6.4.2 Representing a Queue using a Linked List

A queue represented using a linked list is also known as a *linked queue*. The array based representation of queues suffers from following limitations

- Size of the queue must be known in advance.
- We may come across situations when an attempt to enqueue an element causes overflow. However, queue as an abstract data structure can not be full. Hence, abstractly, it is always possible to enqueue an element in a queue. Therefore, implementing queue as an array prohibits the growth of queue beyond the finite number of elements.

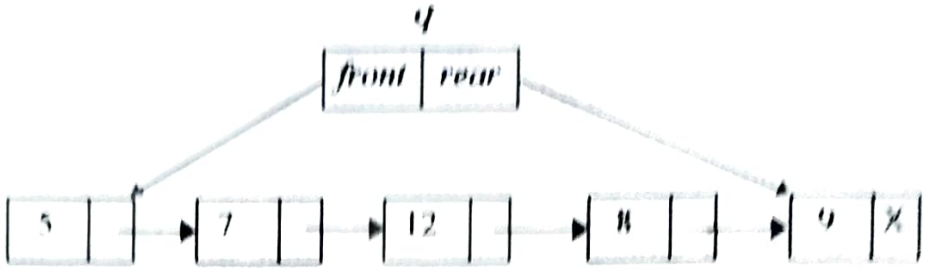
The linked list representation allow a queue to grow to a limit of the computer's memory.

The following are the necessary declarations

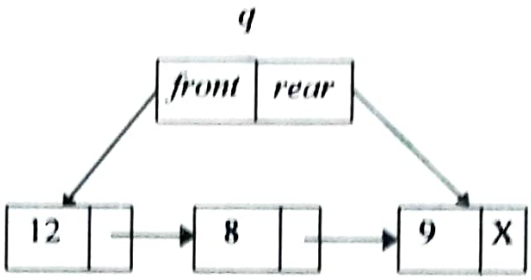
```
typedef struct node_type
{
    int  info;
    struct node_type *next;
 } node;

typedef struct
{
    node *front;
    node *rear;
} queue;

queue q;
```

Here we have defined two data type named *node* and *queue*. The *node* type, a self-referential structure, whose first element *info* hold the element of the queue and the second element *next* holds the address of the element after it in the queue. The second type is *queue*, a structure, whose first element *front* holds the address of the first element of the queue, and the second element *rear* holds the address of the last element of the queue. The last line declares a variable *q* of type *queue*.

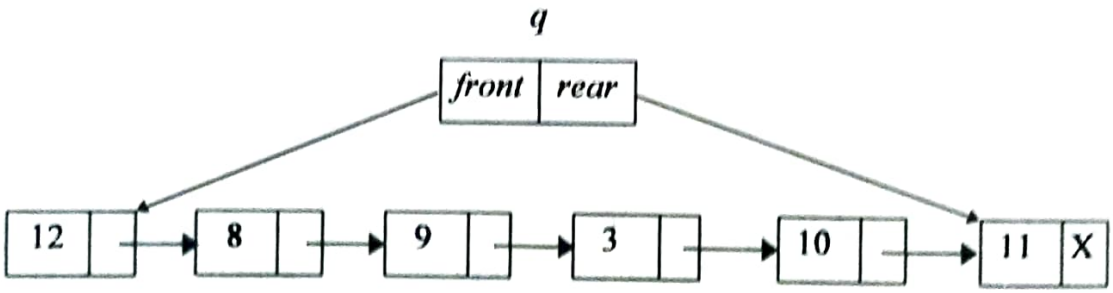
With these declarations, we will write functions for various operation to be performed on a stack represented using linked list.



(*a*) Representation of queue of Figure 6.1(*a*) in memory



(*b*) Representation of queue of Figure 6.1(*b*) in memory



(*c*) Representation of queue of Figure 6.1(*c*) in memory

Figure 6.5 Linked representation of queue of Figure 6.1

6.4.2.1 Implementation of Operations on a Linear Queue

The implementation of different operations on a linear queue are described in this section.

6.4.2.1.1 Creating an Empty Queue

Before we can use a queue, it is to be created/initialized. To initialize a queue, we will create an empty linked list. The empty linked list is created by setting pointer variables *front* and *rear* of a structure variable *q* to value NULL.

