Queues are one of the most common data processing structures. They are frequently used in most system software such as operating systems, network and database implementations, and other areas. Queues are very useful in time-sharing and distributed computer systems where many widely distributed users share the system simultaneously. Whenever a user places a request, the operating system adds the request at the end of the queue of jobs waiting to be executed. The CPU executes the job at the front of the queue.

5.2 QUEUE AS ABSTRACT DATA TYPE

Look at the queue at the bus stop in Fig. 5.1. Here, the person to get inside the bus is the one who is at the front. The new person joining would stand at the rear end.

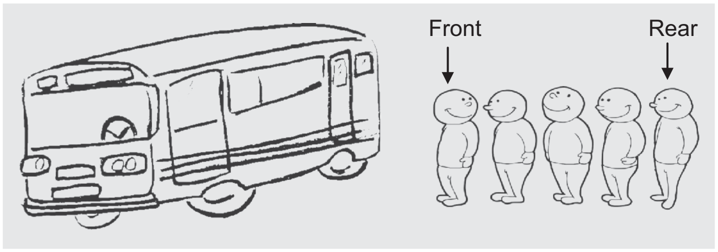


Fig. 5.1 Example of queue—passengers waiting at bus stop

To realize a queue as an abstract data type (ADT), we need a suitable data structure for storing the elements in the queue and the functions operating on it. The basic operations performed on the queue include adding and deleting an element, traversing the queue, checking whether the queue is full or empty, and finding who is at the front and who is at the rear ends.

A minimal set of operations on a queue is as follows:

1. `create()`—creates an empty queue, *Q*
2. `add(i,Q)`—adds the element *i* to the rear end of the queue, *Q* and returns the new queue
3. `delete(Q)`—takes out an element from the front end of the queue and returns the resulting queue
4. `getFront(Q)`—returns the element that is at the front position of the queue
5. `Is_Empty(Q)`—returns true if the queue is empty; otherwise returns false

The complete specification for the queue ADT is given in Algorithm 5.1.

ALGORITHM 5.1

```
class queue(element)
   declare create() → queue
   add(element, queue) → queue
   delete(queue) → queue
   getFront(queue) → queue
   Is_Empty(queue) → Boolean;
   For all Q ∈ queue, i ∈ element let
   Is_Empty(create()) = true
   Is_Empty(add(i,Q)) = false
   delete(create()) = error
   delete(add(i,Q)) =
      if Is_Empty(Q) then create
      else add(i, delete(Q))
   getFront(create) = error
   getFront(add(i, Q)) =
      if Is_Empty(Q) then i
      else getFront(Q)
   end
end queue
```

Since a queue is a linear data structure, it can be implemented using either arrays or linked lists. For the former, we use static memory allocation and for the latter, we use dynamic memory allocation. Let us see how a queue can be implemented using arrays.

5.3 REALIZATION OF QUEUES USING ARRAYS

We already know that an array is not a suitable data structure for frequent insertion and deletion of data elements. Another drawback of arrays is that they use static memory allocation, and so they can store only a fixed number of elements. In many practical applications, we come across a situation where the size of the data set keeps changing by such frequent insertions and deletions. Let us see the implementation of the various operations on the queue using arrays.

Create This operation should create an empty queue. Here `max` is the maximum initial size that is defined.

```
#define max 50
int Queue[max];
int Front = Rear = -1;
```

In addition to a one-dimensional array `Queue`, we need two more variables, `Front` and `Rear`. This declaration creates an empty queue of size `max`. The two variables `Front` and `Rear` are initialized to represent an empty queue. In general, it is suitable to set `Front` to one position behind the actual front of the queue and set the rear to the last element in the queue. Thus, the condition `Front = Rear` indicates an empty queue. As our array index ranges between `0` and `(max - 1)`, the front and rear are initialized to –1.

Is_Empty This operation checks whether the queue is empty or not. This is confirmed by comparing the values of `Front` and `Rear`. If `Front = Rear`, then `Is_Empty` returns true, else returns false.

```
bool Is_Empty()
{
   if(Front == Rear)
      return 1;
   else
      return 0;
}
```

Is_Full In the definition of the queue ADT, the function for checking the `Queue_Full` condition is not included. When we go in for an array implementation, due to its fixed size, we need to check the state of the queue for being full. It is recommended that before we delete an element from the queue, we must check whether the queue is empty or not. Similarly, before insertion, the queue must be checked for the `Queue_Full` state. When `Rear` points to the last location of the array, it indicates that the queue is full, that is, there is no space to accommodate any more elements.

```
bool Is_Full()
{
   if(Rear == max - 1)
      return 1;
   else
      return 0;
}
```

Add This operation adds an element in the queue if it is not full. As `Rear` points to the last element of the queue, the new element is added at the (rear + 1)th location.

```
void Add(int Element)

{
   if(Is_Full())
      cout << "Error, Queue is full";
   else
       Queue[++Rear] = Element;
}
```

Delete This operation deletes an element from the front of the queue and sets `Front` to point to the next element. `Front` can be initialized to one position less than the actual front. We should first increment the value of `Front` and then remove the element.

```
int Delete()
{
   if(Is_Empty())
      cout << "Sorry, queue is Empty";
   else
      return(Queue[++Front]);
}
```

getFront The operation `getFront` returns the element at the front, but unlike `delete`, this does not update the value of `Front`.

```
int getFront()
{
   if(Is_Empty())
      cout << "Sorry, queue is Empty";
   else
      return(Queue[Front + 1]);
}
```

Program Code 5.1 shows one way of realization of the queue ADT using arrays.

