Rules for Infix to postfix

1. Print operand as they arrive
2. If stack is empty or contain a left parathesis on top, push the incoming operator onto the stack
3. If incoming symbol is ‘(’ push in to stack
4. If incoming symbol is ‘)’ pop the stack and print the operators until left parathesis is found
5. If incoming symbol has higher precedence then the top of the stack, push it on the stack
6. If incoming symbol has lower precedence’s then the top of the stack,pop and print the top then test the incoming operator against the new top of the stack
7. If incoming operator has equal precedence with the top of the stack use associativity rule
8. At the end of the expression , pop and print all operators of the stack

Associativity L to R then pop and print the top of the stack and then push the incoming operator

R to L then push the incoming operator

```c
#include<stdio.h>
#include<stdlib.h>
 typedef struct node
{
   int data;
   int prty;
   struct node *next;
}node;

void insert();
void disp();
struct node *start=NULL;
```

```c

void main()
{
  int ch;
  while(1)
  {
  printf("\n----Options Are----");
  printf("\n1-Insertion");
  printf("\n2-Display");
  printf("\n3-Exit");
  printf("\nEnter your choice: ");
  scanf("%d",&ch);
  switch(ch)
  {
    case 1: insert();
            break;
    case 2: disp();
            break;
    case 3: exit(0);
    default: printf("\n Wrong Choice");
  }
 }
}

```

```
36
37  void insert()
38  {
39     node *ptr;
40     ptr = (node*)malloc(sizeof(node));
41     printf("\nEnter the value: ");
42     scanf("%d",&ptr->data);
43     printf("\nEnter the priority: ");
44     scanf("%d",&ptr->prty);
45     ptr->next=NULL;
46     if(start==NULL)
47       start=ptr;
48    else if(ptr->prty<start->prty)
49     {
50        ptr->next=start;
51        start=ptr;
52     }
53     else
54     {
55         node *temp;
56         temp=start;
57         while((temp->next!=NULL) && ((temp->next)->prty<ptr->prty) )
58          temp=temp->next;
59         ptr->next=temp->next;
60         temp->next=ptr;
61     }
62  }
63
```

```
63
64  void disp()
65  {
66    node *temp;
67    temp = start;
68    if(temp==NULL)
69      printf("\nList is empty");
70    else
71   {
72       printf("\nYou have entered following data\n");
73       printf("Priority  Data\n");
74      while(temp!=NULL)
75    {
76       printf(" %d          %d ",temp->prty,temp->data);
77       temp=temp->next;
78       printf("\n");
79    }
80   }
81  }
82
```

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.

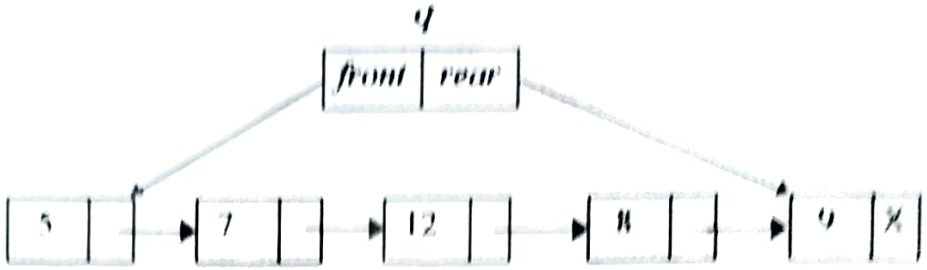
The linked list representation allow a queue to grow to a limit of the computer's memory.

