QUEUES AND ITS APPLICATIONS

Queues

- Definition: A queue is an ordered list in which insertions and deletions takes place at different ends.

- **New elements are added to rear end**
- **Old elements are deleted from front end**

- **Since first element inserted is the first element deleted queues is also known as First-In-First-Out (FIFO) lists.**

Queue: **a First-In-First-Out (FIFO) list**

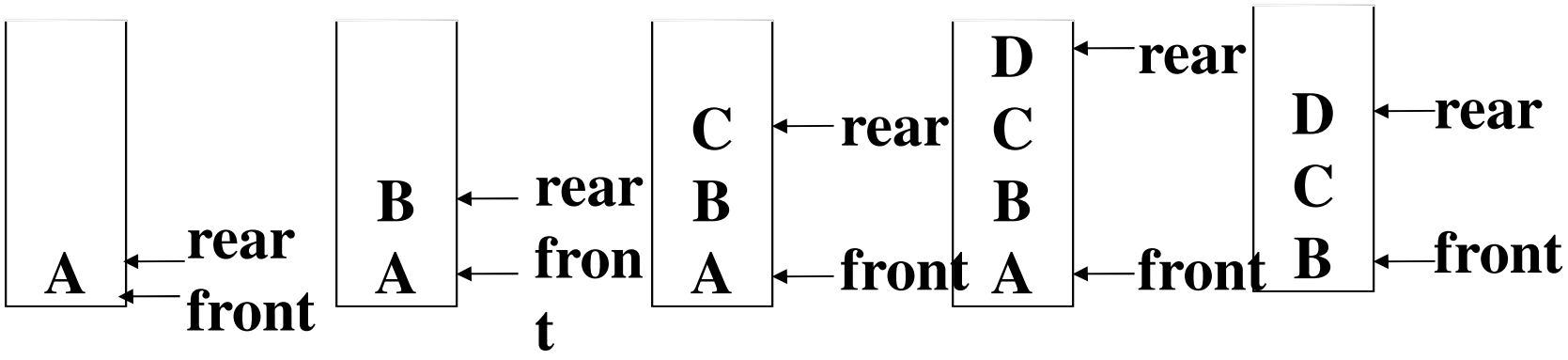


*Figure 3.4: **Inserting and deleting elements in a queue (p.106**

Queues - Application

- **Used by operating system (OS) to create job queues.**
- **If OS does not use priorities then the jobs are processed in the order they enter the system**

Application: **Job scheduling**

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

*Figure 3.5: **Insertion and deletion from a sequential queue (p.108)**

Abstract data type of queue

structure ***Queue*** **is**

objects: **a finite ordered list with zero or more elements.**

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*,

max_ queue_ size **∈ positive integer**

Queue **CreateQ(*max_queue_size*) ::=**

create an empty queue whose maximum size is

max_queue_size

Implementation 1: **using array**

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE 100/* Maximum queue size */
typedef struct {
        int key;
        /* other fields */
        } element;

element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
```

Abstract data type of queue

***Boolean* IsFullQ(*queue, max_queue_size*) ::=**
if**(number of elements in *queue* ==**
***max_queue_size*)**
return ***TRUE***
else return ***FALSE***

***Queue* AddQ(*queue, item*) ::=**
if **(IsFullQ(*queue*)) *queue_full***
else **insert *item* at rear of *queue* and return *queue***

Implementation 1: **using array**

Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

Add to a queue

```
void addq(int *rear, element item)
{
/* add an item to the queue */
   if (*rear == MAX_QUEUE_SIZE_1) {
      queue_full( );
      return;
   }
   queue [++*rear] = item;
}
```

*Program 3.3: **Add to a queue (p.108)**

***Boolean* IsEmptyQ(*queue*) ::=**
if **(*queue* ==CreateQ(*max_queue_size*))**
return ***TRUE***
else return ***FALSE***

***Element* DeleteQ(*queue*) ::=**
if **(IsEmptyQ(*queue*))** return
else **remove and return the *item* at front of queue.**

*Structure 3.2: **Abstract data type *Queue (p.107)***

Implementation 1: **using array**

Boolean IsEmpty(queue) ::= front == rear

Delete from a queue

