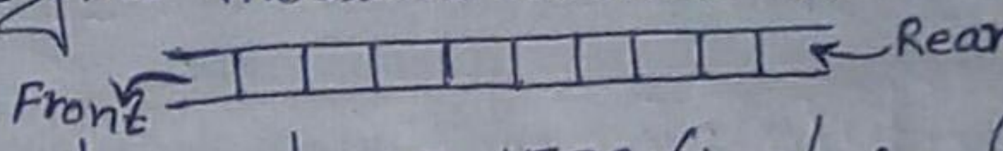
Unit-3
Queue

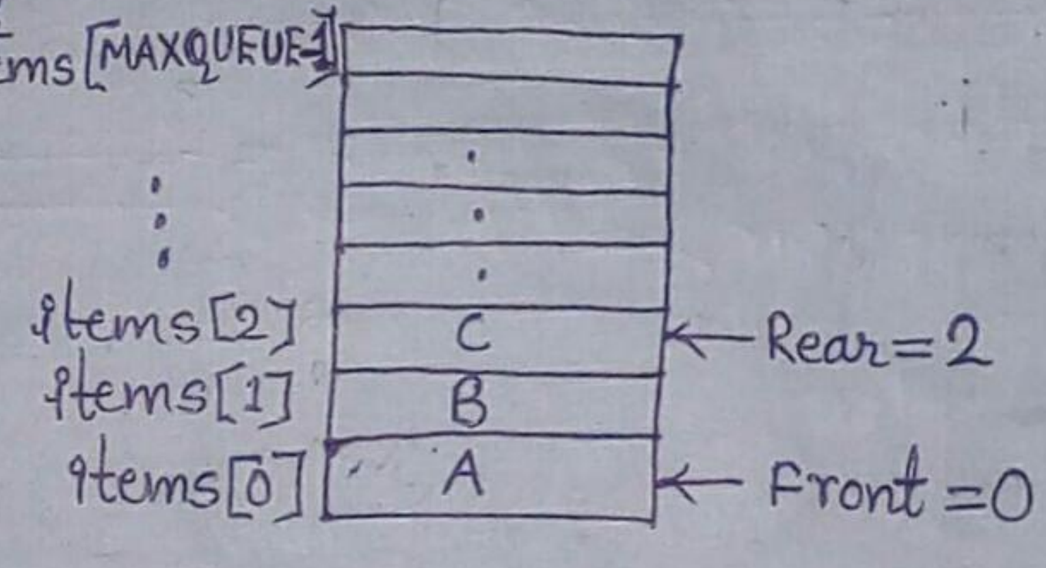
⊛. Definition and concept of queue:

A queue is an ordered collection of items from which items may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end (the rear of the queue).



The stack works as LIFO (last-in-first-out) technique but the queue works as FIFO technique (first-in-first-out). i.e, the first element inserted into the queue is the first element to be removed.

Example:



Applications of queue:

→ Task waiting for the printing
→ Time sharing system for use of CPU
→ For access to disk storage.
→ Task scheduling in operating system.

Initialization of queue:

→ The que is initialized by having the rear set to −1, and front set to 0. We can assume the maximum number of the element in queue as MAXQUEUE and the maximum number of item containing topmost address as [MAXQUEUE−1]. We subtract 1 from MAXQUEUE since we refer queue address starting from 0.

⊛ Operations on queue:

i) MakeEmpty(q): To make q as an empty queue.

ii) Enqueue(q,x): To insert an item x at the rear of the queue.

iii) Dequeue (q): To delete an item from the front of the queue q.

iv) IsFull(q): To check whether the queue q is full.

v) IsEmpty(q): To check whether the queue q is empty.

vi) Traverse (q): To read entire queue that is to display the content of the queue.