The following are the necessary declarations

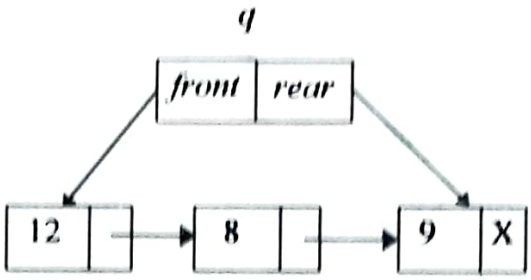
```
typedef struct node_type
{
    int  info;
    struct node_type *next;
 } node;

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*.

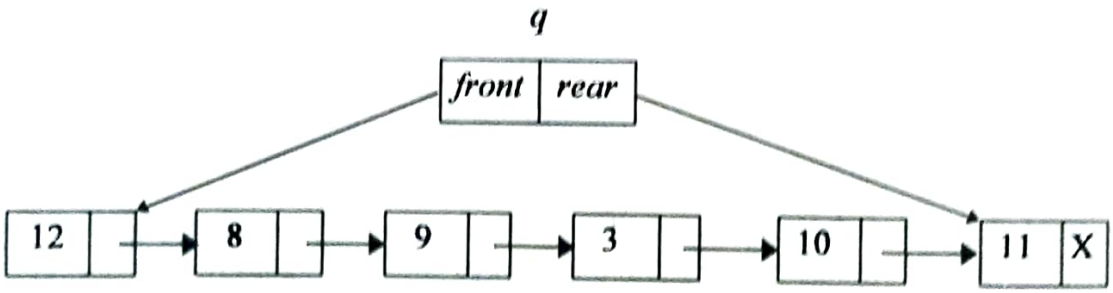
With these declarations, we will write functions for various operation to be performed on a stack represented using linked list.



(*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.

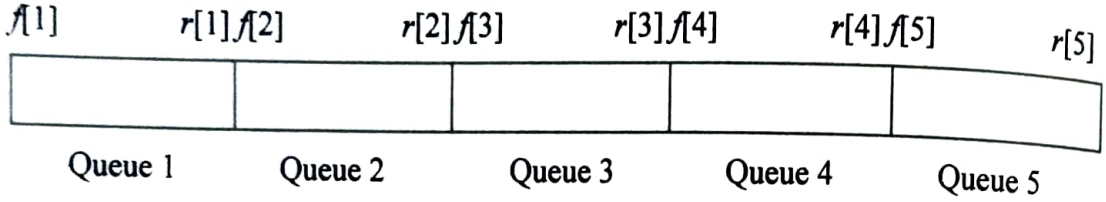


Figure 6.6 Representation of five queues by a single array

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.

6.1 INTRODUCTION

All of us are familiar with queues. In simple terms, a queue is a line of persons waiting for their turn at some service counter/window. A service counter can be a ticketing window of a cinema hall, a ticketing window of a bus stand or railway station, a service counter of a bank, a service counter of a gas agency, a fees deposit counter in a college, etc.. Depending on the type of service provided by the service counter and the number of persons interested in this service, there can be queues of varying lengths. The service at the service counter is provided on the First-Come First-Serve (FCFS) basis i.e. in order of their arrival in the queue.

Suppose that at a service counter, t_1 units of time is needed to provide a service to a single person, and on average a new person arrives at the service counter in t_2 units of time. The following possibilities may arise:

1. If $t_1 < t_2$, then the service counter will be free for some time before a new person arrives at the service counter. Hence, no queue in this case.
2. If $t_1 > t_2$, then service counter will be busy for some time even after the arrival of a new person. Therefore, this person have to wait for some time. Similarly, other persons arriving at the service counter have to wait. Hence, a queue will be formed.
3. If $t_1 = t_2$, then as a person leaves the service counter, a new persons arrives at the service counter. Hence, no queue in this case also.

This chapter describes a queue, its representation in computer memory, and the implementation of different operations that can be performed on them. It also describes a deque and a priority queue in brief.

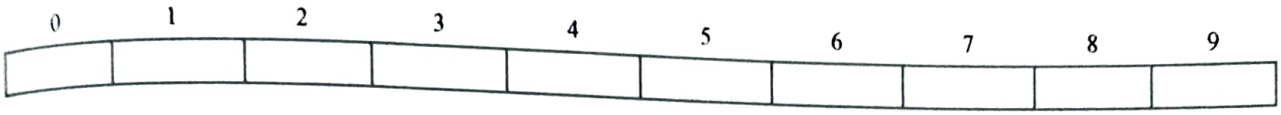
6.2 QUEUE DEFINED

In the context of data structures, a *queue* is a linear list in which insertions can take place at one end of the list, called the *rear* of the list, and deletions can take place only at other end, called the *front* of the list. The behaviour of a queue is like a Fast-In-First-Out (FIFO) system. Figure 6.1 shows a queue as black-box in which elements enters from one side and leaves from other side.

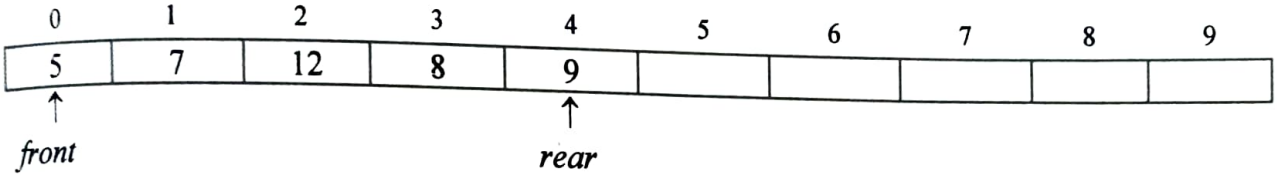
In queue terminology, the insertion and deletion operations are known as *enqueue* and *dequeue* operations.

Example 6.1

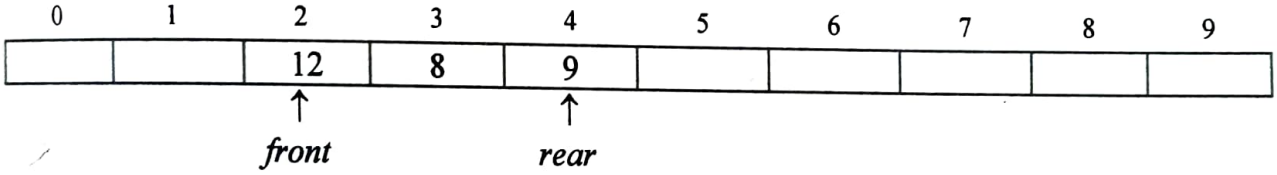
The following figure illustrates a queue which can accommodate maximum of 10 elements.



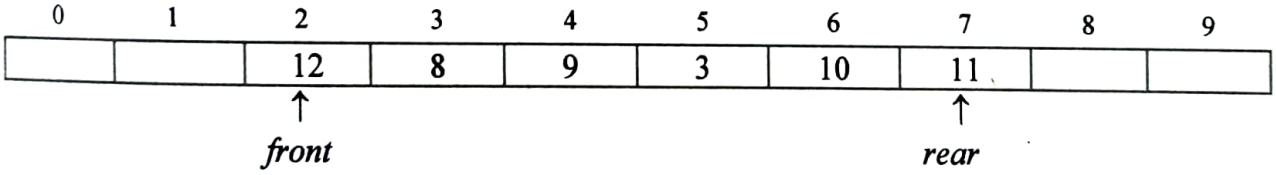
(*a*) An empty queue



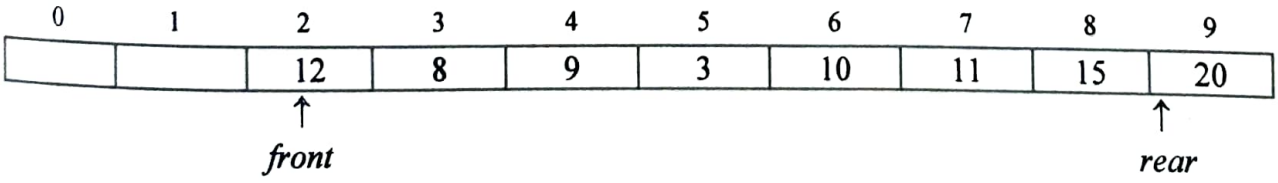
(*b*) Queue after enqueuing elements 5, 7, 12, 8, and 9 in order



(*c*) Queue after dequeuing elements 5, 7 from queue



(*d*) Queue after enqueuing elements 3, 10, 11 in a queue



(*e*) Queue after enqueuing elements 15 and 20 in a queue

Figure 6.1 Illustration of a queue whose elements are integers values

6.3 OPERATIONS ON QUEUES

The following operations are performed on queues.

1. **CreateQueue(q)** – to create q as an empty queue.
2. **Enque(q, i)** – to insert (add) element i in a queue q.
3. **Dequeue(q)** – to access and remove an element of queue q.
4. **Peek(q)** – to access the first element of the queue q without removing it from the queue q.
5. **IsFull(q)** – to check whether the queue q is full.
6. **IsEmpty(q)** – to check whether the queue q is empty.

All of these operation runs in O(1) time.

Note that if the queue q is empty then it is not possible to dequeue the queue q. Similarly, as there is no element in the queue q, the *Peek(q)* operations is also not valid. Therefore, we must ensure that the queue q is not empty before attempting to perform these operations.

Likewise, if the queue q is full then it is not possible to enqueue a new element in to the queue q. Therefore, we must ensure that the queue q is not full before attempting to perform enqueue operation.

6.4 REPRESENTATION OF QUEUES IN MEMORY

Queues, like stacks, can also be represented in memory using a linear array or a linear linked list. Even using a linear, we have two implementation of queues viz. *linear* and *circular*. The representation of these two is same, the only difference is in their behaviour. To start with, let us consider array representation of queues.

6.4.1 Representing a Queue using an Array

To implement a queue we need two variables, called *front* and *rear*, that holds the index of the first and last element of the queue and an array to hold the elements of the queue. Let us suppose that the elements of the queue are of integer type, and the queue can store maximum of 10 such elements i.e. queue size is 10.

The following are the necessary declarations