PROGRAM CODE 5.1

```
//Queue ADT
class queue
{
   private:
      int Rear, Front;
      int Queue[50];
      int max;
      int Size;
   public:
      queue()
      {
         Size = 0; max = 50;
         Rear = Front = -1 ;
      }
      int Is_Empty();
      int Is_Full();
      void Add(int Element);
      int Delete();
      int getFront();
};

int queue :: Is_Empty()
{
   if(Front == Rear)
      return 1;
   else
      return 0;
}
int queue :: Is_Full()
{
   if(Rear == max - 1)
      return 1;
   else
      return 0;
}

void queue :: Add(int Element)
```

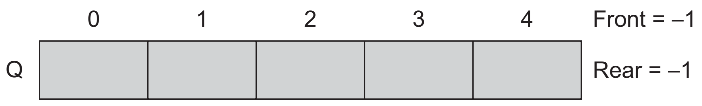
```
{
   if(!Is_Full())
      Queue[++Rear] = Element;
   Size++;
}

int queue :: Delete()
{
   if(!Is_Empty())
      {
         Size--;
         return(Queue[++Front]);
      }
}

int queue :: getFront()
{
   if(!Is_Empty())
      return(Queue[Front + 1]);
}
```

This implementation of queues using arrays has some flaws in it. Let us discuss these flaws through Program Code 5.2.

PROGRAM CODE 5.2

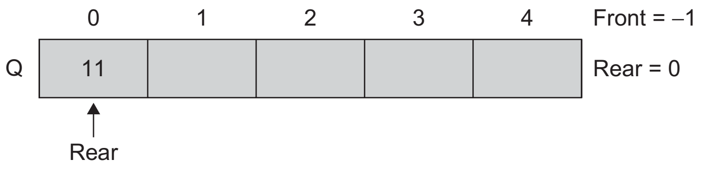
```
void main(void)
{
   queue Q;
   Q.Add(11);
   Q.Add(12);
   Q.Add(13);
   cout << Q.Delete() << endl;
   Q.Add(14);
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   Q.Add(15);
   Q.Add(16);
   cout << Q.Delete() << endl;
}
```

Let Q be an empty queue with `Front = Rear = –1`. Let `max = 5.`

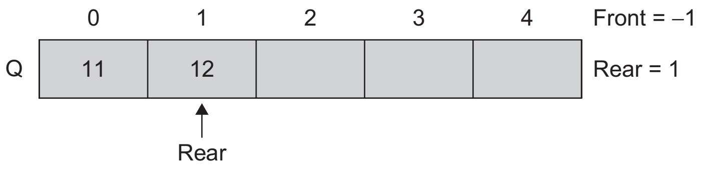


Consider the following statements:

