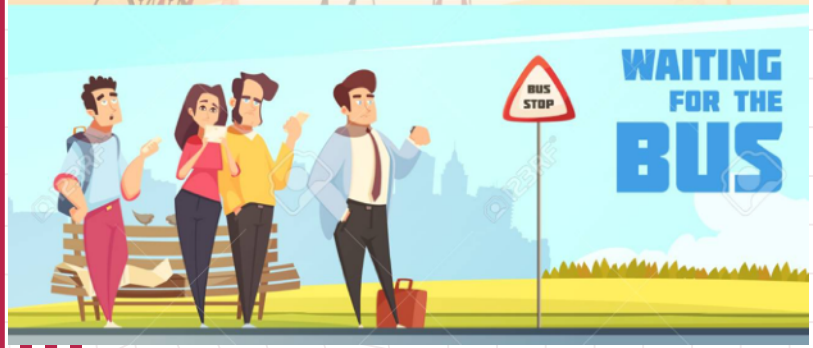
MODULE 1-5 [CPE11201]

QUEUE

คิว

Assoc. Prof. Dr. Natasha Dejdumrong


QUEUE
DOCTOR


BUS STOP
WAITING
FOR THE
BUS


Gates 46-54

OUTLINES

- Queue
- Array Representation of Queues
- Operations on a Queues
- Linked List Representation of Queues
- Operations on a Linked Queues
- Types of Queues
- Applications of Queues

Basic Idea

- Queue is an abstract data structure, somewhat similar to Stacks.
- Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

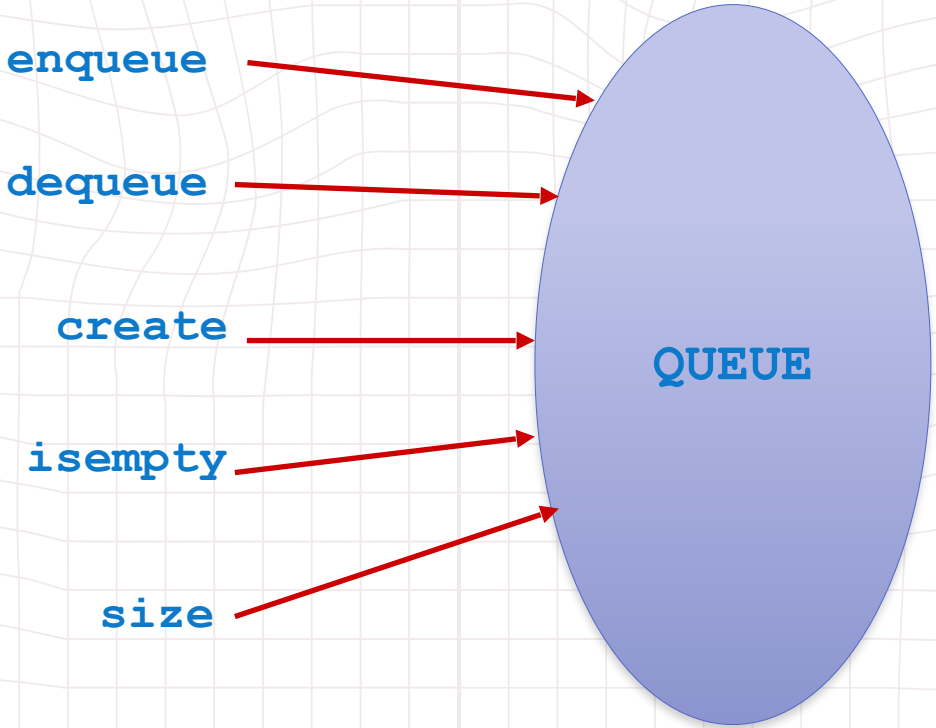
Queue

- In all these examples, 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. Next, we will see how queues are implemented using each of these data structures.

Queue Representation

★ As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.