```
#define  MAX  10

typedef struct
{
   int front, rear;
   int elements[MAX];
} queue;

queue q;
```

We have defined our own data type named *queue*, which is a structure whose first element *front* will be used to hold the index of the first element of the queue. The second element *rear* will be used to hold the index of the last element of the queue. An array *elements* of size MAX of integer type to hold the elements of the queue. The last line declares variable *q* of type *queue*.

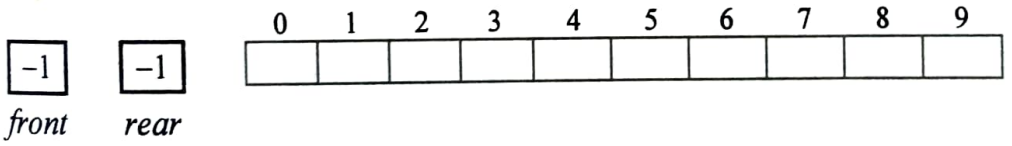
In addition to above declarations, we will use the declaration

```
typedef enum { false, true } boolean;
```

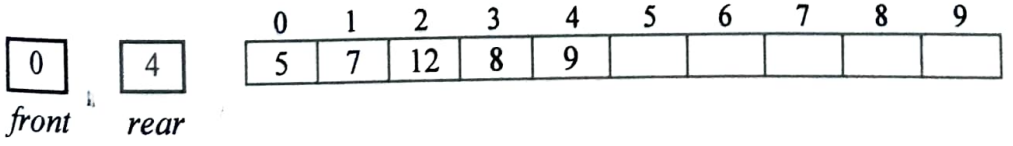
to signify the use of *boolean* kind of data and/or operations. This statement defined new data type named *boolean* which can take value *false* or *true*.

With these declarations, we will write functions for various operation to be performed on queue.

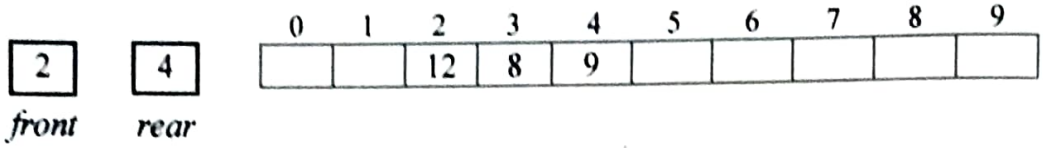
With these declarations, queue of Figure 6.1 will be represented in computer memory as shown in Figure 6.2.



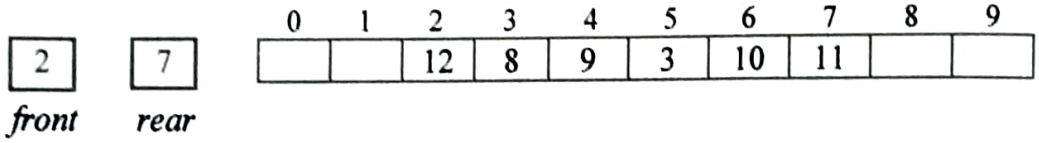
(*a*) Representation of queue of Figure 6.1(*a*) in memory



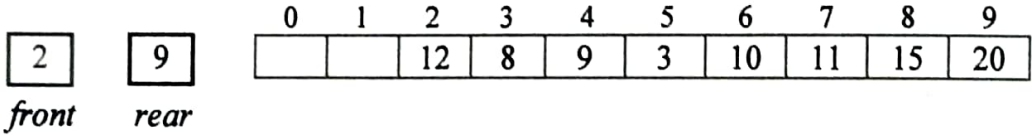
(*b*) Representation of queue of Figure 6.1(*b*) in memory



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



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



(*e*) Representation of queue of Figure 6.1(*d*) in memory

Figure 6.2 Array representation of linear queue of Figure 6.1

You must have observed, as shown in Figure 6.2(*e*), that it is not possible to enqueue more elements as such, though two positions in the linear queue are vacant. To overcome this problem, the elements of the queue are moved forward, so that the vacant positions are shifted towards the rear end of the linear queue. After shifting, *front* and *rear* are adjusted properly, and then the element is enqueued in the linear queue as usual. For example. We want to enqueue one more element, say 60, after making adjustments the queue will look as shown in Figure 6.3 .