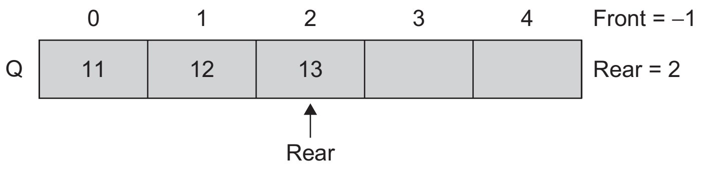
1. `Q.Add(11)`



2. `Q.Add(12)`

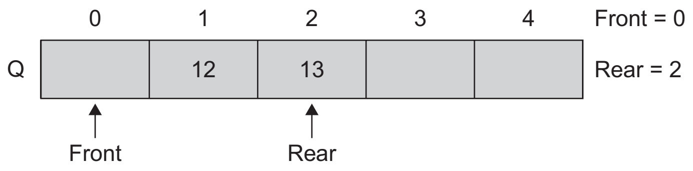


3. `Q.Add(13)`

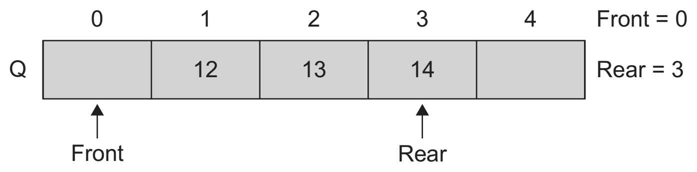


4. `A = Q.Delete()`

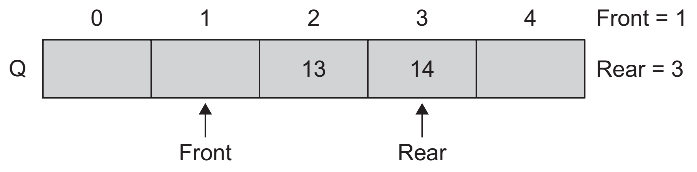
   Here, `A = Q[++Front] = Q[0] = 11`



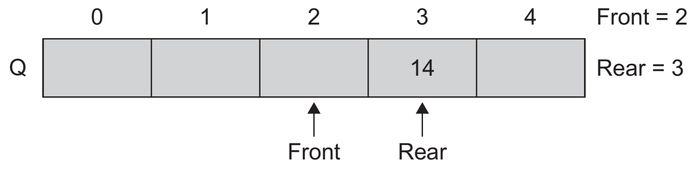
5. `Q.Add(14)`



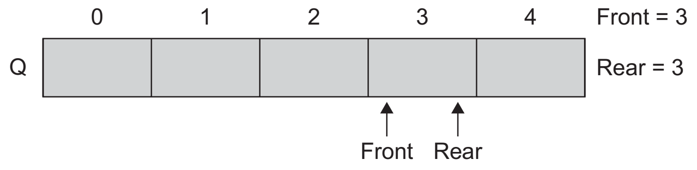
6. `A = Q.Delete()`
   `A = Q[++ Front] =   Q [1] = 12`



7. `A = Q.Delete()`
   `A = 13`

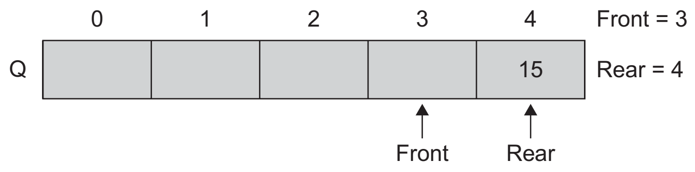


8. `A = Q.Delete()`



9. `A = Q.Delete()`
   Here we get the `Queue_empty` error condition as `Front = Rear = 3`
   Let us execute a few more statements.

10. `Q.Add(15)`



11. `Q.Add(16)`

This statement will generate the message `Queue_Full` because `Rear = 4`. If one carefully observes whether the queue is really full, it actually is not. The `Queue_Full` state should have five elements in it, whereas currently, there is only one element in the queue. This means that the implementation needs to be modified.

The precision of this implementation may be established in a manner similar to that used for stacks. With this setup, notice that unless the front regularly catches up with the rear and both the pointers are reset to zero, the `Queue_Full` condition does not necessarily indicate that it is full. One obvious thing to do when `Queue_Full` is signalled is to move the entire queue to the left so that the first element is again at the 0th location and `Front = −1`. This is obviously not a feasible solution as it is time consuming and involves a lot of data movement. This becomes impractical, especially when the queue is of a large size. The queue we have discussed so far is called the *linear queue*. There are two solutions to this problem: one is using a circular queue and the other is using a linked organization for realization of the queue. Let us discuss circular queues in Section 5.4.

5.4 CIRCULAR QUEUE

From the demonstration of the execution of a few `push` and `pop` operations it can be concluded that the linear queues using arrays have certain drawbacks listed as follows:

1. The linear queue is of a fixed size. So the user does not have the flexibility to dynamically change the size of the queue.
2. An arbitrarily declared maximum size of queues leads to poor utilization of memory. For example, the queue is declared of size 1000 and only 20 of them are used.
3. We need to write a suitable code to make the *front* regularly catch up with the *rear* and reset both. Array implementation of linear queues leads to the `Queue_Full` state even though the queue is not actually full.
4. To avoid this, when `Queue_Full` is signalled, we need to rewind the entire queue to the original start location (if there are empty locations) so that the first element is at the 0th location and `Front` is set to −1. Such movement of data is an efficient way to avoid this drawback.

The technique that essentially allows the queue to wraparound upon reaching the end of the array eliminates these drawbacks. Such a technique which allows the queues to wraparound from end to start is called a *circular queue*. Virtually, we want the insertion process and the rear to wraparound the queue.

Hence, a more efficient queue representation is obtained by implementing the array Q as circular. Here, as we go on adding elements to the queue and reach the end of the array, the next element is stored in the first slot of the array if it is empty. Suppose the queue Q is of size n. Now, if we go on adding elements in the queue, we may reach the location $n - 1$. If it is not circular, no more elements can be added even though there are empty locations at the front of the array. Instead, if there are empty locations at the front, using a circular queue we can add elements at that location rather than signalling an error as the queue is full or is shifting the data.

