

# BLM267

## Chapter 8: Queues

**Data Structures Using C, Second Edition**  
Reema Thareja

- ◆ **Introduction to Queues**
- ◆ **Array Representation of Queues**
- ◆ **Linked Representation of Queues**
- ◆ **Types of Queues**
  - ◆ **Circular Queues**
  - ◆ **Dequeues**
  - ◆ **Priority Queues**
  - ◆ **Multiple Queues**
- ◆ **Applications of Queues**

# Introduction to Queues

- ❖ “People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it. ”
- ❖ We see that the element at the first position is served first.
- ❖ Same is the case with queue data structure.
- ❖ A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- ❖ The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT.
- ❖ Queues can be implemented by using either arrays or linked lists.

# Array Representation Of Queues

- ❖ **Operations on Queues**
- ❖ **In Fig. 8.1, front = 0 and rear = 5.**
- ❖ **Suppose we want to add another element with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.**
- ❖ **The queue after addition would be as shown in Fig. 8.2. Here, front = 0 and rear = 6.**
- ❖ **Every time a new element has to be added, we repeat the same procedure.**

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

**Figure 8.1** Queue

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

**Figure 8.2** Queue after insertion of a new element

	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

**Figure 8.3** Queue after deletion of an element

# Array Representation Of Queues

- ❖ If we want to delete an element from the queue, then the value of front will be incremented.
- ❖ Deletions are done from only this end of the queue.
- ❖ The queue after deletion will be as shown in Fig. 8.3.
- ❖ Here, front = 1 and rear = 6

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Figure 8.1 Queue

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Figure 8.2 Queue after insertion of a new element

	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Figure 8.3 Queue after deletion of an element

## Array Representation Of Queues

- ❖ However, before inserting an element in a queue, we must check for overflow conditions.
- ❖ An overflow will occur when we try to insert an element into a queue that is already full.
- ❖ When  $\text{rear} = \text{MaX} - 1$ , where MaX is the size of the queue, we have an overflow condition.
- ❖ Note that we have written  $\text{MaX} - 1$  because the index starts from 0.
- ❖ Similarly, before deleting an element from a queue, we must check for underflow conditions.
- ❖ An underflow condition occurs when we try to delete an element from a queue that is already empty.
- ❖ If  $\text{front} = -1$  and  $\text{rear} = -1$ , it means there is no element in the queue.
- ❖ Let us now look at Figs 8.4 and 8.5 which show the algorithms to insert and delete an element from a queue.

# Array Representation Of Queues

```
Step 1: IF REAR = MAX-1
        Write OVERFLOW
        Goto step 4
    [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
        SET FRONT = REAR = 0
    ELSE
        SET REAR = REAR + 1
    [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

**Figure 8.4** Algorithm to insert an element in a queue

```
Step 1: IF FRONT = -1 OR FRONT > REAR
        Write UNDERFLOW
    ELSE
        SET VAL = QUEUE[FRONT]
        SET FRONT = FRONT + 1
    [END OF IF]
Step 2: EXIT
```

**Figure 8.5** Algorithm to delete an element from a queue

# Array Representation Of Queues

```
void insert()
{
    int num;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d", &num);
    if(rear == MAX-1)
        printf("\n OVERFLOW");
    else if(front == -1 && rear == -1)
        front = rear = 0;
    else
        rear++;
    queue[rear] = num;
}

int delete_element()
{
    int val;
    if(front == -1 || front > rear)
    {
        printf("\n UNDERFLOW");
        return -1;
    }
    else
    {
        val = queue[front];
        front++;
        if(front > rear)
            front = rear = -1;
        return val;
    }
}
```



# Array Representation Of Queues