```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
   if ( *front == rear)
      return queue_empty( );     /* return an error key */
   return queue [++ *front];
}
```

*Program 3.4: **Delete from a queue(p.108)**

problem:

there may be available space when IsFullQ is true

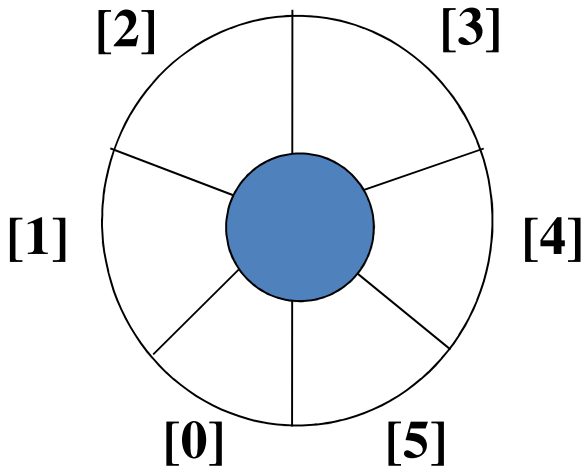
i.e.. movement is required.

Implementation 2: **regard an array as a circular queue**

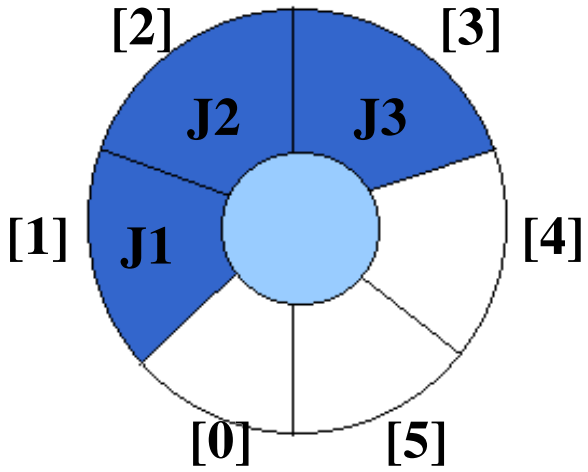
front: one position counterclockwise from the first element

rear: current end

EMPTY QUEUE



front = 0
rear = 0



front = 0
rear = 3

*Figure 3.6: **Empty and nonempty circular queues (p.109)**

Problem: **one space is left when queue is full**

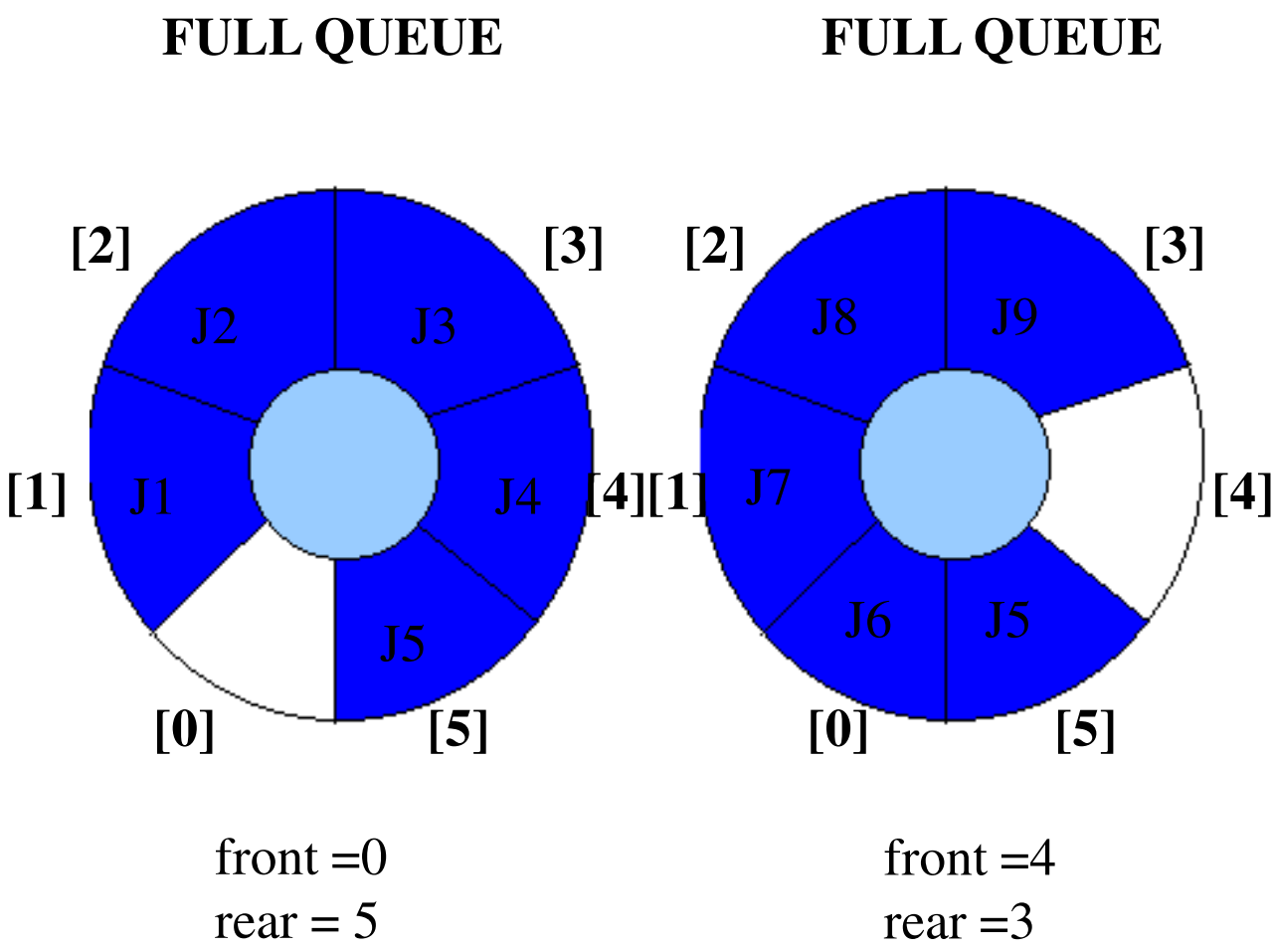


*Figure 3.7: **Full circular queues and then we remove the item (p.110)**

Add to a circular queue