Some examples to understand operations on queue:

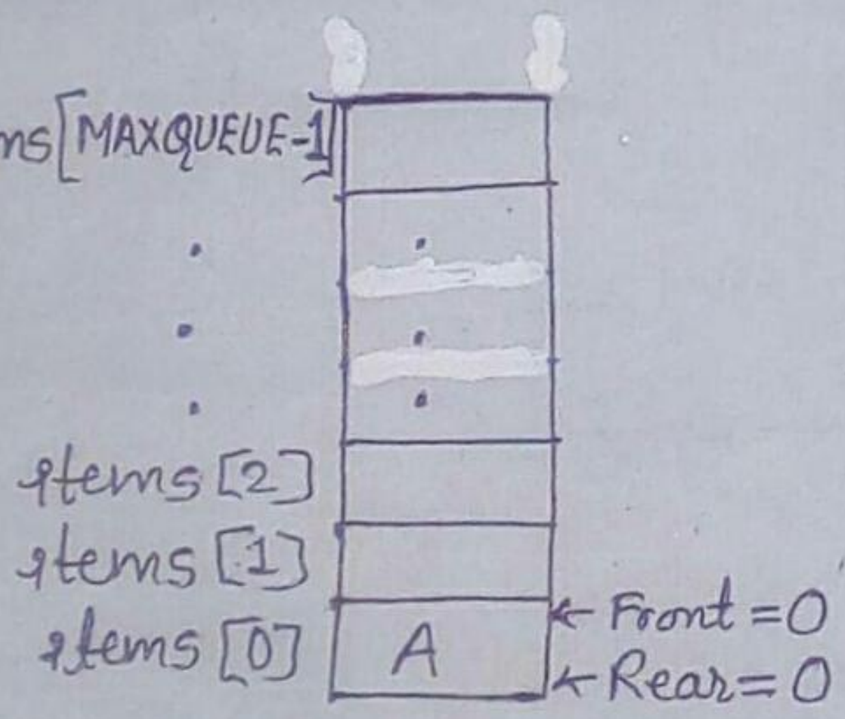


fig. Enqueue(A)

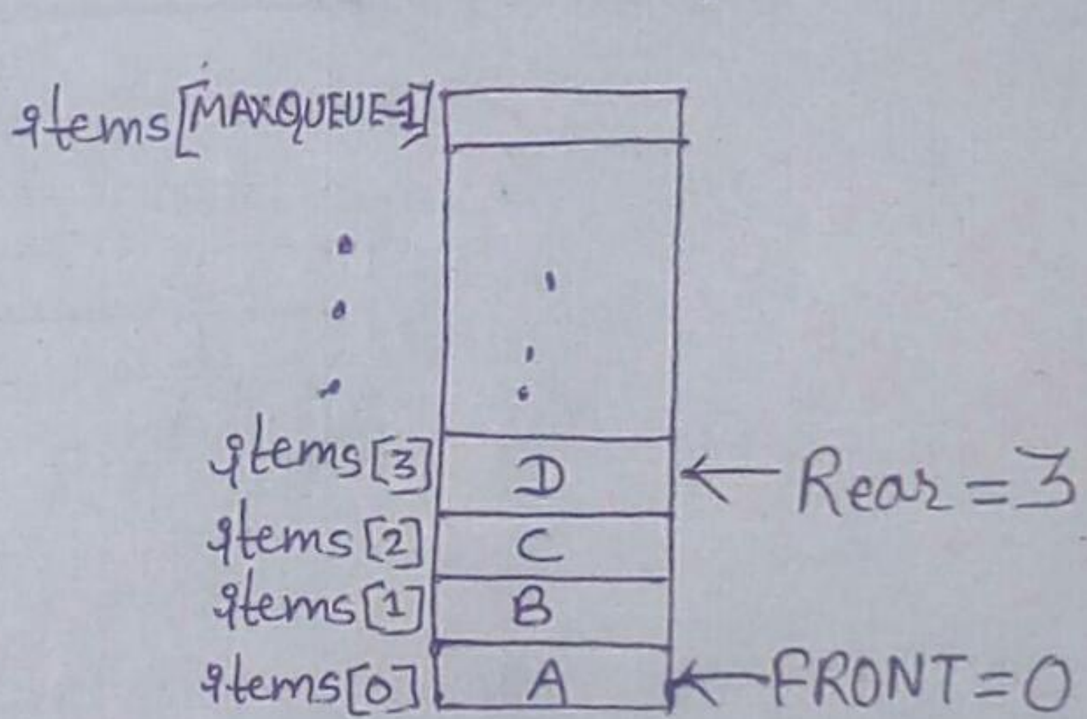


fig. Enqueue (B,C,D)