```
int peek()
{
    if(front==-1 || front>rear)
    {
        printf("\n QUEUE IS EMPTY");
        return -1;
    }
    else
    {
        return queue[front];
    }
}

void display()
{
    int i;
    printf("\n");
    if(front == -1 || front > rear)
        printf("\n QUEUE IS EMPTY");
    else
    {
        for(i = front;i <= rear;i++)
            printf("\t %d", queue[i]);
    }
}
```

## Linked Representation Of Queues

- ❖ We have seen how a queue is created using an array.
- ❖ Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size.
- ❖ If we allocate space for 50 elements in the queue and it hardly uses 20-25 locations, then half of the space will be wasted.
- ❖ And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.
- ❖ In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation.
- ❖ But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.

## Linked Representation Of Queues

- ❖ In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element.
- ❖ The START pointer of the linked list is used as FRONT.
- ❖ Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.
- ❖ All insertions will be done at the rear end and all the deletions will be done at the front end.
- ❖ If  $\text{FRONT} = \text{REAR} = \text{NULL}$ , then it indicates that the queue is empty.

## Linked Representation Of Queues

- ❖ **Operations on Linked Queues**
- ❖ **A queue has two basic operations: insert and delete.**
- ❖ **The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start of the queue.**
- ❖ **Apart from this, there is another operation peek which returns the value of the first element of the queue.**

# Linked Representation Of Queues

- **Insert Operation**
- In Step 1, the memory is allocated for the new node.
- In Step 2, the DATA part of the new node is initialized with the value to be stored in the node.
- In Step 3, we check if the new node is the first node of the linked queue.
  - If this is the case, then the new node is tagged as FRONT as well as REAR.
  - Also NULL is stored in the NEXT part of the node
- However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or

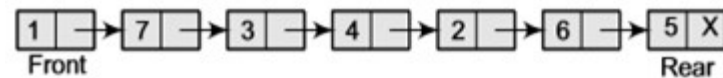


Figure 8.7 Linked queue

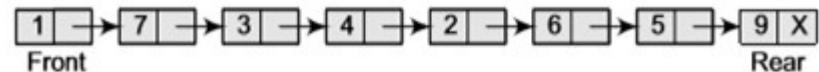


Figure 8.8 Linked queue after inserting a new node

```

Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR->DATA = VAL
Step 3: IF FRONT = NULL
        SET FRONT = REAR = PTR
        SET FRONT->NEXT = REAR->NEXT = NULL
      ELSE
        SET REAR->NEXT = PTR
        SET REAR = PTR
        SET REAR->NEXT = NULL
      [END OF IF]
Step 4: END
  
```

Figure 8.9 Algorithm to insert an element in a linked queue

# Linked Representation Of Queues

- ❖ **Delete Operation**
- ❖ **In Step 1, we first check for the underflow condition.**
- ❖ **If the condition is true then an appropriate message is displayed, otherwise in Step 2, we use a pointer PTR that points to FRONT.**
- ❖ **In Step 3, FRONT is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.**

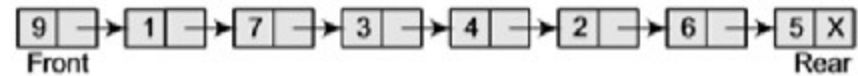


Figure 8.10 Linked queue

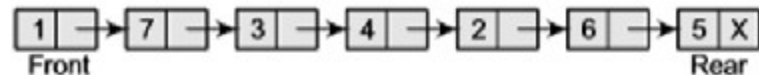


Figure 8.11 Linked queue after deletion of an element

```

Step 1: IF FRONT = NULL
        Write "Underflow"
        Go to Step 5
      [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
  
```

Figure 8.12 Algorithm to delete an element from a linked queue

# Linked Representation Of Queues

```
struct node
{
    int data;
    struct node *next;
};
struct queue
{
    struct node *front;
    struct node *rear;
};
struct queue *q;
void create_queue(struct queue *);
struct queue *insert(struct queue *,int);
struct queue *delete_element(struct queue *);
struct queue *display(struct queue *);
int peek(struct queue *);
```

# Linked Representation Of Queues

```
void create_queue(struct queue *q)
{
    q->rear = NULL;
    q->front = NULL;
}
struct queue *insert(struct queue *q,int val)
{
    struct node *ptr;
    ptr = (struct node*)malloc(sizeof(struct node));
    ptr->data = val;
    if(q->front == NULL)
    {
        q->front = ptr;
        q->rear = ptr;
        q->front->next = q->rear->next = NULL;
    }
    else
    {
        q->rear->next = ptr;
        q->rear = ptr;
        q->rear->next = NULL;
    }
    return q;
}
```



# Linked Representation Of Queues