```
void addq(int front, int *rear, element item)
{




}
```

*Program 3.5: **Add to a circular queue (p.110)**

Add to a circular queue

```
void addq(int front, int *rear, element item)
{
/* add an item to the queue */
   *rear = (*rear +1) % MAX_QUEUE_SIZE;
    if (front == *rear) /* reset rear and print error */
    return;
  }

    queue[*rear] = item;
}
```

*Program 3.5: **Add to a circular queue (p.110)**

Delete from a circular queue

```
element deleteq(int* front, int rear)
{




}
```

*Program 3.6: **Delete from a circular queue (p.111)**

Delete from a circular queue

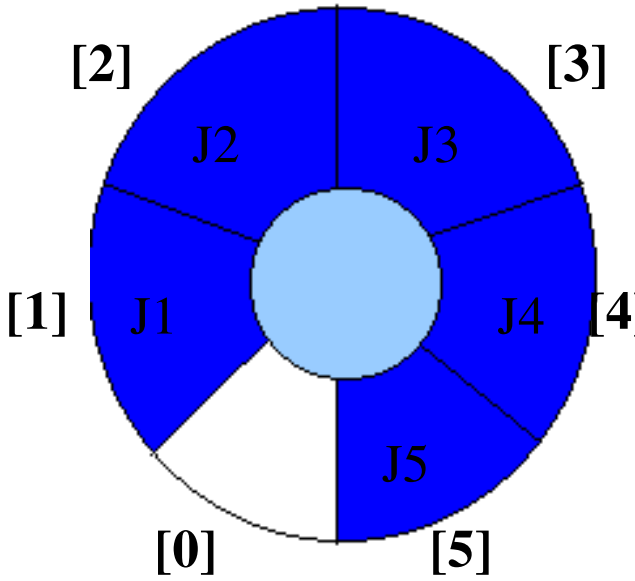
```
element deleteq(int* front, int rear)
{
  element item;
  /* remove front element from the queue and put it in item
*/
     if (*front == rear)
       return queue_empty( );
              /* queue_empty returns an error key */
    *front = (*front+1) % MAX_QUEUE_SIZE;