The empty slots will be filled with new incoming elements even though `Rear = n − 1`. Hence, the circular queue allows us to continue adding elements even though we have reached the end of the array. The queue is said to be full only when there are n elements in the queue. The pictorial representation of a circular queue is shown in Figs 5.2(a) and 5.2(b).

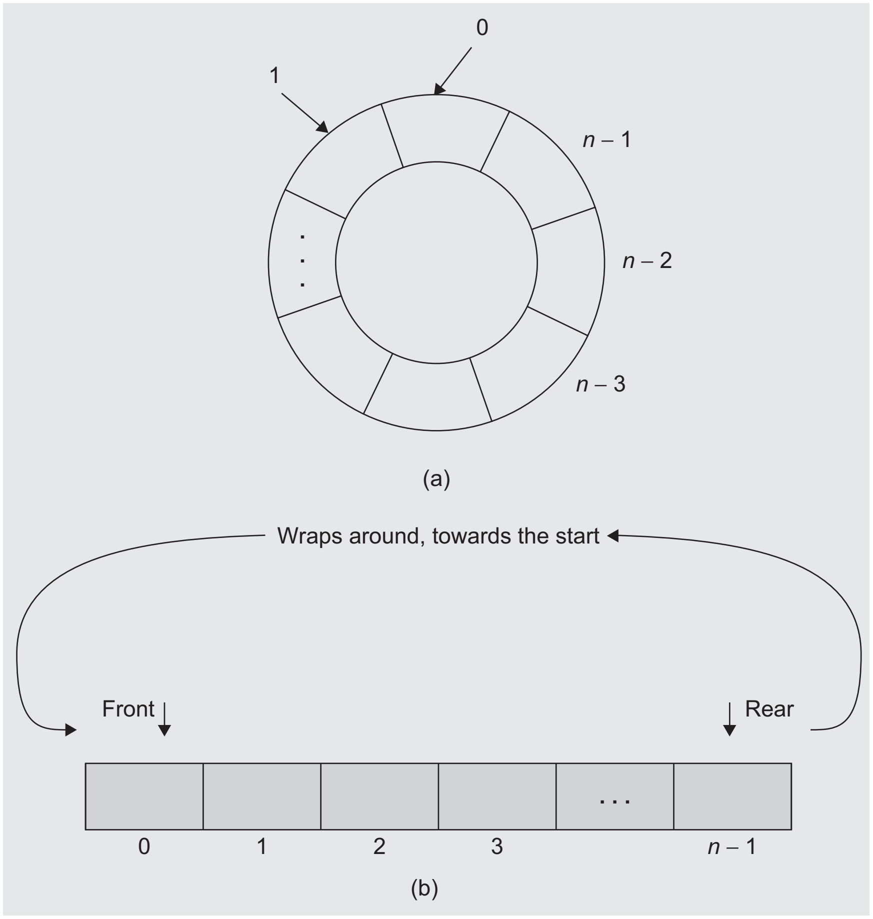


Fig. 5.2 Circular queue (a) Conceptual view (b) Physical view

Let us consider the queue *Q* which is of size *n*. We have already studied the operations on linear queues using arrays. We also studied its corresponding functions in the C++ language. Let us see whether the same functions can be used for circular queues. In a circular queue, when the *rear* is *n* – 1 and a new element is to be added, the *rear* should be set to 0.

Initially, both the *front* and the *rear* are set to –1. The value of *front* will always be one less than that of the actual *front*. The functions to add and delete elements are rewritten with a few modifications in Program Code 5.3.

PROGRAM CODE 5.3

```
#include<iostream.h>
class Cqueue
{
   private:
      int Rear, Front;
      int Queue[50];
      int Max;
      int Size;
   public:
      Cqueue() {Size = 0; Max = 50; Rear = Front = -1;}
      int Empty();
      int Full();
      void Add(int Element);
      int Delete();
      int getFront();
};

int Cqueue :: Empty()
{
   if(Front == Rear)
      return 1;
   else
      return 0;
}

int Cqueue :: Full()
{
   if(Rear == Front)
      return 1;
   else
      return 0;
}
```

```
void Cqueue :: Add(int Element)
{
   if(!Full())
      Rear = (Rear + 1) % Max;
   Queue[Rear] = Element;
   Size++;
}

int Cqueue :: Delete()
{
   if(!Empty())
      Front = (Front + 1) % Max;
   Size--;
   return(Queue[Front]);
}

int Cqueue :: getFront()
{
   int Temp;
   if(!Empty())
      Temp = (Front + 1) % Max;
   return(Queue[Temp]);
}

void main(void)
{
   Cqueue Q;
   Q.Add(11);
   Q.Add(12);
   Q.Add(13);
   cout << Q.Delete() << endl;
   Q.Add(14);
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   Q.Add(15);
   Q.Add(16);
   cout << Q.Delete() << endl;
}
```

The implementation of a circular queue using an array is provided in Program Code 5.3. Let us see its working with an example. Consider `max = 5` and initially, `Front = Rear = 0`. The iterations are shown in Table 5.1.