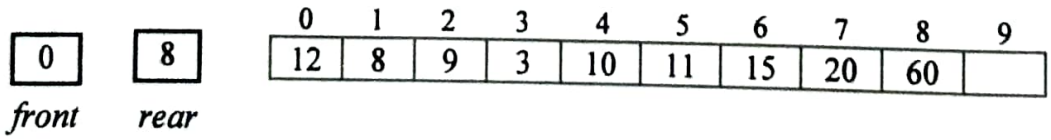
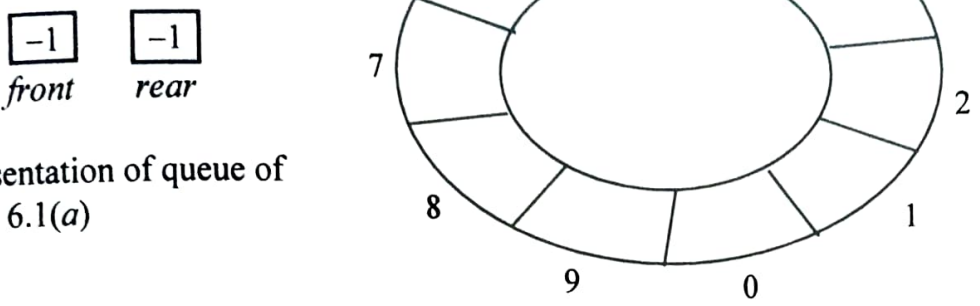
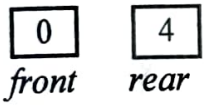


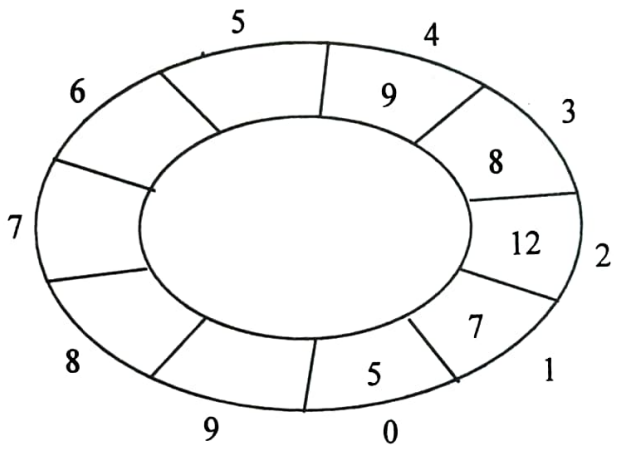
Figure 6.3 State of the queue after making adjustments and then enqueueing element 60

However, this difficulty can be overcome if we treat the queue position with index 0 as a position that comes after position with index 9 i.e. we treat the queue as circular as shown in Figure 6.4.

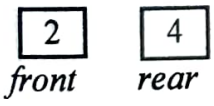


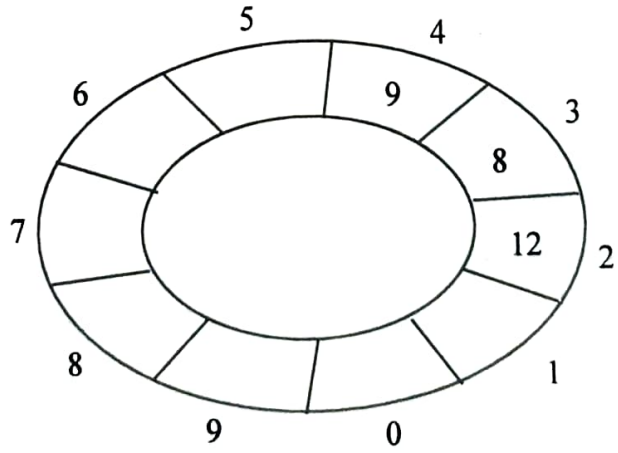
(*a*) Representation of queue of Figure 6.1(*a*)



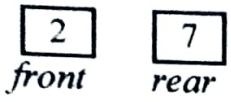


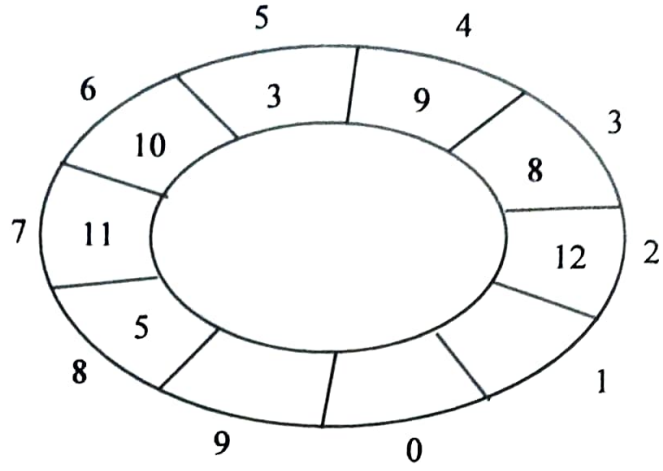
(*b*) Representation of queue of Figure 6.1(*b*)



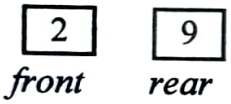


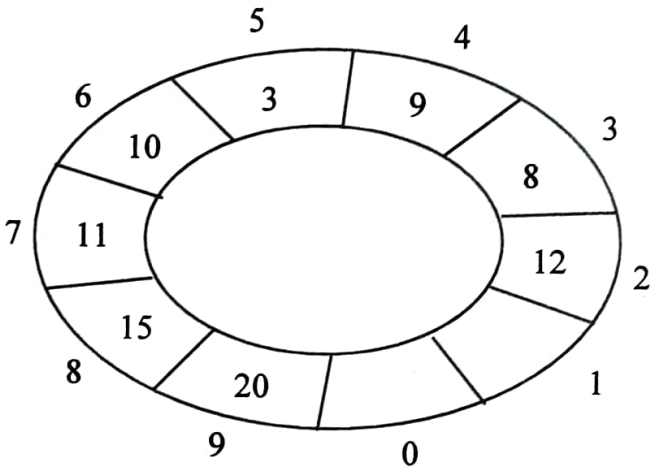
(*c*) Representation of queue of Figure 6.1(*c*)





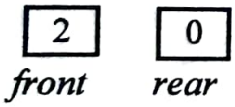
(*d*) Representation of queue of Figure 6.1(*d*)

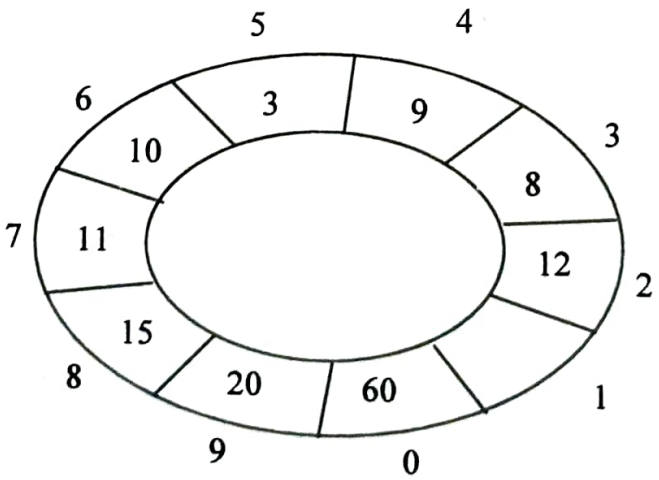




(*e*) Representation of queue of Figure 6.1(*e*)

Now to enqueue one more element, say 60, the *rear* is reset to 0, and the elements is inserted at that index as shown below.





(*f*) Representation of queue of Figure 6.3

Figure 6.4 (*a*) – (*f*) Illustration of Operations on Circular Queues

6.4.1.1 Implementation of Operations on a Linear Queue

The implementation of different operations on a linear queue are described below.

6.4.1.1.1 Creating an Empty Linear Queue

Before we can use a queue, it must be created/initialized. As the index of array elements can take any value in the range 0 to MAX–1, the purpose of initializing the queue is served by assigning value –1 (as sentinel value) to *front* and *rear* variables. This simple task can be accomplished by the following function

Listing 6.1