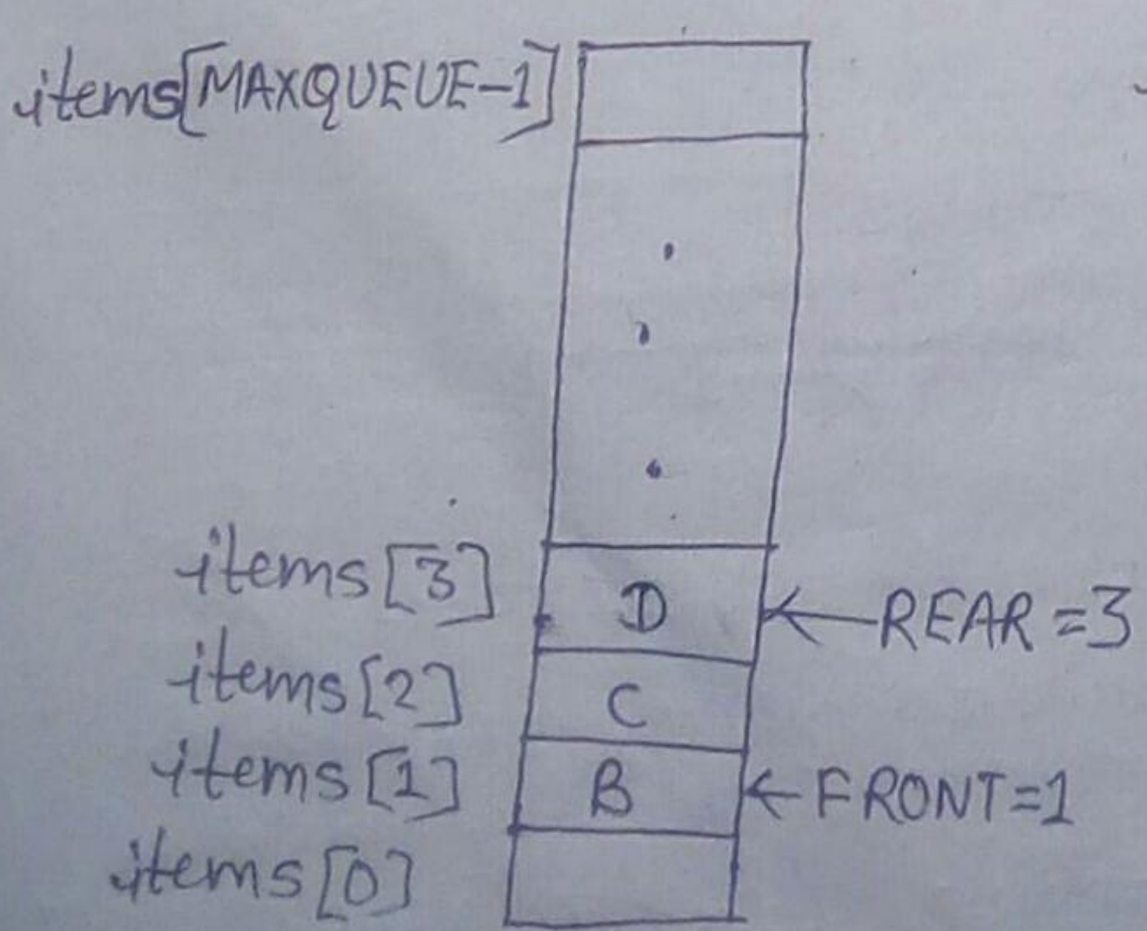


fig. Dequeue(A)