Table 5.1 Implementation of circular queue

0	1	2	3	4	Front	Rear	Action
					0	**0**	`Q_Empty`
	11				0	1	Insert 11
	11	12			0	2	Insert 12
	11	12	13		0	3	Insert 13
	11	12	13	14	0	4	Insert 14
	11	12	13	14	0		Insert 15 Can't insert 15 as `Q_Full` since `Rear` = (4 + 1)%5 = 0 which is equal to `Front`
	–	12	13	14	1	4	Delete
		–	13	14	2	4	Delete
			–	14	3	4	Delete
				–	4	4	Delete Can't delete as `Front = Rear` makes `Q_Empty`

To check the `Queue_Full` and `Queue_Empty` conditions, we need to check whether the values of `Front` and `Rear` are equal. In the programming languages C/C++, the array index varies from 0 to $n - 1$, so that one location of the circular queue always remains unused. Such is not the case in languages such as Pascal. Hence, in a circular queue that uses arrays in C/C++, we can store $n - 1$ elements, where n is declared as the size of the array. Hence, for storing n elements, we should declare the array of size $n + 1$.

5.4.1 Advantages of Using Circular Queues

The following are the merits of using circular queues:

1. By using circular queues, data shifting is avoided as the *front* and *rear* are modified by using the `mod()` function. The `mod()` operation wraps the queue back to its beginning.
2. If the number of elements to be stored in the queue is fixed (i.e., if the queue size is specific), the circular queue is advantageous.
3. Many practical applications such as printer queue, priority queue, and simulations use the circular queue.

5.5 MULTI-QUEUES

If more number of queues is required to be implemented, then an efficient data structure to handle multiple queues is required. It is possible to utilize all the available spaces in a single array. When more than two queues, say n, are represented sequentially, we can divide the available memory `A[size]` into n segments and allocate these segments to n queues, one to each. For each queue i we shall use `Front[i]` and `Rear[i]`. We shall use the condition `Front[i] = Rear[i]` if and only if the i^{th} queue is empty, and the condition `Rear[i] = Front[i]` if and only if the i^{th} queue is full.

If we want five queues, then we can divide the array `A[100]` into equal parts of 20 and initialize *front* and *rear* for each queue, that is, `Front[0] = Rear[0] = 0` and `Front[1] = Rear[1] = 20`, and so on for other queues (Fig. 5.3).

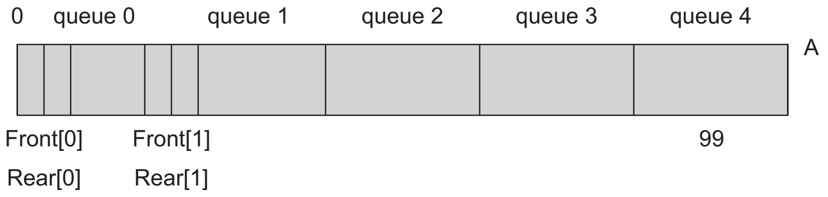


Fig. 5.3 A multi-queue

After adding elements 5 and 8 in the second queue, the resultant queue will be as in Fig. 5.4.

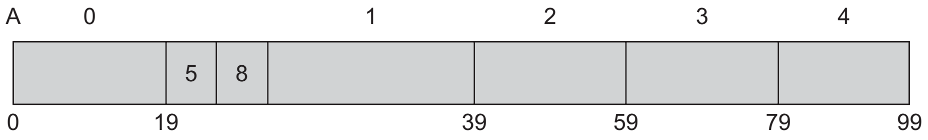


Fig. 5.4 Queue in Fig. 5.3 after addition of elements

5.6 DEQUE

The word *deque* is a short form of double-ended queue. It is pronounced as 'deck'. *Deque* defines a data structure where elements can be added or deleted at either the front end or the rear end, but no changes can be made elsewhere in the list. Thus, *deque* is a generalization of both a stack and a queue. It supports both stack-like and queue-like capabilities. It is a sequential container that is optimized for fast index-based access and efficient insertion at either of its ends. Deque can be implemented as either a continuous deque or as a linked deque. Figure 5.5 shows the representation of a deque.

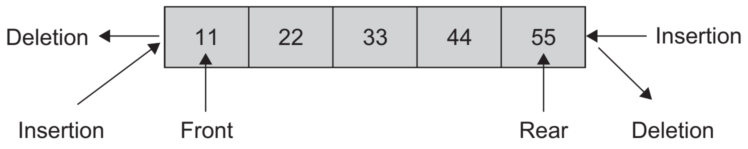


Fig. 5.5 Representation of a deque

The *deque ADT* combines the characteristics of stacks and queues. Similar to stacks and queues, a deque permits the elements to be accessed only at the ends. However, a deque allows elements to be added at and removed from either end. We can refer to the operations supported by the deque as `EnqueueFront`, `EnqueueRear`, `DequeueFront`, and `DequeueRear`. When we complete a formal description of the deque and then implement it using a dynamic, linked implementation, we can use it to implement both stacks and queues, thus achieving significant code reuse.