```
void CreateQueue( queue *v )
{
   v->front = v->rear = -1;
}
```

6.4.1.1.2 Testing a Linear Queue for Underflow

Before we remove an element from a queue, it is necessary to test whether a queue still have some elements i.e. to test that whether the queue is empty or not. If it is not empty then the dequeue operation can be performed to remove the front element. This test is performed by comparing the value of *front* with sentinel value –1 as shown in the following function

Listing 6.2

```
boolean IsEmpty( queue *v )
{
   if ( v->front == -1  )
      return true;
   else
      return false;
}
```

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

6.4.1.1.3 Testing a Linear Queue for Overflow

Before we insert new element in a queue, it is necessary to test whether queue still have some space to accommodate the incoming element i.e. to test that whether the queue is full

or not. If it is not full then the ***enqueue*** operation can be performed to insert the element at rear end of the queue. This test is performed by test condition

```
( front = 0 ) and ( rear = MAX - 1 )
```

i.e. front element is at position with index 0 and the last element is at position with index MAX–1, which indicates that no position in linear queue is vacant.

This task is accomplished by the following function

Listing 6.3

```
boolean IsFull( queue *v )
{
    if (  ( v->front == 0 ) && ( v->rear == (MAX - 1) ) )
        return true;
    else
        return false;
}
```

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

6.4.1.1.4 Enqueue Operation on Linear Queue

Even if a linear queue is not full, there is another problem that may arise during enqueue operation. That problem is that, the value of the *rear* variable is equal to MAX–1, though the value of the *front* variable is not equal to 0. This possibility implies that the incoming element cannot be enqueued unless the elements are moved forward, and variables *front* and *rear* are adjusted accordingly. Which is, of course, a time consuming process.

There are two conditions, which can occur, even if the queue is not full. These are

- If a linear queue is empty, then the value of the *front* and *rear* variables will be NIL (sentinel value), then both *front* and *rear* are set to 0.
- If a linear queue is not empty, then there are further two possibilities:
  - ◊ If the value of the *rear* variable is less than MAX–1, then the *rear* variable is incremented.
  - ◊ If the value of the *rear* variable is equal to MAX–1, then the elements of the linear queue are moved forward, and the *front* and *rear* variables are adjusted accordingly.

This task is accomplished as shown in the following function

Listing 6.4

```
void Enqueue( queue *v, int value )
{
   if ( IsEmpty(v) )
      v->front = v->rear = 0;
   else if ( v->rear == ( MAX - 1) )
   {
      for ( i = v->front, i <= v->rear; i++ )
           v->elements[i - v->front] = v->elements[i];
      v->rear = v->rear - v->front + 1;
      v->front = 0;
   }
   else
      v->rear++;
   v->elements[v->rear] = value;
}
```

6.4.1.1.5 Dequeue Operation on a Linear Queue

The front element of a linear queue is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the front element of a linear queue to a local variable, the value of the *front* variable is modified so that it points to the new front.

There are two possibilities:

- If there was only one element in a linear queue, then after dequeue operation queue will become empty. This state of a linear queue is reflected by setting *front* and *rear* variables to sentinel value NIL.
- Otherwise the value of the *front* variable is incremented.

This task is accomplished as shown in the following function

Listing 6.5

```
int Dequeue( queue *v )
{
   int temp;
   temp = v->elements[v->front];
   if ( v->front == v->rear )
      v->front = v->rear = -1;
   else
```

```
        v->front++;
    return temp;
}
```

6.4.1.1.6 Accessing Front Element

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

Listing 6.6

```
int Peek( queue *v )
{
    return (v->elements[v->front]);
}
```

6.4.1.2 Limitations of a Linear Queue

The only limitation of linear queue is that —

If the last position of the queue is occupied, it is not possible to enqueue any more elements even though some positions are vacant towards the front positions of the queue.

However, this limitation can be overcome by moving the elements forward, such that the first element of the queue goes to position with index 0, and the rest of the elements move accordingly. And finally the *front* and *rear* variables are adjusted appropriately.

But this operation may be very time consuming if the length of the linear queue is very long. As mentioned earlier, this limitation can be overcome if we treat that the queue position with index 0 comes immediately after the last queue position with index MAX–1. The resulting queue is known as *circular queue*.

6.4.1.3 Implementation of Operations on a Circular Queue

The operations of initialization, testing for underflow for circular queue is similar to that of a linear queue. However, other operations are slightly different. These are described below.

6.4.1.3.1 Testing a Circular Queue for Overflow

Before we insert new element in a circular queue, it is necessary to test whether a circular queue still have some space to accommodate the incoming element i.e. to test that whether a

circular queue is full or not. If it is not full then the enqueue operation can be performed to insert the element at rear of a circular queue. This test is performed by test conditions

- (front = 0) and (rear = MAX-1)
- front = rear + 1

If any one of these two conditions is satisfied, it means circular queue is full.

This task is accomplished by the following function

Listing 6.7

```
boolean Isfull( queue *v )
{
   if  (((v->front==0)&&(v->rear==(MAX-1)))||(v->front==v->rear+1))
      return true;
  else
      return false;
}
```

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

6.4.1.3.2 Enqueue Operation on a Circular Queue

Before the enqueue operation, there are three conditions which can occur, even if a circular queue is not full. These are

- If the queue is empty, then the value of the *front* and *rear* will be –1 (sentinel value), then both *front* and *rear* are set to 0.
- If the queue is not empty then the value of the *rear* will be the index of the last element of the queue, then the *rear* variable is incremented.
- If the queue is not full and the value of *rear* variable is equal to MAX–1, then *rear* variable is set to 0.

This task is accomplished as shown in the following function

Listing 6.8