Queue



QUEUE: First-In-First-Out (LIFO)

```
void enqueue (queue *q, int element);
                    /* Insert an element in the queue */
int dequeue (queue *q);
                    /* Remove an element from the queue */
queue *create();
                    /* Create a new queue */
int isempty (queue *q);
                    /* Check if queue is empty */
int size (queue *q);
                    /* Return the no. of elements in queue */
```

Array Representation of Queue

★ Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

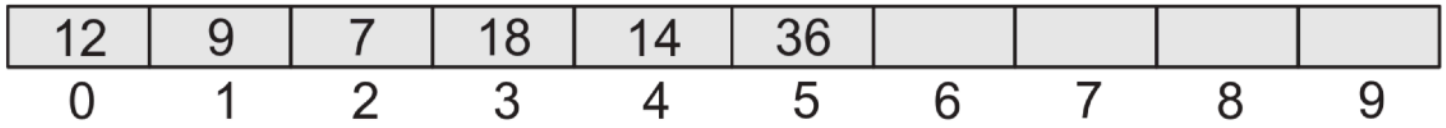


Figure 8.1 Queue

Array Representation of Queue

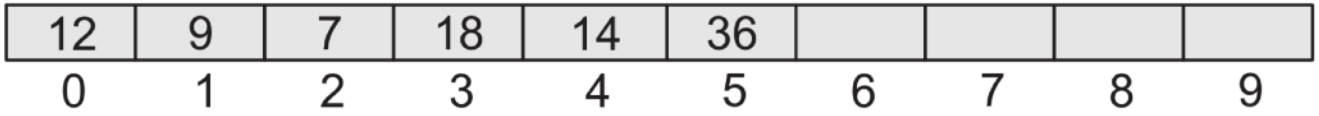


Figure 8.1 Queue

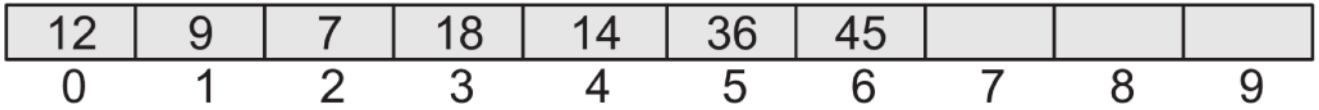


Figure 8.2 Queue after insertion of a new element

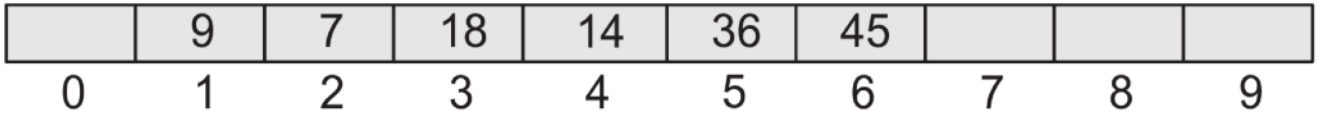


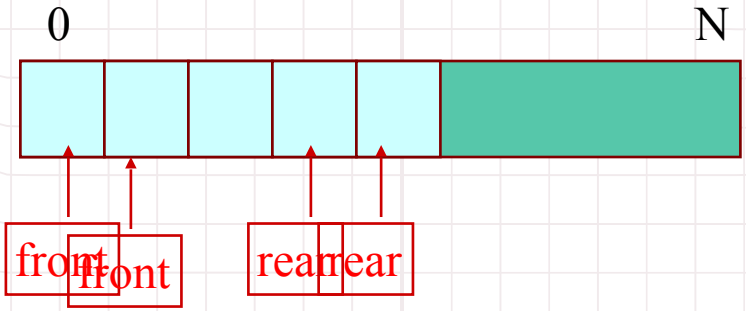
Figure 8.3 Queue after deletion of an element

Problem With Array Implementation

- The size of the queue depends on the number and order of enqueue and dequeue.
- It may be situation where memory is available but enqueue is not possible.