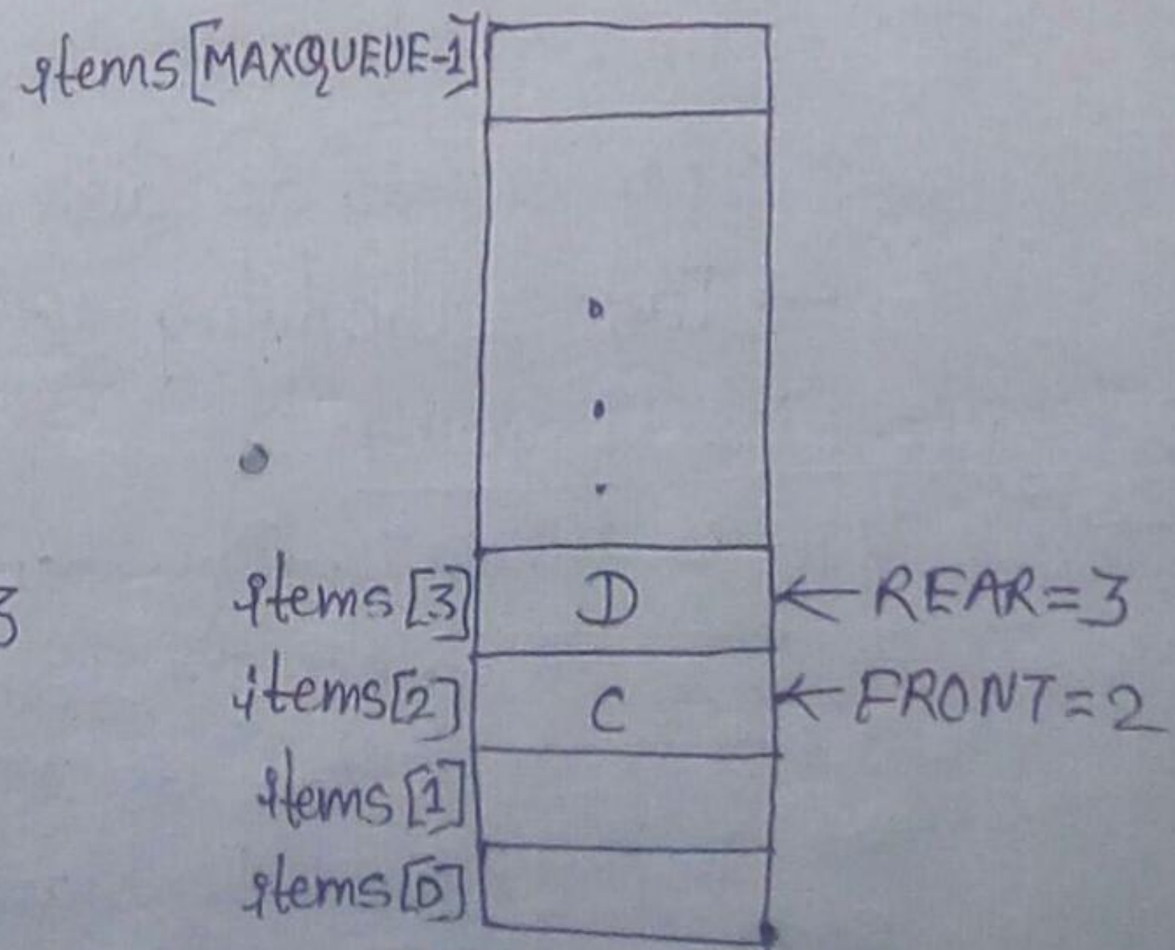


fig. Dequeue(B)

⊛. The Queue as a ADT:

A queue q of type T is a finite sequence of elements with the operations:

i) MakeEmpty(q): To make q as an empty queue.

ii) IsEmpty(q): To check whether the queue q is empty. Return true if q is empty; return false otherwise.

iii) IsFull(q): To check whether the queue q is full. Return true if q is full, return false otherwise.

iv) Enqueue(q,x): To insert an item x at the rear of the queue, if and only if q is not full.

v) Dequeue(q): To delete an item from the front of queue q, if and only if q is not empty.

vi) Traverse(q): To read entire queue that is to display the content of queue.

⊛. Sequential representation of queue:

Sequential representation of queues is most commonly done using arrays. Most arrays have a fixed size, meaning that arrays cannot be increased in size by simply appending elements to the end of the array.

Arrays are commonly used to implement bounded queues. The enqueue operation for a bounded queue must ~~be~~ not be performed on a queue that is full, just as a dequeue operation must not be performed on an empty queue.

The problem faced in array of fixed size can be overcomed using linked lists to implement unbound queues which are limited in size only by the amount of available memory in the computer. The queue then never be full. Each type of representation has its uses. We choose the representation that best serves the need of our program.

⊛. Implementation of queue:

Implementation of queue

- Array implementation of queue. (static memory allocation)
  - Linear array implementation
  - Circular array implementation
- Linked list implementation of queue. (dynamic memory allocation)

⊛. Linear array implementation (Linear queue):-

Algorithm for insertion an item in queue:

1. Initialize front=0 and rear=-1.

```
if rear >= MAXSIZE-1
    print "queue overflow" and return.
else
    set rear = rear+1
    queue[rear] = item
```

2. end.

Algorithm to delete an element from the queue:

1. if rear < front
   print "queue is empty" and return.
   else,
   item = queue [front++]
2. end.