```
void Enqueue( queue *v, int value )
{
    if ( v->front == NIL )
        v->front = v->rear = 0;
```

```
    else if ( v->rear == (MAX-1) )
        v->rear = 0;
    else
        v->rear++;
    v->elements[v->rear] = value;
}
```

6.4.1.2.3 Dequeue Operation on a Circular Queue

The front element of a circular queue is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the front element of a circular queue, the value of the *front* variable is modified so that it points to the new front.

There are two possibilities:

- If there was only one element in a circular queue, then after dequeue operation the circular queue will become empty. This state of a circular queue is reflected by setting *front* and *rear* variables to sentinel value NIL.
- If value of the front variable is equal to MAX–1, then set *front* variable to 0.

If none of the above conditions hold, then the *front* variable is incremented.

This task is accomplished as shown in the following function.

Listing 6.9

```
int Dequeue( queue *v )
{
    int temp;
    temp = v->elements[v->front];
    if ( v->front == v->rear )
        v->front = v->rear = -1;
    else if ( v->front == (MAX-1))
        v->front = 0;
    else
        v->front++;
    return temp;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
#define MAX 5
typedef struct
{
   int top;
   int ele[MAX];
}stack;
 stack s;
void push();
void pop();
void disp();

```

```c

void main()
{
  int ch;
  s.top = -1;
  while(1)
  {
  printf("\n----Options Are----");
  printf("\n1-Push");
  printf("\n2-Pop");
  printf("\n3-Display");
  printf("\n4-Exit");
  printf("\nEnter your choice: ");
  scanf("%d",&ch);
  switch(ch)
  {
    case 1: push();
            break;
    case 2: pop();
            break;
    case 3: disp();
            break;
    case 4: exit(0);
    default: printf("\n Wrong Choice");
  }
 }
}
```

```
40
41  void push()
42  {
43      int e;
44      if(s.top==MAX-1)
45        printf("\nStack is Full");
46      else
47      {
48          printf("\nEnter element to push");
49           scanf("%d",&e);
50
51           s.top++;
52           s.ele[s.top]=e;
53       }
54  }
55
```

```
55
56  void pop()
57  {
58      int e;
59      if(s.top==-1)
60        printf("\nStack is empty");
61      else
62      {
63          e=s.ele[s.top];
64          printf("\nDeleted element is %d",e);
65          s.top--;
66      }
67  }
68
```

```
68
69  void disp()
70  {
71      int top1;
72      top1=s.top;
73      if(s.top==-1)
74        printf("\nStack is empty");
75      else
76      {
77        while(top1!=-1)
78        { printf("\n%d",s.ele[top1]);
79           top1--;
80        }
81      }
82  }
83
```

5.3.2 Representing a Stack using a Linked List

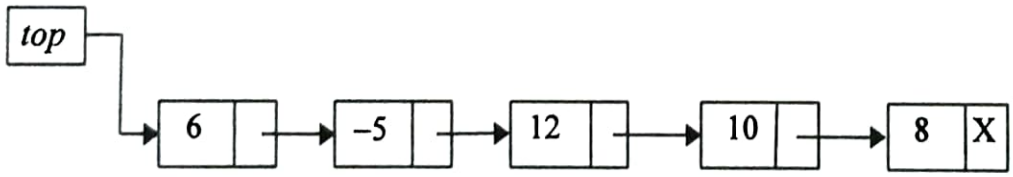
A stack represented using a linked list is also known as *linked stack*. The array based representation of stacks suffers from following limitations

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

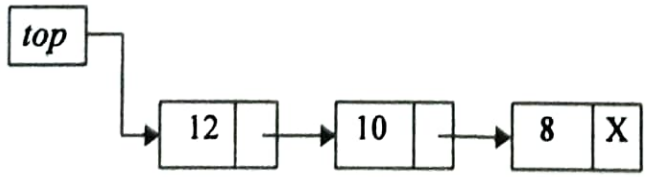
The linked list representation allow a stack to grow to a limit of the computer's memory. The following are the necessary declarations

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

stack *top;
```

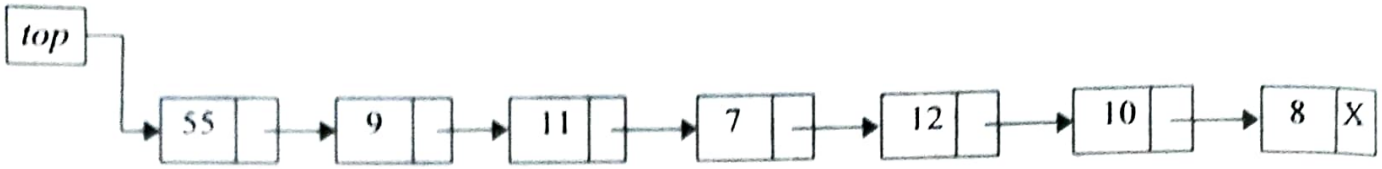
We have defined our own data type named *stack*, which is a self-referential structure and whose first element *info* hold the element of the stack and the second element *next* holds the address of the element under it in the stack. The last line declares a pointer variable *top* of type *stack*.



(*a*) Representation of stack of Figure 5.1(*a*) in memory



(*b*) Representation of stack of Figure 5.1(*b*) in memory



(*c*) Representation of stack of Figure 5.1(*c*) in memory

Figure 5.3 Linked representation of stack of Figure 5.1

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

5.3.2.1 Creating an Empty Stack

Before we can use a stack, it is to be initialized. To initialize a stack, we will create an empty linked list. The empty linked list is created by setting pointer variable *top* to value NULL.

This simple task can be accomplished by the following function.