ENQUEUE DEQUEUE

Effective queuing storage area of array gets reduced.



Operations on a Queue

- A queue supports three basic operations: insert, and delete.
- The insert operation adds an element to the rear of the queue.
- The delete operation removes the element from the front of the queue.

INSERT Operation

```
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

INSERT Operation

```
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

```
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;
}
```

DELETE Operation

```
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

DELETE Operation

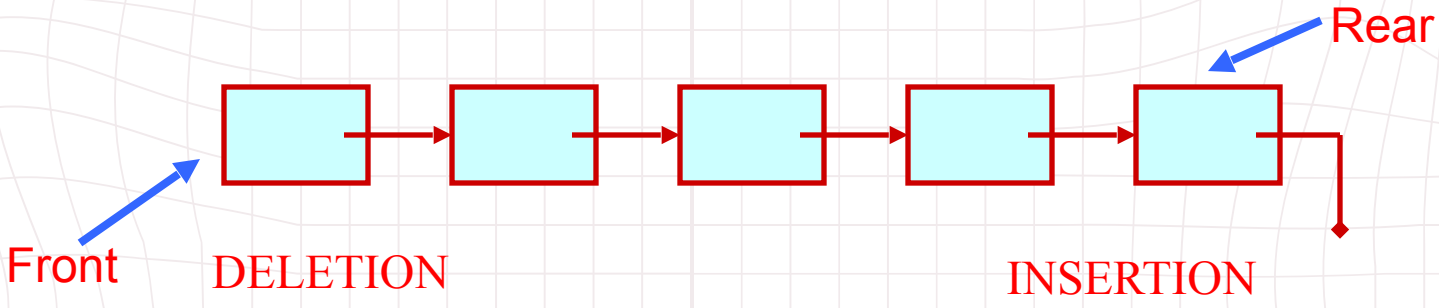
```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Figure 8.5 Algorithm to delete an element from a queue

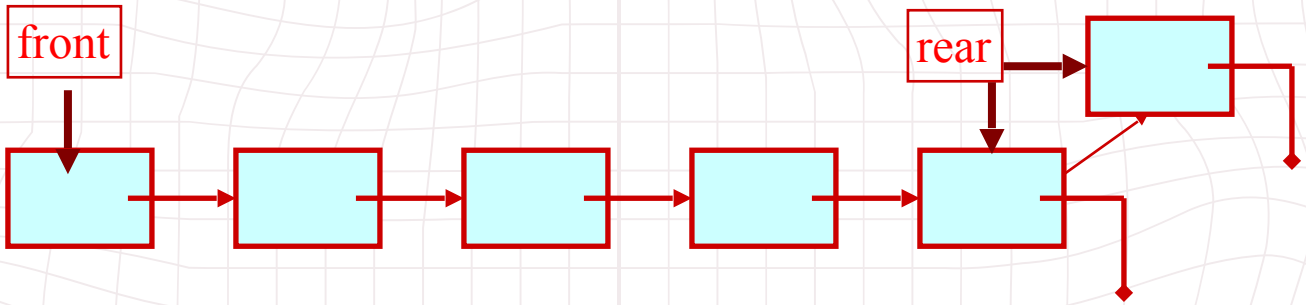
```
int delete_element()
{
        int val;
        if(front == -1 || front>rear)
        {
                printf("\n UNDERFLOW");
                return -1;
        }
        else
        {
                val = queue[front];
                front++;
                if(front > rear)
              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

Linked List Representation of Queues

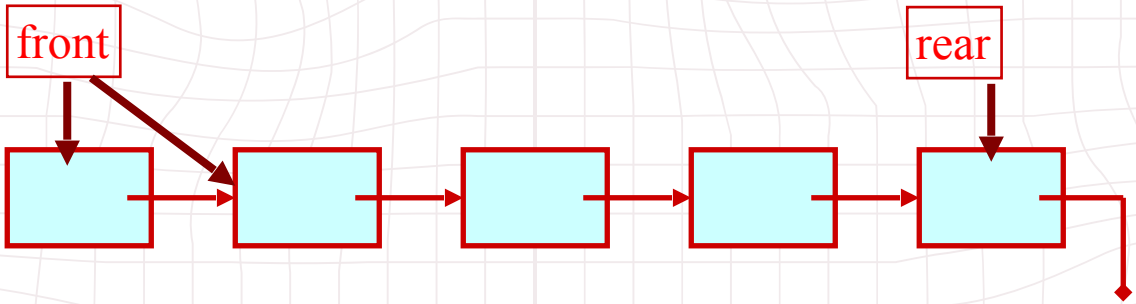
- Basic idea:
  - Create a linked list to which items would be added to one end and deleted from the other end.
  - Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted).
    - Another pointing to the end of the list (point where new elements will be inserted).



ENQueue: Linked List Structure



DEQueue: Linked List Structure



Linked List 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 List Representation of Queues

- The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).
- In a linked queue, every element has two parts,
one that stores the data and another that stores the address of the next element.

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.

INSERT Operation

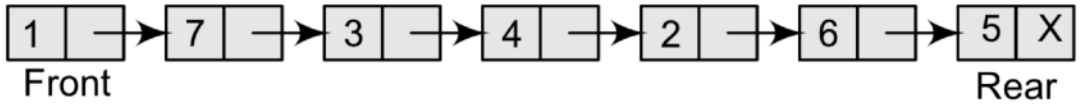


Figure 8.7 Linked queue

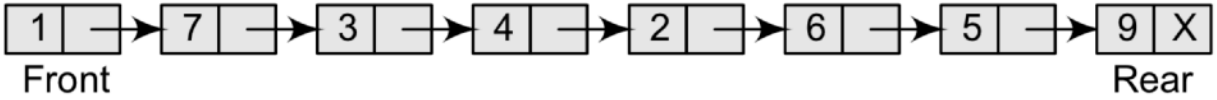


Figure 8.8 Linked queue after inserting a new node

INSERT Operation