Declaration of a Queue:

```
#define MAXQUEUE 50    /* size of the queue items */
struct queue {
    int front;
    int rear;
    int items[MAXQUEUE];
};
typedef struct queue qt;
```

⊛. Defining the operations of linear queue: (less imp)

i) The MakeEmpty function:

```
void MakeEmpty (qt *q).
{
   q -> rear = -1;
   q -> front = 0;
}
```

ii) The IsEmpty function:

```
int IsEmpty (qt *q)
{ if (q -> rear < q -> front)
      return 1;
  else return 0;
}
```

iii) The Isfull function:

```
int IsFull (qt *q)
{ if (q -> rear == MAXQUEUESIZE-1)
    else return 1;
      return 0;
}
```

iv) The Enqueue function:

```
void Enqueue (qt *q, int newitem)
{ if(IsFull(q))
  { printf ("queue is full");
    exit (1);
  }
else{
  { q -> rear++;
    q -> items [q -> rear] = newitem;
  }
}
```

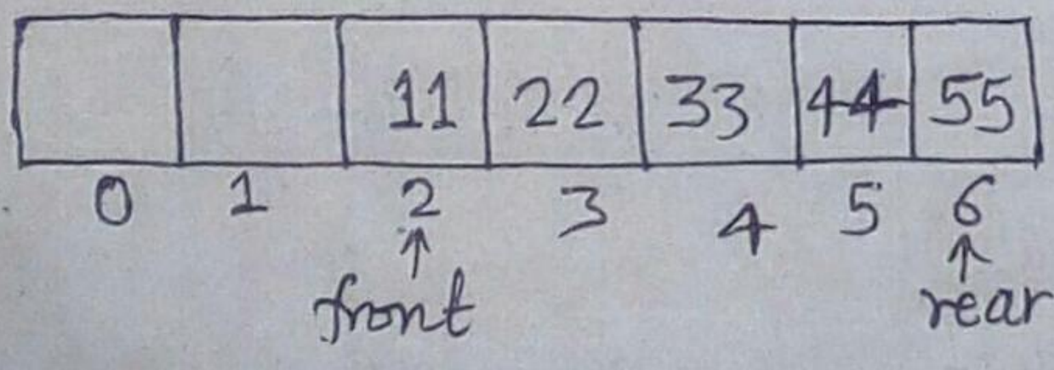
v) The Dequeue function:

```
int Dequeue (qt *q)
{
  if (IsEmpty(q))
    {
      printf ("queue is Empty");
      exit(1);
    }
else
  {
    return(q->items[q->front]);
    q->front++;
  }
}
```

⊛. Problems with Linear queue implementation:

i) Both rear and front indices are increased but never decreased.

ii) As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again. Wastage of the space is the main problem with linear queue which is illustrated by the following example:

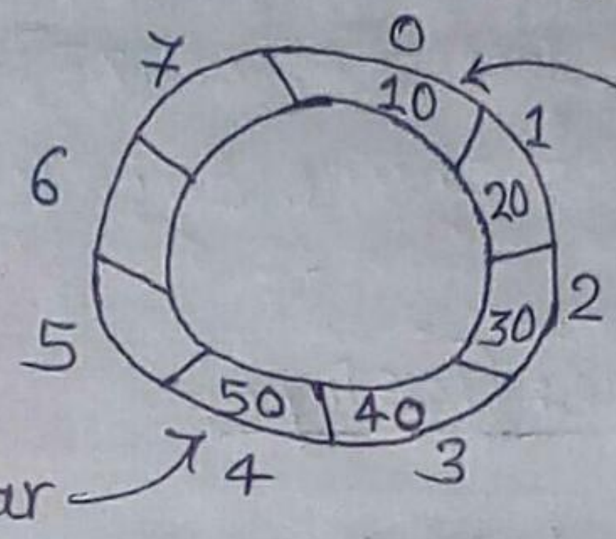


front = 2, rear = 6.

This queue is considered full, even though the space at beginning is vacant.

⊛. Circular queue (Circular array implementation): (V.V.I)

Circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.



→ A circular queue overcomes the problem of unutilized space in linear queue implementation as array.
→ We can insert one less element than the size of the array in circular queue.

Initialization of circular queue:

rear = front = MAXSIZE-1

⊛. Algorithm for inserting an element in a circular queue:

Assume that rear and front are initially set to MAXSIZE-1.