This simple task can be accomplished by the following function

Listing 6.10

```
void CreateQueue( queue *pq )
{
   pq->front = pq->rear = ( node * ) NULL;
}
```

6.4.2.1.2 Testing Queue for Underflow

The stack is tested for underflow condition by checking whether the linked list is empty. The empty status of the linked lists will be indicated by the NULL value of pointer variable *top*. This is shown in the following function

Listing 6.11

```
boolean IsEmpty( queue *pq )
{
   if ( pq->front == ( queue * ) NULL )
      return true;
   else
      return false;
}
```

The above function returns *boolean* value *true* if the test condition is satisfied; otherwise it returns value *false*.

The *IsEmpty()* function can also be written using conditional operator (?:) as

Listing 6.12

```
boolean IsEmpty( queue *pq )
{
   return ( pq->front == ( queue * ) NULL ? true : false);
}
```

6.4.2.1.3 Testing Queue for Overflow

Since a queue represented using a linked list can grow to a limit of a computer's memory, therefore overflow condition never occurs. Hence, this operation is not implemented for linked queues.

6.4.2.1.4 Enqueue Operation

To enqueue a new element onto the queue, the element is inserted in the end of the linked list. This task is accomplished as shown in the following function

Listing 6.13

```
void Enqueue( queue *pq, int value )
{
   node *ptr;
   ptr = ( node * ) malloc ( sizeof ( node ) );
   ptr->info = value;
   ptr->next = ( node * ) NULL;
   if ( pq->rear == ( node * ) NULL ) /*  queue initially empty  */
      pq->front = pq->rear = ptr;
   else
   {
      (pq->rear)->next = ptr;
      pq->rear = ptr;
   }
}
```

6.4.2.1.5 Dequeue Operation

To dequeue/remove an element from the queue, the element is removed from the beginning of the linked list. This task is accomplished as shown in the following function

Listing 6.14

```
int Dequeue( queue *pq )
{
   int temp;
   node *ptr;
   temp = (pq->front)->info;
   ptr = pq->front;
   if ( pq->front == pq->rear ) /*  only one element    */
       pq->front = pq->rear = ( node * ) NULL;
   else
       pq->front = (pq->front)->next;
```

```
    free ( ptr );
    return temp;
}
```

The element in the beginning of the linked list is assigned to a local variable *temp*, which later on will be returned via the *return* statement. And the first node from the linked list is removed.

6.4.2.1.6 Accessing Front Element

The element in the front of queue is accessed as shown in the following function

Listing 6.15

```
int Peek( queue *pq )
{
    return ( (pq->front)->info );
}
```

6.4.2.1.7 Disposing a Queue

Because queue is implemented using linked lists, therefore it is programmers job to write the code to release the memory occupied by the queue. The following listing shows the different steps to be performed.

Listing 6.16

```
void DisposeQueue ( queue *pq )
{
   node *ptr;
   while ( pq->front != ( node * ) NULL )
   {
      ptr = pq->front;
      pq->front = (pq->front)->next;
      free ( ptr );
   }
   pq->rear = ( node * ) NULL;
}
```

6.4 MULTIPLE QUEUES

We have seen that if we represent a queue using an array, we have to allocate sufficient space for the queue. If the space allocated to an array is less, it may result in frequent overflows and

we may have to modify the code and reallocate more space for array. On other side, if we attempt to reduce the number of overflows by allocating large space for array, it may result in sheer wastage of space if the typical size of the queue is very small as compared to actual size of the array. In this case, we may say that there exists a trade-off between the number of overflows and the space.

One possibility to reduce this trade-off is to represent more than one queue in the same array of sufficient size. Figure 6.6 shows the possible mapping of five queues to same array. Each queue is allocated a same amount of space.

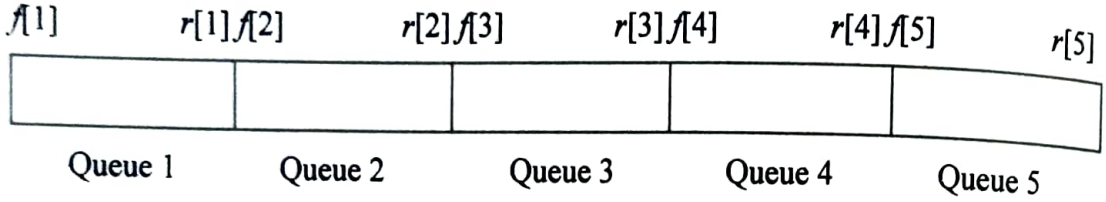


Figure 6.6 Representation of five queues by a single array

The implementations of different operations on these queues is also left as an exercise for the readers.