    return queue[*front];
}
```

*Program 3.6: **Delete from a circular queue (p.111)**

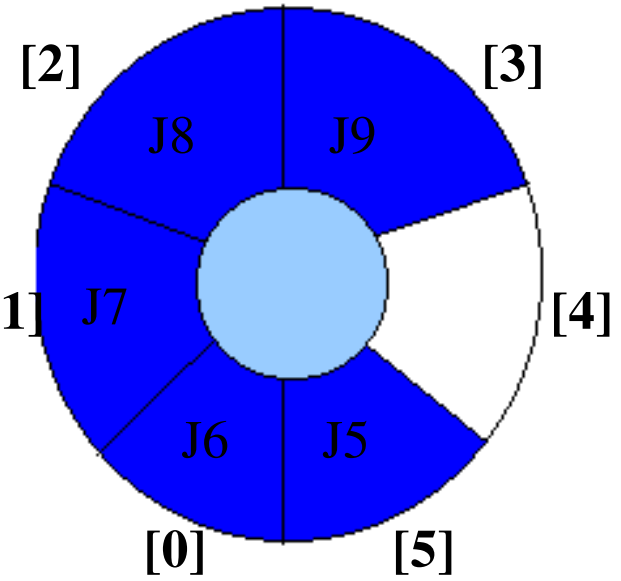
Circular Queue FULL: front = = rear condition

FULL QUEUE



front =0
rear = 5

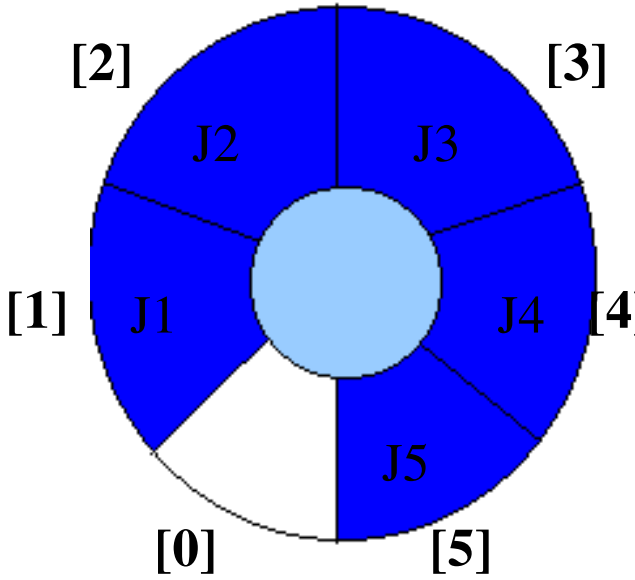
FULL QUEUE



front =4
rear =3

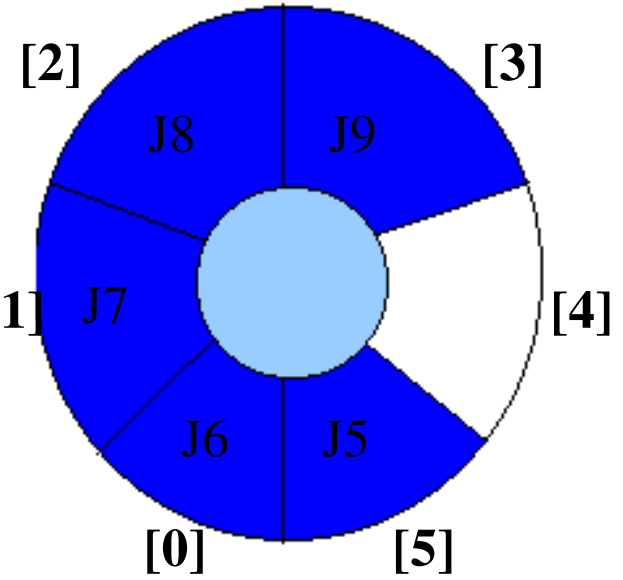
Circular Queue using Dynamically Allocated Arrays

FULL QUEUE

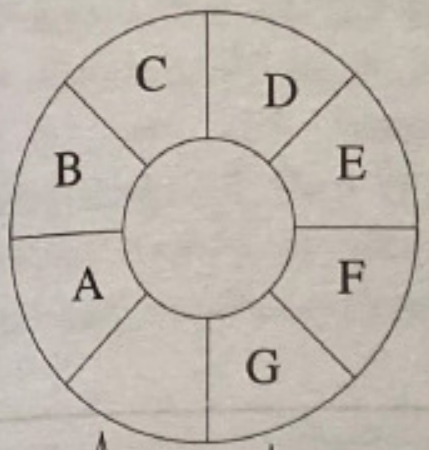


front =0
rear = 5

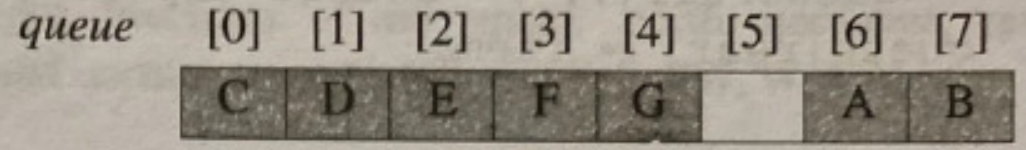
FULL QUEUE