1. if (front == (rear+1) % MAXSIZE)
   print Queue is full and exit.
   else
   rear = (rear+1) % MAXSIZE;
2. cqueue[rear] = item;
3. end.

⊛. Algorithm for deleting an element from a circular queue:

Assume that rear and front are initially set to MAXSIZE-1.

1. if (rear == front) [Checking empty condition]
   print Queue is empty and exit.
2. front = (front+1) % MAXSIZE;
3. item = cqueue[front];
4. return item;
5. end.

⊛. Declaration of a circular queue:

```
#define MAXSIZE 50 /*size of the circular queue items*/
struct cqueue {
                int front;
                int rear;
                int items[MAXSIZE];
              };
typedef struct cqueue cq;
```

⊛. Operations of a circular queue: (less imp)

i) The MakeEmpty function:

```
void MakeEmpty(cq *q)
{
  q->rear = MAXSIZE-1;
  q->front = MAXSIZE-1;
}
```

ii) The IsEmpty function:

```
int IsEmpty(cq *q)
{ if(q->rear < q->front)
    return 1;
  else
    return 0;
}
```

iii) The IsFull function:

```
int IsFull(cq *q)
{ if(q->front == (q->rear+1)% MAXSIZE)
    return 1;
  else
    return 0;
}
```

iv) The Enque function:

```
void Enqueue(cq *q, int newitem)
{ if(IsFull(q))
  { printf("queue is full").
    exit(1);
  }
  else
  { q->rear = (q->rear+1)% MAXSIZE.
    q->items[q->rear] = newitem;
  }
}
```

v) The Dequeue function:

```
int Dequeue (cq *q)
{ if (IsEmpty(q))
    { printf ("queue is Empty").
      exit(1);
    }
  else {
    { q->front=(q->front+1)%MAXSIZE;
      return (q->items[q->front]);
    }
}
```

⊛ Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed. comes from following rules:

i) An element of higher priority is processed before any element of lower priority.

ii) If two elements has same priority then they are processed according to the order in which they were added to the queue.

The best application of priority queue is observed in CPU scheduling.

→ The jobs which have higher priority are processed first.

→ If the priority of two jobs is same these jobs are processed according to their position in queue.

→ A shorter job is given higher priority over the longer one.

There are two types of priority queue an ascending priority queue in which only the smallest item can be removed and other is descending priority queue in which only the largest item can be removed.

⊛ Priority QUEUE Operations:

i) Insertion → The insertion in Priority queues is the same as in non-priority queues.

Declaration → The declaration in priority queues is also the same as in non-priority queues.

iii) Deletion → Deletion requires a search for the element of highest priority and deletes the element with highest priority. The following methods can be used for deletion from a given priority queue:

→ An empty indicator replaces deleted elements.
→ After each deletion elements can be moved up in the array decrementing the rear.
→ The array in the queue can be maintained as an ordered circular array.

⊛. The priority queue ADT:

An ascending priority queue of elements of type T is a finite sequence of elements of T together with the operations:

i) Make Empty (p): Create an empty priority queue p.
ii) Empty (p): Determine if the priority queue p is empty or not.
iii) Insert (p, x): Add element x on the priority queue p.
iv) DeleteMin (p): If the priority queue p is not empty, remove the minimum (smallest) element of the queue and return it.
v) FindMin (p): Retrive the minimum element of the priority queue p.

⊛. Array implementation of priority queue:

i) Unordered array implementation:-

→ To insert an item, insert it at the rear end of queue.
→ To delete an item, find the position of minimum element and
  → either mark it as deleted
  → OR shift all elements past the deleted element by on position and then decrement rear.