The following are the four operations associated with deque:

1. `EnqueueFront()`—adds elements at the front end of the queue
2. `EnqueueRear()`—adds elements at the rear end of the queue
3. `DequeueFront()`—deletes elements from the front end of the queue
4. `DequeueRear()`—deletes elements from the rear end of the queue

For stack implementation using deque, `EnqueueFront` and `DequeueFront` are used as `push` and `pop` functions, respectively.

Applications of deque Deque is useful where the data to be stored has to be ordered, compact storage is needed, and the retrieval of data elements has to be faster.

Variations of deque We can have two variations of a deque: the *input-restricted deque* and the *output-restricted* deque. The output-restricted deque allows deletions from only one end and the input-restricted deque allows insertions only at one end.

The functions to operate an output-restricted deque could be as follows:

`DequeueFront()`(or `DequeueRight()`), `EnqueueFront()`, and `EnqueueRear()`

The functions to operate an input-restricted deque are as follows:

`DequeueFront()`, `DequeueRight()`, and `EnqueueFront()`(or `EnqueueRear()`)

5.7 PRIORITY QUEUE

A *priority queue* is a collection of a finite number of prioritized elements. *Priority queues* are those in which we can insert or delete elements from any position based on some fundamental ordering of the elements. Elements can be inserted in any order in a priority queue, but when an element is removed from the priority queue, it is always the one with the highest priority.

In other words a priority queue is a collection of elements where the elements are stored according to their priority levels. The order in which the elements should be removed is decided by the priority of the element. The following rules are applied to maintain a priority queue:

1. The element with a higher priority is processed before any element of lower priority.
2. If there were elements with the same priority, then the element added first in the queue would get processed first.

Priority queues are used for implementing job scheduling by the operating system where jobs with higher priority are to be processed first. Another application of priority queues is in simulation systems where the priority corresponds to event times. The following are some examples of a priority queue:

1. A list of patients in an emergency room; each patient might be given a ranking that depends on the severity of the patient's illness.
2. A list of jobs carried out by a multitasking operating system; each background job is given a priority level. Suppose in a computer system, jobs are assigned three priorities, namely, *P*, *Q*, *R* as first, second, and third, respectively. According to the priority of the job, it is inserted at the end of the other jobs having the same priority. Consider the priority queue given in Fig. 5.6.

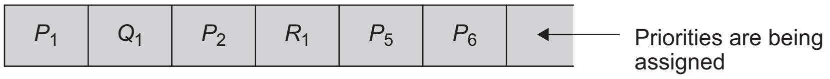


Fig. 5.6 System queue

There are two ways to implement priority queues.

Implementation method 1 The priority queue implementation in the first case can be visualized as three separate queues, each following the FIFO behaviour strictly as shown in Figs 5.7(a)–(c). In this example, jobs are always removed from the front of the queue. The elements in the second queue are removed only when the first queue is empty, and the elements from the third queue are removed only when the second queue is empty, and so on.

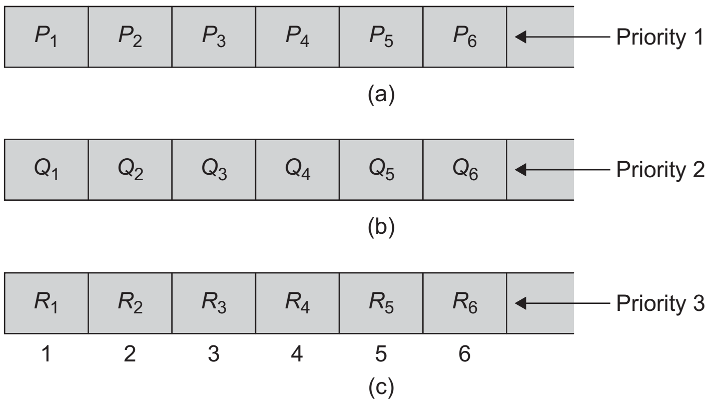

P_1 P_2 P_3 P_4 P_5 P_6
Priority 1
(a)
Q_1 Q_2 Q_3 Q_4 Q_5 Q_6
Priority 2
(b)
R_1 R_2 R_3 R_4 R_5 R_6
Priority 3
1 2 3 4 5 6
(c)

Fig. 5.7 System queues for each priority level (a) Priority 1 queue (b) Priority 2 queue (c) Priority 3 queue

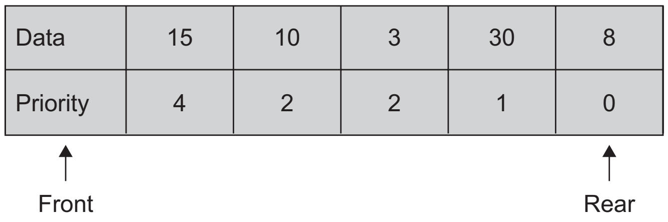
Operations on a priority queue The following is the list of operations performed on the priority queue PQ:

1. Initialize PQ to be the empty priority queue.
2. Determine if PQ is empty.
3. Determine if PQ is full.
4. If PQ is not full, insert an element *X* into PQ.
5. If PQ is not empty, remove an element *X* of the highest priority.