front =4
rear =3

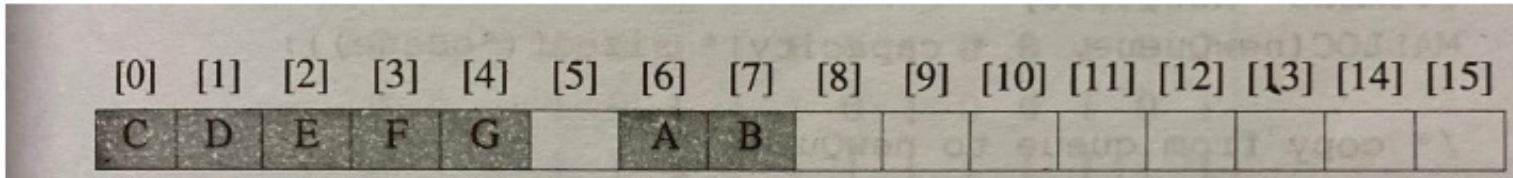


(a) A full circular queue



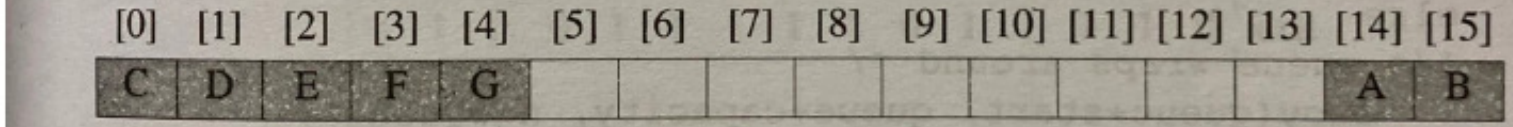
front = 5, *rear* = 4

(b) Flattened view of circular full queue



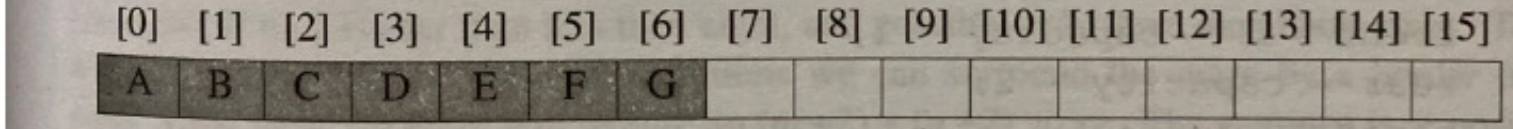
front = 5, *rear* = 4

(c) After array doubling



front = 13, *rear* = 4

(d) After shifting right segment



front = 15, *rear* = 6

(e) Alternative configuration

Figure 3.7: Doubling queue capacity

Circular Queue using Dynamically Allocated Arrays