```
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

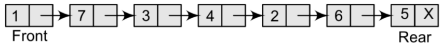
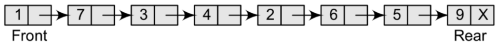


Figure 8.7 Linked queue




```
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;
}
```

Example :Queue using Linked List

```
struct qnode
{
  int val;
   struct qnode *next;
};

struct queue
{
   struct qnode *qfront, *qrear;
};
typedef struct queue QUEUE;
```

```
void enqueue (QUEUE *q,int element)
{
   struct qnode *q1;
   q1=(struct qnode *)malloc(sizeof(struct qnode));
   q1->val= element;
   q1->next=q->qfront;
   q->qfront=q1;
}
```

DELETE Operation

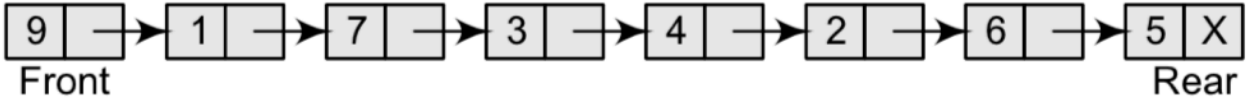


Figure 8.10 Linked queue

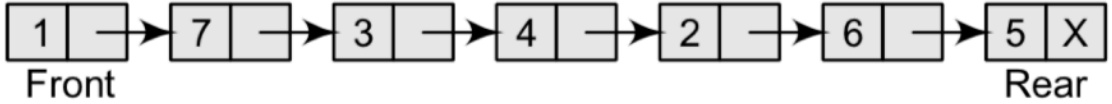


Figure 8.11 Linked queue after deletion of an element

DELETE Operation

```
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

```
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;
}
```

Example :Queue using Linked List

```
int size (queue *q)
{
    queue *q1;
    int count=0;
    q1=q;
    while(q1!=NULL)
    {
      q1=q1->next;
      count++;
    }
return count;
}
```

