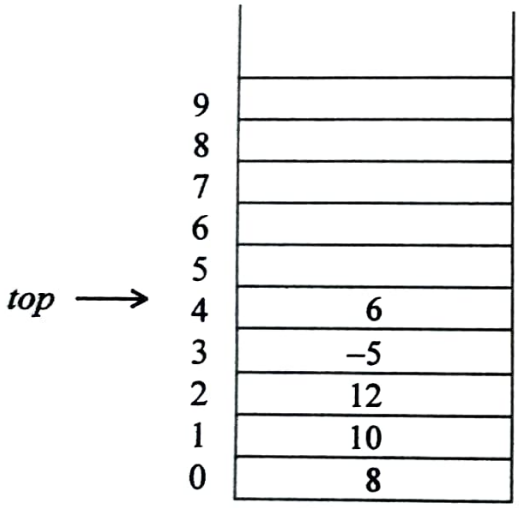
5.1 INTRODUCTION

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.

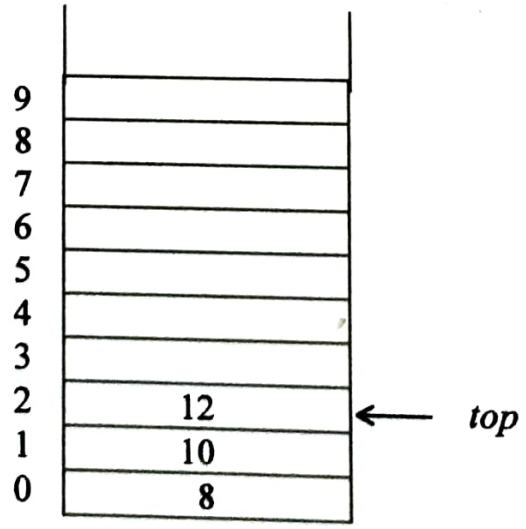
Example 5.1

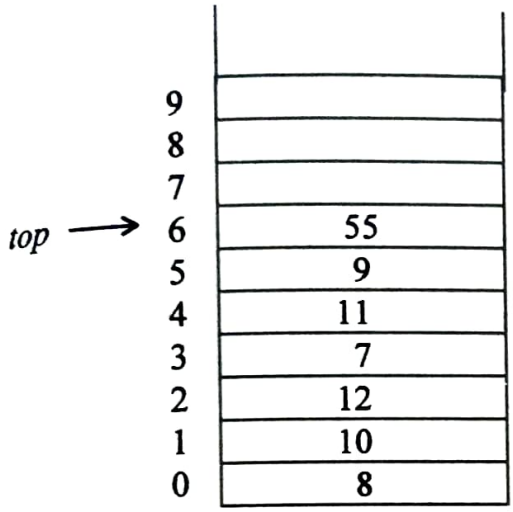
The following figure illustrates a stack which can accommodate maximum of 10 elements.



(*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;
```

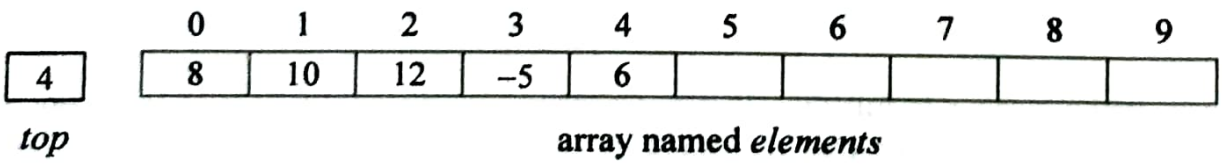
We have defined our own data type named *stack*, which is structure and whose first element *top* will be used as an index to the top element, and an array *elements* of size MAX whose elements are of integer type, to hold the elements of the stack. The last line declares variable *s* of type *stack*. 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

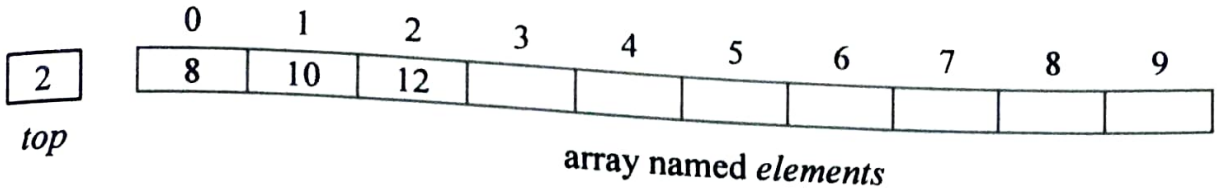
```
typedef enum { false, true } boolean;
```

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*.

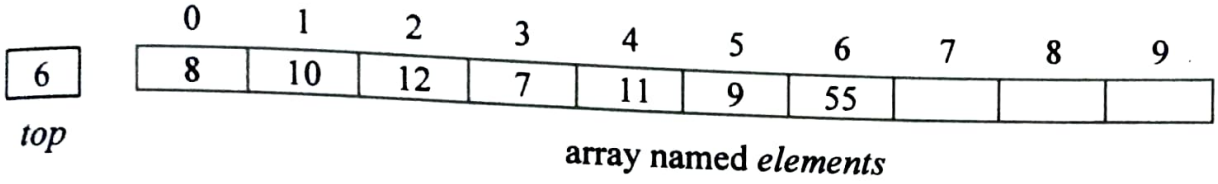
With these declarations, stack of Figure 5.1 will be represented in computer memory as



(*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] );
}
```