```
void addq(element item)
{
……….

queueFull();

……..

}
```

Circular Queue using Dynamically Allocated Arrays

```
void queueFull()
{ /*allocate an array with twice the capacity*/

/* Copy from queue to newQueue*/

/* Switch to newQueue
}
```

Circular Queue using Dynamically Allocated Arrays

```
void queueFull()
{ /*allocate an array with twice the capacity*/
Element * newQueue;
newQueue = (newQueue*) malloc((2*capacity *
sizeof(*queue));
/* Copy from queue to newQueue*/

/* Switch to newQueue
}
```

Circular Queue using Dynamically Allocated Arrays

```
void queueFull()
{ /*allocate an array with twice the capacity*/
/* Copy from queue to newQueue*/
Start = (front+1) % capacity;
If (Start < 2) /* No wrap around */
{…}
Else /* wrap around */
{…}
/* Switch to newQueue
}
```

Circular Queue using Dynamically Allocated Arrays

```
 void queueFull()
{ /*allocate an array with twice the capacity*/
/* Copy from queue to newQueue*/
Start = (front+1) % capacity;
If (Start < 2) /* No wrap around */
copy(queue+start, queue+capacity-1, newQueue);
Else /* wrap around */
{copy(queue+start, queue+capacity, newQueue);
 copy(queue, queue+rear-1, newQueue);
}
/* Switch to newQueue
}
```

Circular Queue using Dynamically Allocated Arrays

```
void queueFull()
{ /*allocate an array with twice the capacity*/
/* Copy from queue to newQueue*/
/* Switch to newQueue
front = 2*capacity-1;
rear = capacity -2;
capacity *= 2;
free(queue);
queue = newQueue;
}
```

Priority Queues

The priority queue is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations.
Two types of priority queues:

- **an ascending priority queue**
- **a descending priority queue.**

An ascending priority queue - collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. If apq is an ascending priority queue. the operation pqinsert(apq,x) inserts element x into apq and pqmindelete(apq) removes the minimum element from apq and returns its value.

A descending priority queue - allows deletion of only the largest item. The operations applicable to a descending priority queue dpq, are pqinsert(dpq.x) and pqmaxdelete(dpq), pqinsert(dpq.x) inserts element x into dpq and is logically identical to pqinsert for an ascending priority queue. pqmaxdelete(dpq) removes the maximum element from dpq and returns its value.

The operation empty(pq) applies to both types of priority queue and determines whether a priority queue is empty.

Elements of a priority queue –

- **Need not be numbers or characters that can be compared directly.**
- **May be complex structures that are ordered on one or several fields.**
- **The *field on* which the elements of a priority queue is ordered need not be part of the elements themselves - it may be a special external value used specifically for the purpose of ordering the priority queue.**

- **Example:**
  - **Stack may be viewed as a descending priority queue whose elements are ordered by time of insertion.**
  - **Queue may be viewed as an ascending priority queue whose elements are ordered by time of insertion.**

Array Implementation of a Priority Queue

Suppose that the n elements of a priority queue pq are maintained in positions 0 to n-1 of an an-array *pq.items* of size maxpq and suppose that *pq.rear* equals to first empty array position, n.

pqinsert(pq, x)