```
struct queue *display(struct queue *q)
{
    struct node *ptr;
    ptr = q->front;
    if(ptr == NULL)
        printf("\n QUEUE IS EMPTY");
    else
    {
        printf("\n");
        while(ptr!=q->rear)
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
        printf("%d\t", ptr->data);
    }
    return q;
}
```

# Linked Representation Of Queues

```
struct queue *delete_element(struct queue *q)
{
    struct node *ptr;
    ptr = q->front;
    if(q->front == NULL)
        printf("\n UNDERFLOW");
    else
    {
        q->front = q->front->next;
        printf("\n The value being deleted is : %d", ptr->data);
        free(ptr);
    }
    return q;
}

int peek(struct queue *q)
{
    if(q->front==NULL)
    {
        printf("\n QUEUE IS EMPTY");
        return -1;
    }
    else
        return q->front->data;
}
```

## Types Of Queues

◆ A queue data structure can be classified into the following types:

◆ 1. Circular Queue

◆ 2. Deque

◆ 3. Priority Queue

◆ 4. Multiple Queue

◆ We will discuss each of these queues in detail in the following sections.

## Circular Queues

- ❖ In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.
- ❖ Look at the queue shown in Fig. 8.13.
- ❖ Here,  $\text{FRONT} = 0$  and  $\text{REAR} = 9$ .

54	9	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

**Figure 8.13** Linear queue

## Circular Queues

- ❖ Now, if you want to insert another value, it will not be possible because the queue is completely full.
- ❖ There is no empty space where the value can be inserted.
- ❖ Consider a scenario in which two successive deletions are made. The queue will then be given as shown in Fig. 8.14.
- ❖ Here,  $\text{FRONT} = 2$  and  $\text{REAR} = 9$ .
- ❖ Suppose we want to insert a new element in the queue shown in Fig. 8.14.
- ❖ Even though there is space available, the overflow condition still exists because the condition  $\text{REAR} = \text{MAX} - 1$  still holds true.
- ❖ This is a major drawback of a linear queue.

		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

**Figure 8.14** Queue after two successive deletions

## Circular Queues

- ❖ To resolve this problem, we have two solutions.
- ❖ First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently.
- ❖ But this can be very time-consuming, especially when the queue is quite large. The second option is to use a circular queue.
- ❖ In the circular queue, the first index comes right after the last index.
- ❖ Conceptually, you can think of a circular queue as shown in Fig. 8.15.
- ❖ The circular queue will be full only when  $(\text{REAR} + 1) \% \text{Max} = \text{FRONT}$ .

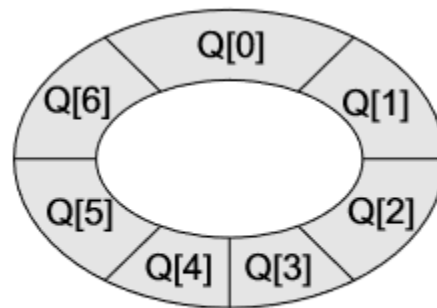
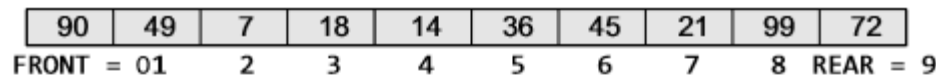


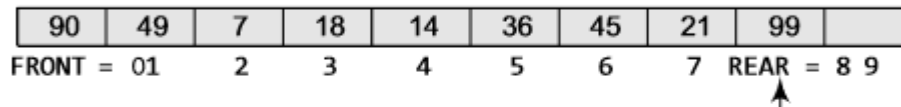
Figure 8.15 Circular queue

# Circular Queues

- ❖ For insertion, we now have to check for the following three conditions:
- ❖ If  $(\text{REAR} + 1) \% \text{Max} = \text{FRONT}$ , then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.
- ❖ If  $\text{REAR} \neq \text{MAX} - 1$ , then REAR will be incremented and the value will be inserted as illustrated in Fig. 8.17.
- ❖ If  $\text{FRONT} \neq 0$  and  $\text{REAR} = \text{MAX} - 1$ , then it means that the queue is not full. So, set  $\text{REAR} = 0$  and insert the new element there, as shown in Fig. 8.18.

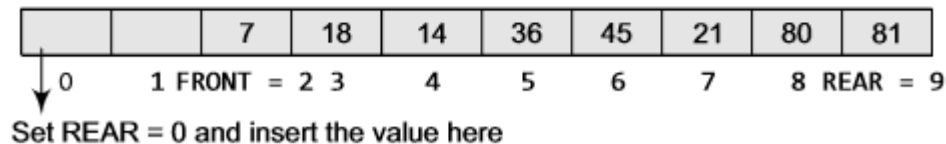


**Figure 8.16** Full queue



Increment rear so that it points to location 9 and insert the value here

**Figure 8.17** Queue with vacant locations



**Figure 8.18** Inserting an element in a circular queue

# Circular Queues

- ❓ Let us look at Fig. 8.19 which shows the algorithm to insert an element in a circular queue.
- ❓ In Step 1, we check for the overflow condition. In Step 2, we make two checks.
- ❓ First to see if the queue is empty, and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end.
- ❓ In Step 3, the value is stored in the queue at the location pointed by REAR.