Listing 5.12

```
void CreateStack( stack **top )
{
    *top = ( stack * ) NULL;
}
```

5.3.2.2 Testing Stack 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 5.13

```
boolean IsEmpty( stack *top )
{
   if ( top == ( stack * ) 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 5.14

```
boolean IsEmpty( stack *top )
{
   return ( top == ( stack * ) NULL ? true : false );
}
```

5.3.2.3 Testing Stack for Overflow

Since a stack represented using a linked list can grow to a limit of a computer's memory, there overflow condition never occurs. Hence, this operations is not implemented for linked stacks.

5.3.2.4 Push Operation

To push a new element onto the stack, the element is inserted in the beginning of the linked list. This task is accomplished as shown in the following function.

Listing 5.15

```
void Push( stack **top, int value )
{
   stack *ptr;
   ptr = ( stack * ) malloc ( sizeof ( stack ) );
   ptr->info = value;
   ptr->next = *top;
   *top = ptr;
}
```

5.3.2.5 Pop Operation

To pop an element from the stack, the element is removed from the beginning of the linked list. This task is accomplished as shown in the following function.

Listing 5.16

```
int Pop( stack **top )
{
   int temp;
   stack *ptr;
```

```
temp = *top->data;
ptr = *top
*top = *top->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.

5.3.2.6 Accessing Top Element

The top element is accessed as shown in the following function.

Listing 5.17

```
int Peek( stack *top )
{
   return ( top->data );
}
```

5.3.2.7 Dispose a Stack

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

Listing 5.18

```
void DisposeStack( stack **top )
{
   stack *ptr;
   while ( *top != stack * NULL )
   {
      ptr = *top;
      *top = (*top)->next;
      free ( ptr );
   }
}
```

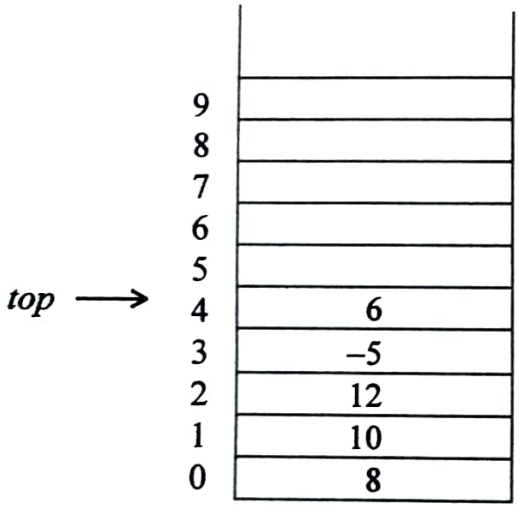
This dispose operations is O(n) time.

5.1 INTRODUCTION

A *stack* is one of the most commonly used data structure. A *stack*, also called a Last-In-First-Out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the *top*. This structure operates in much the same way as stack of trays. If we want to place another tray, it can be placed only at the top. Similarly, if we want to remove a tray from stack of trays, it can only be removed from the top. The insertion and deletion operations in stack terminology are known as *push* and *pop* operations.

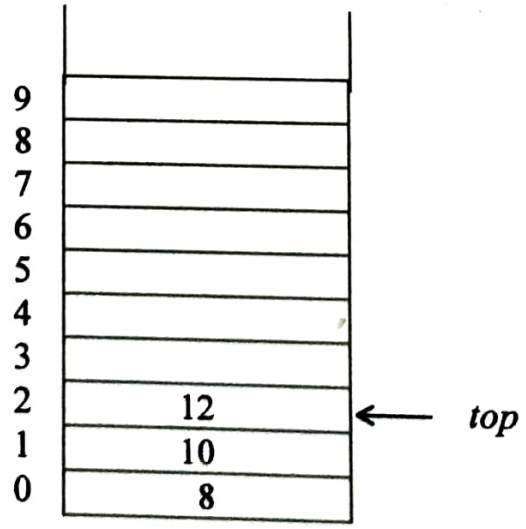
Example 5.1

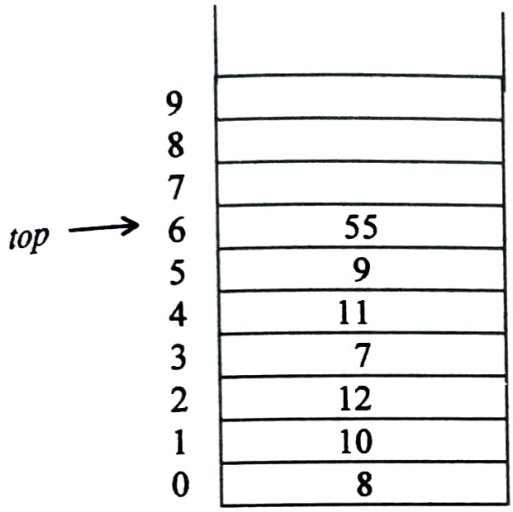
The following figure illustrates a stack which can accommodate maximum of 10 elements.



(*a*) Stack after pushing elements 8, 10, 12, –5, 6 in turn

(*b*) Stack after poping top two elements i.e. 6, –5 in turn





(*c*) Stack after pushing elements 7, 11, 9, 55 in turn

Figure 5.1 Illustration of a stack whose elements are integers values

5.2 OPERATIONS ON STACKS

The following operations are performed on stacks.

1. **CreateStack(*s*)** – to create *s* as an empty stack.
2. **Push(*s*, *i*)** – to push element *i* onto stack *s*.
3. **Pop(*s*)** – to access and remove the top element of the stack *s*.
4. **Peek(*s*)** – to access the top element of the stack *s* without removing it from the stack *s*.
5. **IsFull(*s*)** – to check whether the stack *s* is full.
6. **IsEmpty(*s*)** – to check whether the stack *s* is empty.

All of these operation runs in O(1) time.

Note that if the stack *s* is empty then it is not possible to pop the stack *s*. Similarly, as there is no element in the stack, the *top(s)* operations is also not valid. Therefore, we must ensure that the stack is not empty before attempting to perform these operations.

Likewise, if the stack *s* is full then it is not possible to push a new element on the stack *s*. Therefore, we must ensure that the stack is not full before attempting to perform push operation.

5.3 REPRESENTATION OF A STACK IN MEMORY

A stack can be represented in memory using a linear array or a linear linked list. Let us first discuss array representation of stacks.

5.3.1 Representing a Stack using an Array

To implement a stack we need a variable, called *top*, that holds the index of the top element of the stack and an array to hold the elements of the stack. Let us suppose that the elements of the stack are of integer type, and the stack can store maximum of 10 such elements i.e. stack size is 10.