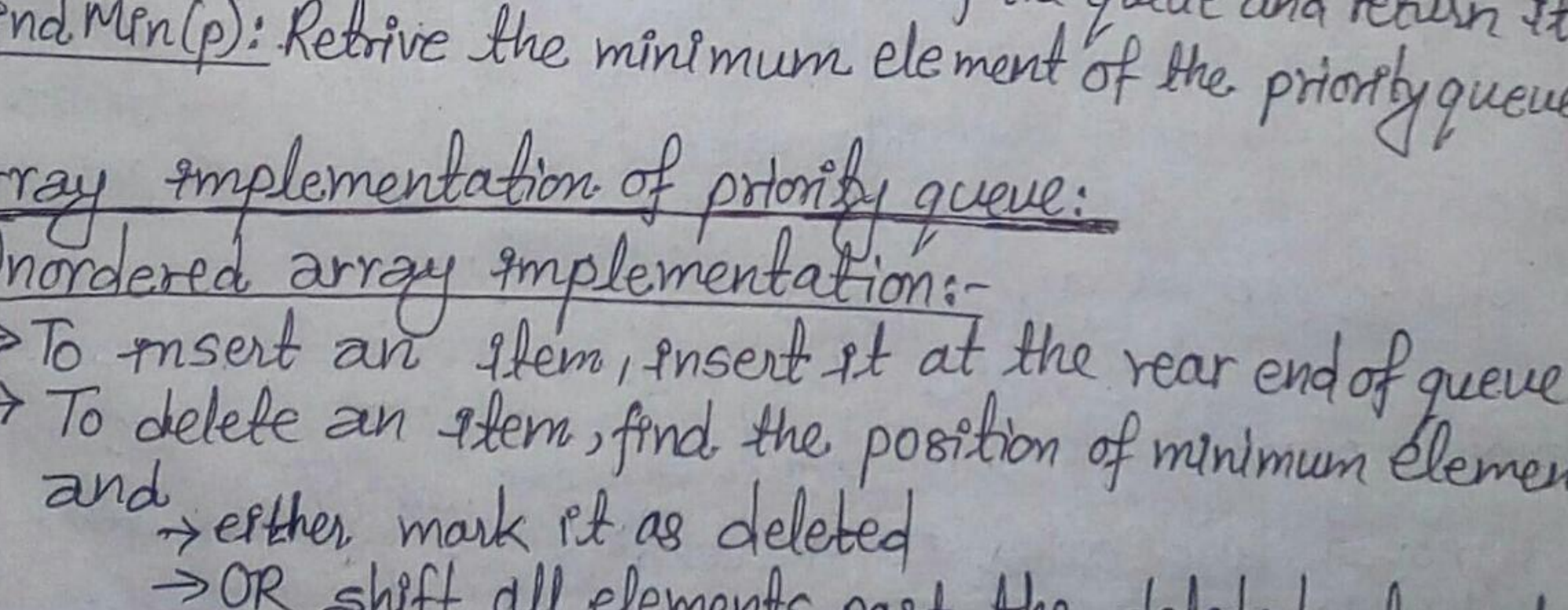


fig. Illustration of unordered array implementation.

Ordered array implementation →

→ Set the front as the position of the smallest element and the rear as the position of the largest element.
→ To insert an element, locate the proper position of the new element and shift preceding elements by one position.
→ To delete the minimum element, increament the front position.