```

Step 1: IF ( REAR + 1 ) % MAX = FRONT
        Write "OVERFLOW"
        Goto step 4
    [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
        SET FRONT = REAR = 0
    ELSE IF REAR = MAX - 1 and FRONT != 0
        SET REAR = 0
    ELSE
        SET REAR = REAR + 1
    [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
  
```

**Figure 8.19** Algorithm to insert an element in a circular queue



# Circular Queues

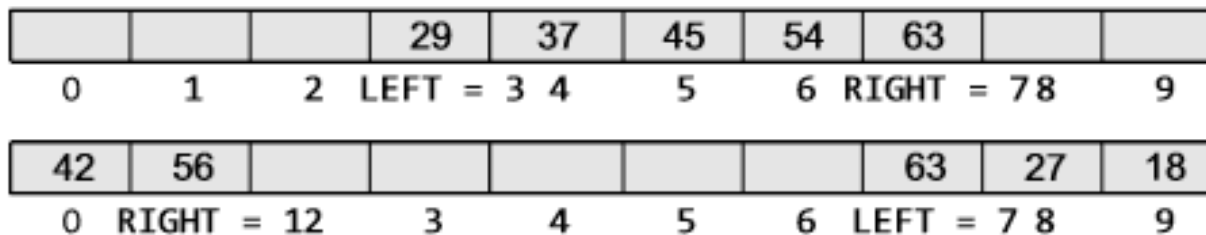
```
void insert()
{
    int num;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d", &num);
    if((rear+1)%MAX==front)
        printf("\n OVERFLOW");
    else if(front==-1 && rear==-1)
    {
        front=rear=0;
        queue[rear]=num;
    }
    else if(rear==MAX-1 && front!=0)
    {
        rear=0;
        queue[rear]=num;
    }
    else
    {
        rear++;
        queue[rear]=num;
    }
}
```

# Circular Queues