Implementation method 2 The second way of priority queue implementation is by using a structure for a queue. This is explained in the following statement:

```
typedef struct
{
   int Data;
   int priority;
}Element;

class PriorityQueue
{
   Private:
      Element PQueue[max];
   public:
      // member functions here
}
```

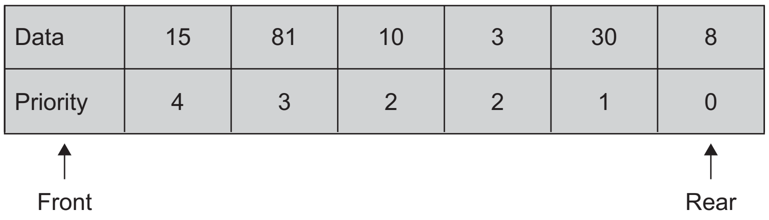
Figure 5.8 represents an example of a priority queue.



Data	15	10	3	30	8
Priority	4	2	2	1	0

Fig. 5.8 Priority queue

After inserting 81 with priority 3, the updated queue is given in Fig. 5.9.



Data	15	81	10	3	30	8
Priority	4	3	2	2	1	0

Fig. 5.9 Priority queue after insertion

The highest priority element is at the front and that of the lowest priority is at the rear. Here, when element 81 of priority 3 is to be added, it is inserted in between priorities 4 and 2 as shown in Fig. 5.9. When we want to delete an element, it behaves as a normal queue, that is, the element at front, which has the highest priority, is deleted first. The elements are sorted according to their priorities in descending order.

Hence, the two ways to implement a priority queue are *sorted list* and *unsorted list*.

Sorted list A sorted list is characterized by the following features:

1. *Advantage*—Deletion is easy; elements are stored by priority, so just delete from the beginning of the list.
2. *Disadvantage*—Insertion is hard; it is necessary to find the proper location for insertion.
3. A linked list is convenient for this implementation such as the list in Fig. 5.9.

Unsorted list An unsorted list is characterized by the following features:

1. *Advantage*—Insertion is easy; just add elements at the end of the list.
2. *Disadvantage*—Deletion is hard; it is necessary to find the highest priority element first.
3. An array is convenient for this implementation.

5.7.1 Array Implementation of Priority Queue

Like stacks and queues, even a priority queue can be represented using an array. However, if any array is used to store elements of a priority queue, then insertion of elements to the queue would be easy, but deletion of elements would be difficult. This is because while inserting elements in the priority queue, they are not inserted in an order. As a result, deleting an element with the highest priority would require examining the entire array to search for such an element. Moreover, an element in a queue can be deleted from the front end only.

There is no satisfactory solution to this problem. However, it would be more efficient if we store the elements in a priority queue. Each element in an array can have the following structure:

```
typedef struct
{
   int Data;
   int priority;
   int order;
}Element;
```

where `priority` represents the priority of the element and `order` represents the order in which the element has been added to the queue.

5.8 APPLICATIONS OF QUEUES

The most useful application of queues is the simulation of a real world situation so that it is possible to understand what happens in a real world in a particular situation without actually observing its occurrence.

Queues are also very useful in a time-sharing computer system where many users share a system simultaneously. Whenever a user requests the system to run a particular program, the operating system adds the request at the end of the queue of jobs waiting to be executed. Now, when the CPU is free, it executes the job that is at the front of the job queue. Similarly, there are queues for shared I/O devices too. Each device maintains its own queue of requests.

Another useful operation of queues is the solution of problems involving searching a non-linear collection of states. A queue is used for finding a path using the *breadth-first search* of graphs.

5.8.1 Josephus Problem

Let us consider a problem that can be solved in an easy manner using a circular queue. The problem is known as the *Josephus problem*, and it postulates a group of soldiers surrounded by an irresistible enemy force. There is no hope for victory without reinforcements, and there is only a single horse available for escape. The soldiers form a circle and a number n is picked. The name of one of the soldiers is also picked from a hat. Beginning with the soldier whose name is picked they begin to count clockwise around the circle. When the count reaches n, that soldier is removed from the circle, and the count begins again with the next soldier. The process continues so that each time the count reaches n, another soldier is removed from the circle. Any soldier removed from the circle is no longer counted. The last soldier left takes the horse and escapes. The problem is that, given a number n, the ordering of the soldiers in the circle, and the soldier from whom the count begins, one needs to determine the order in which soldiers are eliminated from the circle and which soldier escapes.