```
int peek (queue *q)
{
   queue *q1;
   q1=q;
   while(q1->next!=NULL)
      q1=q1->next;
return (q1->val);
}
```

```
int dequeue (queue *q)
{
   int val;
   queue *q1,*prev;
   q1=q;
   while(q1->next!=NULL)
   {
      prev=q1;
      q1=q1->next;
   }
   val=q1->val;
   prev->next=NULL;
   free(q1);
return (val);
}
```

Types of Queues

- A queue data structure can be classified into the following types:
  - Circular Queue
  - Deque
  - Priority Queue
  - Multiple Queue

Circular Queue

★ 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.

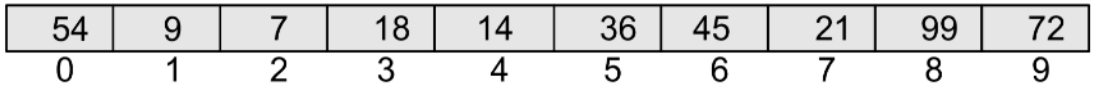


Figure 8.13 Linear queue

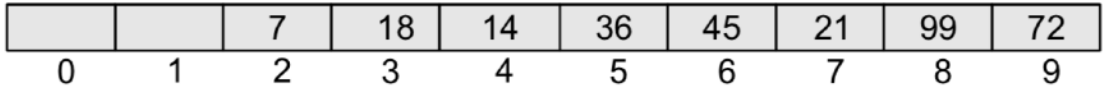


Figure 8.14 Queue after two successive deletions

★ 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 REAR = MAX – 1 still hold true. This is a major drawback of a linear queues

Circular Queue

- 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.

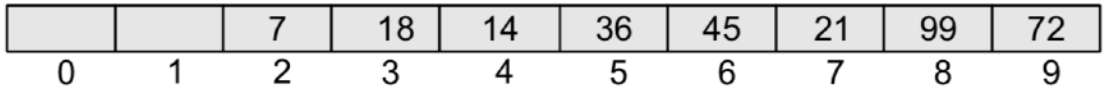


Figure 8.14 Queue after two successive deletions

Circular Queue

★ 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.

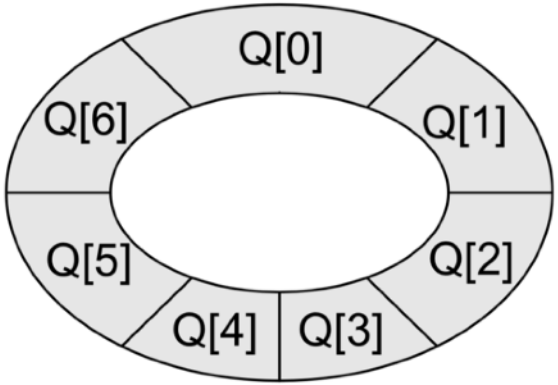


Figure 8.15 Circular queue

Circular Queue

Figure 8.15 Circular queue

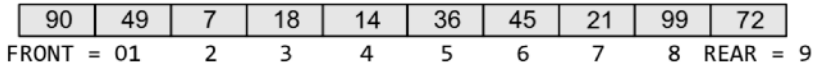


Figure 8.16 Full queue

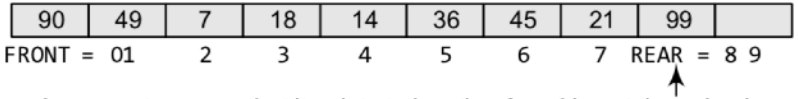


Figure 8.17 Queue with vacant locations

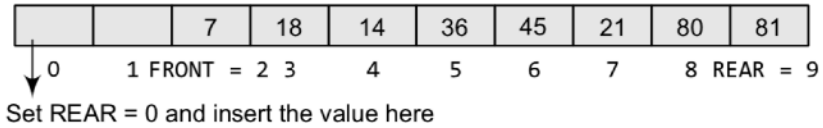


Figure 8.18 Inserting an element in a circular queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
```

The circular queue will be full only when FRONT = 0 and REAR = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations. For insertion, we now have to check for the following three conditions:

- If FRONT = 0 and REAR = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.
- If REAR != MAX – 1, then REAR will be incremented and the value will be inserted as illustrated in Fig. 8.17.
- If FRONT != 0 and REAR = MAX – 1, then it means that the queue is not full. So, set REAR = 0 and insert the new element there, as shown in Fig. 8.18.