```
int delete_element()
{
    int val;
    if(front==-1 && rear==-1)
    {
        printf("\n UNDERFLOW");
        return -1;
    }
    val = queue[front];
    if(front==rear)
        front=rear=-1;
    else
    {
        if(front==MAX-1)
            front=0;
        else
            front++;
    }
    return val;
}
```

## Dequeues

- ❖ A deque (pronounced as 'deck' or 'dequeue') is a list in which the elements can be inserted or deleted at either end.
- ❖ However, no element can be added and deleted from the middle. In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list.
- ❖ In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque.
- ❖ The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N-1] is followed by Dequeue[0].
- ❖ Consider the deques shown in Fig. 8.24.



**Figure 8.24** Double-ended queues

## Dequeues

- ❖ **There are two variants of a double-ended queue.**
- ❖ **Input restricted deque: In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.**
- ❖ **Output restricted deque: In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.**

## Priority Queues

- ❖ **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:**
- ❖ **An element with higher priority is processed before an element with a lower priority.**
- ❖ **Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.**

# Priority Queues

- ❖ 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 systems to execute the highest priority process first.
- ❖ The priority of the process may be set based on the CPU time it requires to get executed completely.
  - For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.
- ❖ Another factor is the importance of one process over another.
  - In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

# Priority Queues

- ❖ **Implementation of a Priority Queue**
- ❖ **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 inefficient and usually a blend of these two approaches is adopted that takes roughly  $O(\log n)$  time or less.**

# Priority Queues

- ❖ **Linked Representation of a 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 we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.**
- ❖ **Consider the priority queue shown in Fig. 8.25.**
  - ❖ **Lower priority number means higher priority.**
  - ❖ **From the queue, we cannot make out whether A was inserted before E or whether E joined the queue before A because the list is not sorted based on FCFS.**
  - ❖ **However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.**

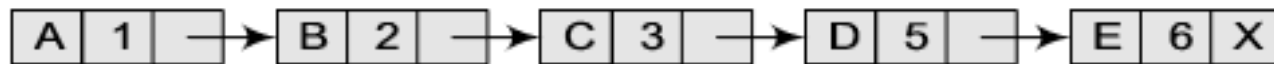


**Figure 8.25** Priority queue



# Priority Queues

- ❖ **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 a 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.
- ❖ If we have to insert a new element with data = F and priority number = 4, then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element.



**Figure 8.26** Priority queue



**Figure 8.27** Priority queue after insertion of a new node

## Priority Queues

❖ However, if we have a new element with data = F and priority number = 2, then the element will be inserted after B, as both these elements have the same priority but the insertions are done on FCFS basis as shown in Fig. 8.28.

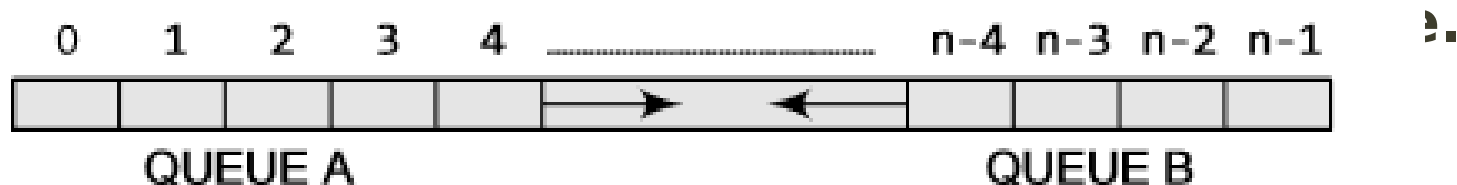


**Figure 8.28** Priority queue after insertion of a new node

❖ **Deletion:** Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

## Multiple Queues

- ❖ When we implement a queue using an array, the size of the array must be known in advance.
- ❖ If the queue is allocated less space, then frequent overflow conditions will be encountered.
- ❖ In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory.
- ❖ Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- ❖ So a better solution to deal with this problem is to have multiple queues or to have more than



**Figure 8.31** Multiple queues

# Priority Queues

```
struct node *insert(struct node *start)
{
    int val, pri;
    struct node *ptr, *p;
    ptr = (struct node *)malloc(sizeof(struct node));
    printf("\n Enter the value and its priority : " );
    scanf( "%d %d", &val, &pri);
    ptr->data = val;
    ptr->priority = pri;
    if(start==NULL || pri < start->priority )
    {
        ptr->next = start;
        start = ptr;
    }
    else
    {
        p = start;
        while(p->next != NULL && p->next->priority <= pri)
            p = p->next;
        ptr->next = p->next;
        p->next = ptr;
    }
    return start;
}
```

# Priority Queues

```
struct node *delete(struct node *start)
{
    struct node *ptr;
    if(start == NULL)
    {
        printf("\n UNDERFLOW" );
        return;
    }
    else
    {
        ptr = start;
        printf("\n Deleted item is: %d", ptr->data);
        start = start->next;
        free(ptr);
    }
    return start;
}
```

# Priority Queues

```
void display(struct node *start)
{
    struct node *ptr;
    ptr = start;
    if(start == NULL)
        printf("\nQUEUE IS EMPTY" );
    else
    {
        printf("\n PRIORITY QUEUE IS : " );
        while(ptr != NULL)
        {
            printf( "\t%d[priority=%d]", ptr->data, ptr->priority );
            ptr=ptr->next;
        }
    }
}
```