The input to the program is the number n and a list of names, which is the clockwise ordering of the circle, beginning with the soldier from whom the count is to start. The last input line contains the string `end`, indicating the end of the input. The program should print the names in the order in which they are eliminated and the name of the soldier who finally escapes.

For example, suppose that $n = 3$ and that there are five soldiers named A, B, C, D, and E. We count three soldiers starting at A so that C is eliminated first. We then begin at D and count D, E and then back to A so that A is eliminated next. Then we count B, D, and E (C has already been eliminated), and finally B, D, and B. Now, D is the one who escapes.

Clearly, a circular list in which each node represents one soldier is a natural data structure to use in solving this problem. It is possible to reach any node from any other by counting around the circle. To represent the removal of a soldier from the circle, a node is deleted from the circular list. Finally, when only one node remains on the list, the result is determined. The algorithm for this problem is given in Algorithm 5.2.

ALGORITHM 5.2

```
1. Let n be the number of members
2. Get the first member
3. Add all members to the queue
4. while (there is more than one member in the queue)
   begin
      count through n - 1 members in the queue;
      print the name of the nth member;
      Remove the nth member from the queue;
   end
5. Print the name of the only member in the list.
```

5.8.2 Job Scheduling

In the job-scheduling problem, we are given a list of n jobs. Every job i is associated with an integer deadline $d_i \geq 0$ and a profit $p_i \geq 0$. For any job i, profit is earned if and only if the job is completed within its deadline. A feasible solution with the maximum sum of profits is to be obtained.

To find the optimal solution and feasibility of jobs, we are required to find a subset J such that each job of this subset can be completed by its deadline. The value of a feasible solution J is the sum of profits of all the jobs in J.

The steps in finding the subset J are as follows:

1. $\Sigma\, p_i \times i \in J$ is the objective function chosen for the optimization measure.
2. Using this measure, the next job to be included should be the one that increases $\Sigma\, p_i \times i \in J$.
3. Begin with $J = \varnothing$, $\Sigma\, p_i = 0$, and $i \in J$.
4. Add a job to J, which has the largest profit.
5. Add another job to J bearing in mind the following conditions:
   (a) Search for the job that has the next maximum profit.
   (b) See if this job in union with J is feasible.
   (c) If yes, go to step 5 and continue; else go to (d).
   (d) Search for the job with the next maximum profit and go to step 2.
6. Terminate when addition of no more jobs is feasible.

Example 5.1 shows a job scheduling algorithm that works to yield an optimized high profit solution.

EXAMPLE 5.1 Consider five jobs with profits $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$ and maximum delay allowed $(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$.

Here, the maximum number of jobs that can be completed is

$$\text{Min}(n, \text{maxdelay}(d_i)) = \text{Min}(5, 3) = 3$$

Hence, there is a possibility of doing 3 jobs, and there are 3 units of time, as shown in Table 5.2.

Table 5.2 Job scheduling

Time slot	Profit	Job
0–1	20	1
1–2	15	2
2–3	0	3 cannot be accommodated
2–3	5	4
Total profit = 40		

In the first unit of time, job 1 is done and a profit of 20 is gained; in the second unit, job 2 is done and a profit of 15 is obtained. However, in the third unit of time, job 3 is not available, so job 4 is done with a gain of 5. Further, the deadline of job 5 has also passed; hence three jobs 1, 2, and 4 are completed with a total profit of 40.

5.8.3 Simulation

Any process or situation that we wish to simulate is considered as a system. A *system* may be defined as a group of objects interacting to produce some result. For example, an industry is a group of people and machines working together to produce some product.

A powerful tool that can be used to study the behaviour of systems is simulation.

Simulation is the process of forming an abstract model of a real world scenario to understand the effect of modifications and the introduction of various strategies on the situation. It allows the user to experiment with real and proposed situations without actually observing its occurrence. The major advantage of simulation is that it permits experimentation without modifying the real solution.

A model of the system must be produced to simulate a situation. Moreover, to determine the structure of a model, the entities, attributes, and activities of the system should be determined. Entities represent the objects of interest in the simulation. Attributes denote the characteristics of these entities. An activity is a process that causes a change of system state. An event is an occurrence of an activity at a particular instant of time. The state of the system at any given time is specified by the attributes of the entities and the relation between the entities at that time. The simulation program must schedule the events in the simulation so that the activities will occur in the correct time sequence.

Let us consider an example. Suppose that a person has to deposit his telephone bill. There are four service windows that can accept the bill. A person can deposit his bill at any of the service windows. Suppose a person enters the office at a specific time ($t1$) to deposit the bill, the transaction may be expected to take a certain period of time ($t2$) before it is completed. If a service window is free, the person can immediately deposit