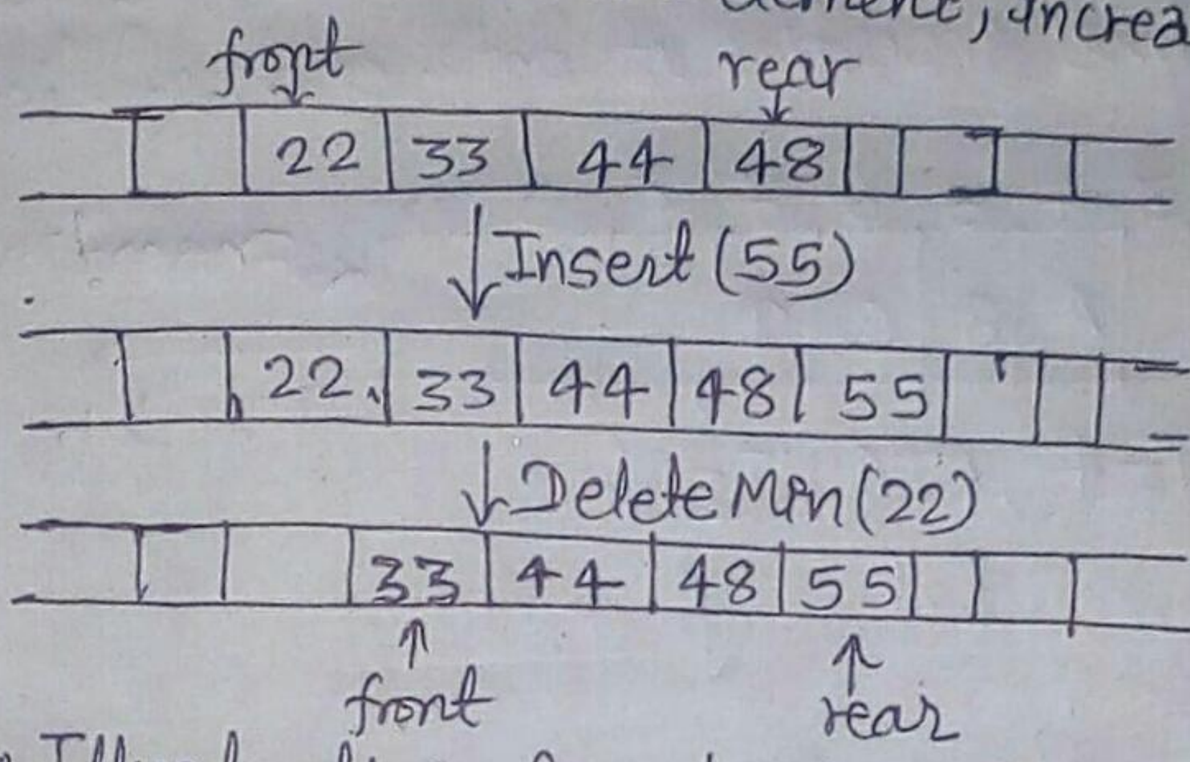


fig:- Illustration of ordered array implementation.

⊛. Application of Priority queue:

In a time-sharing computer system, a large number of tasks may be waiting for the CPU, some of these tasks have higher priority than others. The set of tasks waiting for the CPU forms a priority queue.

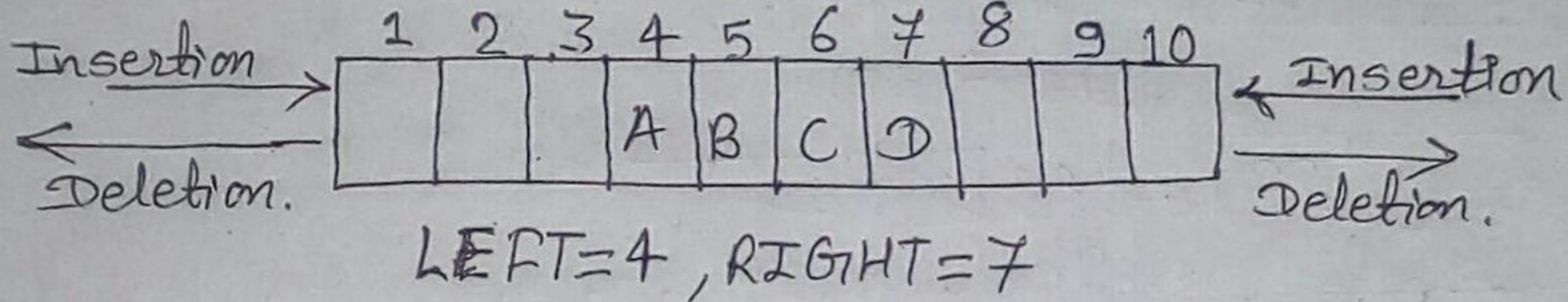
⊛. Dequeue: (Imp):

A dequeue is a linear list in which elements can be added or removed at either end but not the middle. The term Dequeue is construction of the name Double-Ended-Queue.

Dequeue is maintained by a circular array (DEQUEUE) with the pointer LEFT and RIGHT, which points to the two ends of the queue. We assume that the elements extend from the left end to the right end in the array. The condition LEFT = NULL indicates that DEQUEUE is empty. There are following two variables of DEQUEUE.

i) Input restricted DEQUEUE → An input restricted DEQUEUE which allows insertion at only one end of the list but allows deletion at both ends of the list.

ii) Output restricted DEQUEUE → An output restricted DEQUEUE is a QUEUE which allows deletion at only one end of the list but allows insertion at both ends of list.



Memory representation of DEQUE

⊛. Differences between linear and circular queue [Imp]

Linear Queue.	Circular Queue.
i) It organizes the data elements and instructions in a sequential order one after another.	i) It arranges the data in circular pattern where the last element is connected to the first element.
ii) Tasks are executed in order they were placed before (FIFO).	ii) Order of executing a task may change.
iii) The new element is added from rear end and removed from front end.	iii) Insertion and deletion can be done at any position
iv) It is inefficient. than that of circular queue.	iv) It works better than the linear queue.