Circular Queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            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

- If FRONT = 0 and REAR = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.
- If REAR != MAX – 1, then REAR will be incremented and the value will be inserted as illustrated in Fig. 8.17.
- If FRONT != 0 and REAR = MAX – 1, then it means that the queue is not full. So, set REAR = 0 and insert the new element there, as shown

Circular Queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            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

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

Figure 8.20 Empty queue

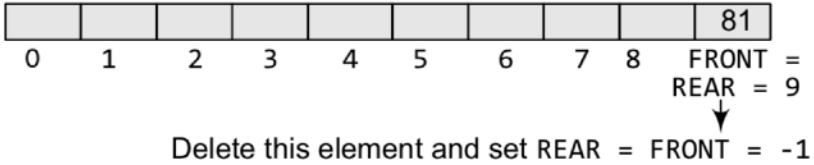


Figure 8.21 Queue with a single element

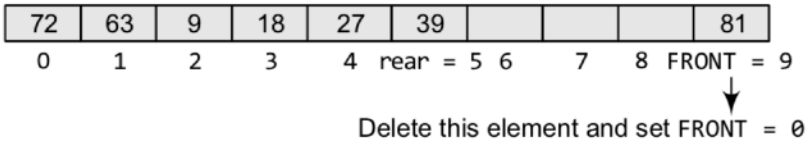


Figure 8.22 Queue where FRONT = MAX−1 before deletion

- Look at Fig. 8.20. If FRONT = -1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and FRONT = REAR, then after deleting the element at the front the queue becomes empty and so FRONT and REAR are set to -1. This is illustrated in Fig. 8.21.
- If the queue is not empty and FRONT = MAX−1, then after deleting the element at the front, FRONT is set to 0. This is shown in Fig. 8.22.

Circular Queue

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

Figure 8.23 Algorithm to delete an element from a circular queue

```
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;
}
```

Deques

- A deque (pronounced as 'deck' or 'dequeue') is a list in which the elements can be inserted or deleted at either end.
- It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) 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.

Deques

★ 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.

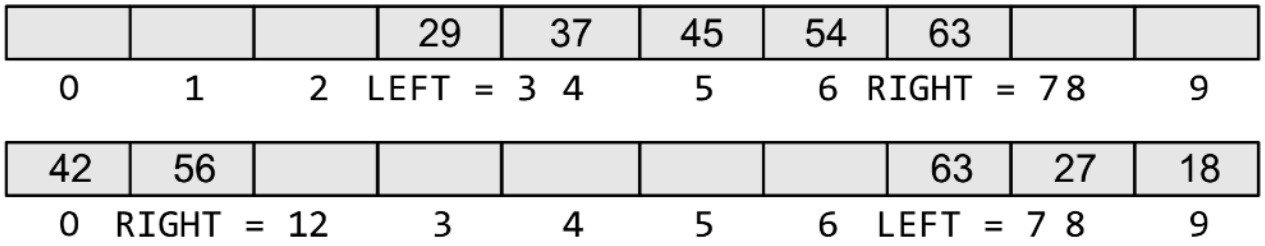


Figure 8.24 Double-ended queues

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
  - 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.

Multiple Queue

- 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.
- To deal with this problem, the code will have to be modified to reallocate more space for the array.
- 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 one queue in the same array of sufficient size.

Applications of Queue

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts.
When programming are real-time system that can be interrupted.

Questions and Answers