6.6 DEQUE

A *deque* is a kind of queue in which elements can be added or removed from at either end but not in the middle. The term deque is a contraction of the name *double-ended queue*.

There are two variations of a deque, which are intermediate between a deque and a queue. These are:

- ***Input-restricted deque*** – which allows insertions only at one end but allows deletions at both ends.
- ***Output-restricted deque*** – which allows insertions at both ends but allows deletions only at one end.

Deques are introduced here just for academic interest of the readers, there use in practical situations is almost nil.

6.7 PRIORITY QUEUE

A *priorities Queue* is a kind of queue in which each element is assigned a priority and the order in which elements are deleted and processed comes from the following rules:

- An element with highest priority is processed first before any element of lower priority.
- Two or more elements with the same priority are processed according to the order in which they were added to the queue.

There can be different criteria's for determining the priority. Some of them are summarized below:

- A shortest job is given higher priority over the longer one.
- An important job is given the higher priority over a routine type job. For example, a transaction for on-line booking of an order is given preference over payroll processing.
- In a commercial computer center, the amount you pay for the job can determine priority for your job. Pay more to get higher priority for your job.

6.7.1 Representing a Priority Queue in Memory

There are various ways of representing (maintaining) a priority queue in memory. These are:

- Using a linear linked list
- Using multiple queues, one for each priority
- Using a heap

6.7.1.1 Linear Linked List Representation

In this representation, each node of the linked list will have three field:

- An information field *info* that hold the element of the queue,
- A priority filed *prn* that holds the priority number of the element, and
- A next pointer filed *link* that holds the pointer to the next element of the queue.

Further, a node X precedes a node Y in the linear linked list if either node X has higher priority that Y or both nodes have same priority but X was added to the list before Y.

In a given system, we can assign higher priority to lower number or higher number. Therefore, if higher priority is for lower number, we have to maintain linear linked lists sorted on ascending order of the priority. However, if higher priority is for higher number, we have to maintain linear linked lists sorted on descending order of the priority. This way, the element to be processed next is available as the first element of the linked list.

6.7.1.2 Multiple Queue Representation

In this representation, one queue is maintained for each priority number. In order to process an element of the priority queue, element from the first non-empty highest priority number queue is accessed. In order to add a new element to the priority queue, the element is inserted in an appropriate queue for given priority number.

6.7.1.3 Heap Representation of a Priority Queue

A *heap* is a complete binary tree and with the additional property – the root element is either smallest or largest from its children. If root element of the heap is smallest from its children, it is known as *min heap*. If root element of the heap is largest from its children, it is known as *max heap*.

A priority queue having highest priority for lower number can be represented using min heap, and a priority queue having highest priority for higher number can be represented using max heap.

Hence, the element of the priority queue to be processed next is in the root node of the heap. Heaps are discussed in detail in *Chapter 8*.

If we compare the effort in adding or removing elements from the priority queue, we find that heap representation is the best.

6.8 APPLICATIONS OF QUEUES

Some of the applications of queues are listed below.

- There are several algorithms that use queues to solve problems efficiently. For example, we have used queue for performing level-order traversal of a binary tree in *Chapter 7*, and for performing breadth-first search on a graph in *Chapter 9*. Also in most of the simulation related problems, queues are heavily used.
- When the jobs are submitted to a networked printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a printer are placed on a queue.
- There is a kind of computer network where disk is attached to one computer, known as *file server*. Users on other computers are given access to files on a first-come first-served basis, so the data structure is a queue.
- Virtually every real-life line (supposed to be) a queue. For example, lines at ticket counters at cinema halls, railway stations, bus stands, etc., are queues, because the service (i.e. ticket) is provided on first-come first-served basis.

6.9 A QUICK REVIEW

In this chapter, we have learnt

- ➢ About the basics of a queue data structure.
- ➢ About the different operations performed on a queue.
- ➢ About the representation of a queue using a linear array.
- ➢ About the linear and circular queues.
- ➢ About the representation of a queue using linear linked list.
- ➢ About the applications of queues.
- ➢ About the deque data structure.
- ➢ About the priority queue data structure.
- ➢ About the representation of a priority queue.

In the next chapter, we will discuss trees in length.