue which allows insertion and deletion from both the ends of list
- Addition or deletion can be performed from front or rear



There are two variations of deque –

1. *Input restricted deque* – Insertions from only one end but deletion from both ends
2. *Output restricted deque* – Insertion from both ends and deletion from one end only

Operations on deque

1. Insertion at beginning – Using front pointer
2. Insertion at end – Using rear pointer
3. Deletion at beginning – Using front pointer
4. Deletion at end – Using rear pointer

Problem

Following is C like pseudo code of a function that takes a Queue as an argument, and uses a stack S to do processing.

```
void fun(Queue *Q)
{
   Stack S;  // Say it creates an empty stack S
   while (!isEmpty(Q))   // Run while Q is not empty
   {
      // deQueue an item from Q and push the dequeued item to S
      push(&S, deQueue(Q));
   }
   while (!isEmpty(&S)) // Run while Stack S is not empty
   {
     // Pop an item from S and enqueue the popped item to Q
     enQueue(Q, pop(&S));
   }
}
```

What does the above function do in general?

(A) Removes the last from Q

(B) Keeps the Q same as it was before the call

(C) Makes Q empty

(D) Reverses the Q

Priority Queue

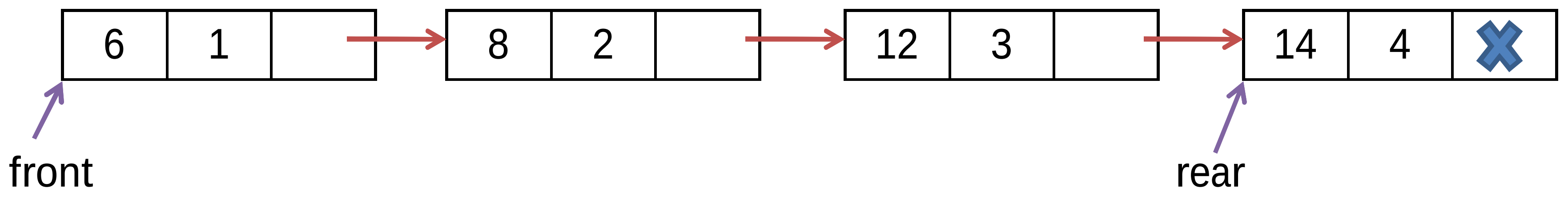
- ➢ A queue where each element is assigned a priority.
- ➢ Lower the number higher the priority.
- ➢ An element with highest priority is processed first before any element of lower priority.
- ➢ If two elements are of same priority then they are processed according to the order in which they were added in queue.

Priority decision parameter –

1. Shortest job
2. Payment
3. Time
4. Type of Job etc.

Implementation of priority queue

- Linear Linked List



Create priority queue

C code:

```
struct Node{
        int INFO;
        int priority;
        struct Node *NEXT;
};
struct Queue{
        struct Node *front;
         struct Node *rear;
}Q;
```

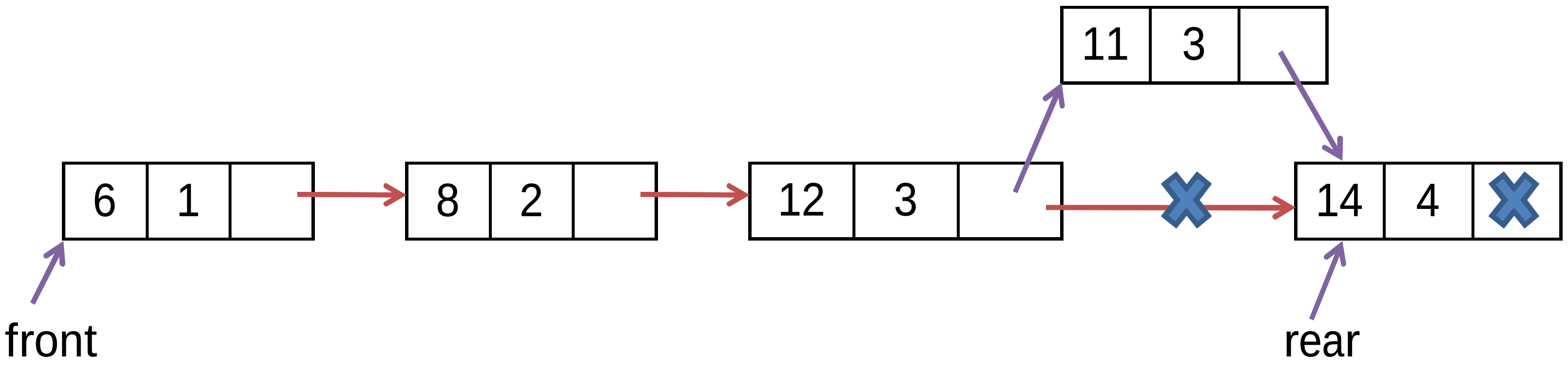
Create queue:

Algorithm Create_PQueue(Q): This algorithm creates a queue with rear = NULL and front = NULL. Here, Q is a linked list. front and rear the pointers, pointing to first and last item of queue.

```
1. Q -> front = Q -> rear = NULL          // NULL means queue is empty
```

Insertion in priority queue

1. Make node with provided item and assigning priority
2. Insert at proper place according to priority



Applications of priority queue

- Very useful in scheduling –

  - Traffic light

  - Job scheduling in operating system

- Bandwidth management –

  When higher priority customer needs more bandwidth, all other lower priority customer are blocked for some duration. Here, priority may be according to plan which customer use.

- Huffman coding –

  To transmit the data efficiently on network.

problem

Following is C like pseudo code of a function that takes a number as an argument, and uses a stack S to do processing.

What does the above function do in general?

```
void fun(int n)
{
    Stack S;  // Say it creates an empty stack S
    while (n > 0)
    {
      // This line pushes the value of n%2 to stack S
      push(&S, n%2);

      n = n/2;
    }

    // Run while Stack S is not empty
    while (!isEmpty(&S))
      printf("%d ", pop(&S)); // pop an element from S and print it
}
```

What does the above function do in general?

(A) Prints binary representation of n in reverse order

(B) Prints binary representation of n

(C) Prints the value of Logn

(D) Prints the value of Logn in reverse order

problem

- Which of the following operations is performed more efficiently by doubly linked list than by linear linked list?

A	Deleting a node whose location is given
B	Searching an unsorted list for a given item
C	Inserting a node after the node with a given location
D	Traversing the list to process each node