The following are the necessary declarations

```
#define  MAX  10
typedef struct
{
   int top;
   int elements[MAX];
} stack;
stack s;
```

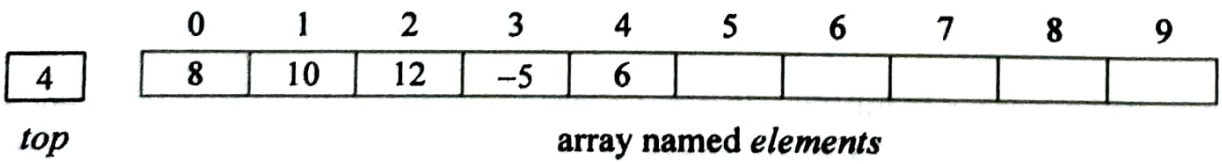
We have defined our own data type named *stack*, which is structure and whose first element *top* will be used as an index to the top element, and an array *elements* of size MAX whose elements are of integer type, to hold the elements of the stack. The last line declares variable *s* of type *stack*. With these declarations, we will write develop functions for various operation to be performed on the stack.

In addition to above declarations, we will use the declaration

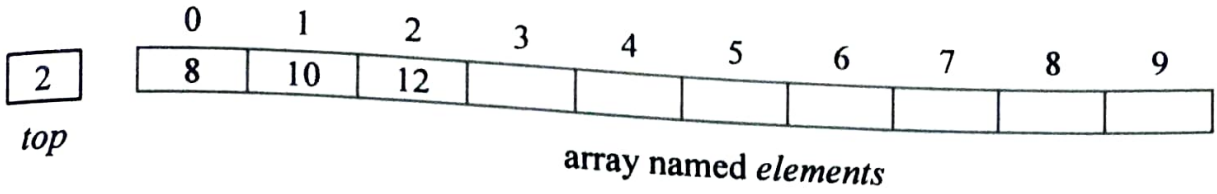
```
typedef enum { false, true } boolean;
```

to signify the use of *boolean* kind of data and/or operations. This statement defined new data type named *boolean* which can take value *false* or *true*.

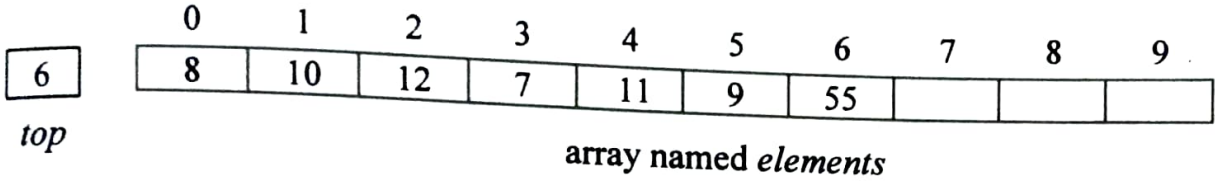
With these declarations, stack of Figure 5.1 will be represented in computer memory as



(*a*) Representation of stack of Figure 5.1(*a*) in memory



(*b*) Representation of stack of Figure 5.1(*b*) in memory



(*c*) Representation of stack of Figure 5.1(*c*) in memory

Figure 5.2 Array representation of stack of Figure 5.1

5.3.1.1 Creating an Empty Stack

Before we can use a stack, it is to be initialized. As the index of array elements can take any value in the range 0 to MAX–1, the purpose of initializing the stack is served by assigning value –1 (sentinel value) to the *top* variable. This simple task can be accomplished by the following function.

Listing 5.1

```
void CreateStack( stack *ps )
{
   ps->top = -1;
}
```

5.3.1.2 Testing Stack for Underflow

Before we remove an item from a stack, it is necessary to test whether stack still have some elements i.e. to test that whether the stack is empty or not. If it is not empty then the pop operation can be performed to remove the top element. This test is performed by comparing the value of *top* with sentinel value –1 as shown in the following function.

Listing 5.2

```
boolean IsEmpty( stack *ps )
{
   if ( ps->top == -1  )
     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 5.3

```
boolean IsEmpty( stack *ps )
{
    return  ( ( ps->top == -1 ) ? true : false );
}
```

5.3.1.3 Testing Stack for Overflow

Before we insert new item onto the stack, it is necessary to test whether stack still have some space to accommodate the incoming element i.e. to test that whether the stack is full or not. If it is not full then the *push* operation can be performed to insert the element at top of the stack. This test is performed by comparing the value of the *top* with value MAX–1, the largest index value that the *top* can take as shown in the following function.

Listing 5.4

```
boolean IsFull( stack *ps )
{
   if ( ps->top == ( MAX -1) )
      return true;
   else
      return false;
}
```

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

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

Listing 5.5

```
boolean IsFull( stack *ps )
{
   return  (( ps->top == MAX - 1 ) ? true : false );
}
```

5.3.1.4 Push Operation

Before the push operation, if the stack is empty, then the value of the *top* will be –1 (sentinel value) and if the stack is not empty then the value of the *top* will be the index of the element currently on the top. Therefore, before we place value onto the stack, the value of the *top* is incremented so that it points to the new top of stack, where incoming element is placed. This task is accomplished as shown in the following function.

Listing 5.6

```
void Push( stack *ps, int value )
{
   ps->top++;
   ps->elements [ps->top] = value;
}
```

The above function can also be written as

Listing 5.7

```
void Push( stack *ps, int value )
{
   ps->elements[++ps->top] = value;
}
```

5.3.1.5 Pop Operation

The element on the top of stack is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the top element to a local variable, the variable *top* is decremented so that it points to the new top. This task is accomplished as shown in the following function.

Listing 5.8

```
int Pop( stack *ps )
{
   int temp;
   temp = ps->elements[ps->top];
   ps->top--;
   return temp;
}
```

The above function can also be written as

Listing 5.9

```
int Pop( stack *ps )
{
   return ( ps->elements[ps->top--] );
}
```

In the above two versions of the *pop()* function, the top element is returned via the *return* statement.

The top element can also be returned via the argument list as shown below:

Listing 5.10

```
void Pop( stack *ps, int *item )
{
   *item =  ps->elements[ps->top--];
}
```

5.3.1.6 Accessing Top Element

There may be instances where we want to access the top element of the stack without removing it from the stack (i.e. without poping it). This task is accomplished as shown in the following function

Listing 5.11

```
int Peek( stack *ps )
{
   return ( ps->elements[ps->top] );
}
```