```
if  (pq. rear ›= maxpq){
  prints ("priority queue overflow");
  exit(1);
) /* end if */
pq.items[pq.rear] = x;
pg.rear++;
```

Note: Under this insertion method the elements of the priority queue are not kept ordered in the array.

pqmindelete(pq) on an ascending priority queue

1. Locate the smallest element - every element of the array from *pq.items[0]* through *pir.items[pq.rear-1] must* be examined. Requires accessing every element of the priority queue.
2. Deleting an element in the middle of the array.

   Solutions:

   i. Special "empty" indicator – an invalid value could be placed

   ii. Special "empty" indicator is used. New item is inserted in the first empty position

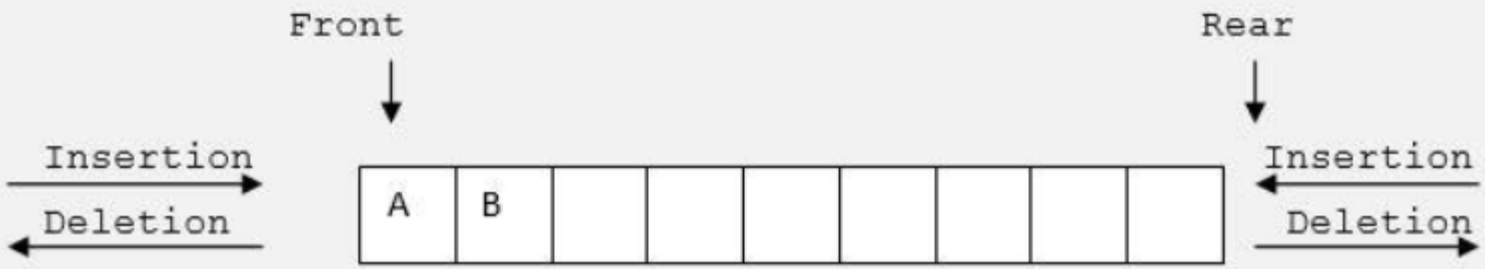
   iii. Each deletion, compact the array

   iv. Maintain the pq as an ordered circular array.

Double Ended Queue (dequeuer)

Description:

A queue that supports insertion and deletion at both the front and rear is called **double-ended queue or Dequeue.** A Dequeue is a linear list in which elements can be added or removed at either end but not in the middle.



The operations that can be performed on Dequeue are

1. Insert to the beginning
2. Insert at the end
3. Delete from the beginning
4. Delete from end

```
#define MAX 30
typedef struct{
    int data[MAX];
    int front, rear;
}dequeue;
Void initialise (deque *P){
    P->front = -1;
    P->rear = -1;
}
int empty(deque *P){
    if(P->rear == -1)
        return 1;
    return 0;
}
int full (deque *P){
    if((P->rear +1)%MAX == P->front)
        return 1;
    return 0;
}
```

```
void enqueueR(dequeue *P, int x){
    if (full(P)){
        // queue file
    }
    if (empty(P)){
        P->rear=0;
        P->front=0;
        P->data[0]=x;
    }
    else{
        P->rear = (P->rear+1)% MAX.
        P->data [P->rear] =x;
    }
}
```

```
void enqueF(deqU* P, int x){
    .
    .
    .
    .
    .
    .
    else{
        P->front=(P->front-1+MAX)% MAX;
        P-> data[P->front]=x;
    }
}
```

```
int dequeF(deque* P){
    // check for empty queue
    int x;
    x = P->data[P->front];
    if(P->rear == P->front)
        initialise(P);
    else P->front = (P->fr +1)% MAX;
    return x;
}
```

```
int dequeR(dequ *P){

    x=P->data[P->rear];
    //same as dequeF
  else
     P->rear=(P->rear-1+MAX)%MAX;
  return x;
}
```

Variations of deque

1) Input restricted deque – insertion at 1 end only

2) Output " " – deletion " "

CHAPTER 3