Chapte 3

Stacks

3.1 INTRODUCTION

One of the simplest, useful and frequently used data structure for problem solving is the **stack**. This chapter introduces such a data structure starting with the definition and the various operations of stack. The implementation of stacks can be done either using linear array structure or using list structures (with pointers). We will study the first one in this chapter and the second in Chapter 6.
Since, stacks work on **Last-In-First-Out (LIFO)** policy and their applications include,

(i) Checking for parenthesis balancing in expressions.
(ii) Postfix expression evaluation.
(iii) Infix-to-Postfix expression conversion.
(iv) Infix-to-Prefix expression conversion.

The algorithm and 'C' implementation for all the above applications will be developed in this chapter.

3.2 DEFINITION

A **stack** is defined as a linear list in which insertions and deletions take place at the same end. This end is called as the **top** of the stack and the other end is called as the **bottom** of the stack.

According to the definition, new items may be inserted from top and the elements are to be deleted from again top. This suggests that the elements inserted most recently can only be deleted or removed. We can call such an operation as the last inserted element comes out first. Hence, a stack works as **Last In First Out (LIFO)** manner.

Figure 3.1 shows a stack with five character type elements and these elements are inserted in the order A, B, C, D and E.

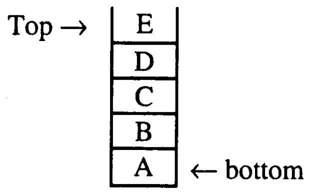


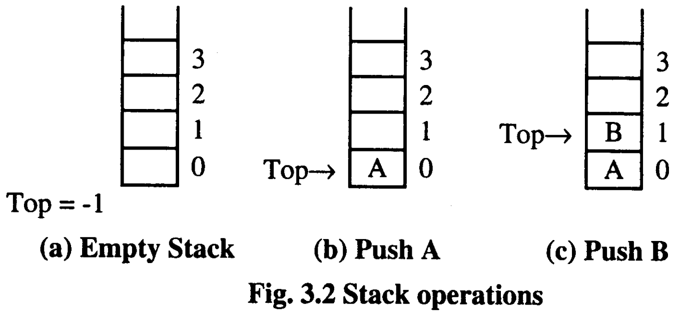
Fig.3.1 A Stack

The element E is at the top and element A is at the bottom. We shall also assume that the stack elements are stored in an array with a maximum capacity or size. In fact, the concept of stack is not new. Take for example the cupboard in your bedroom where clothes are stacked (kept one over the other) or books stacked on your table, files stacked on a clerk's table, etc are some of the common items that are stacked.

3.2.1 Basic operations of Stack – Push and Pop

Having defined the stack data structure, this section describes two important operations performed on a stack. We shall recall the definition of stack as that the elements are generally inserted and deleted. These are the two important operations we need to do on a stack and formally we call them as **PUSH** and **POP**. The push operation inserts an element into a stack and pop operation removes an element from the top of the stack. However, notice that only the top most element can be removed and one can not access the other elements in the stack. Hence, to push and pop from the top of the stack, a pointer is maintained called as **top**. Any insertion or deletion should be based upon the value of *top*.

Assuming a C like array to hold the stack elements, Figure 3.2 shows how a stack grows and shrinks because of *push* and *pop*. When a stack does not contain any elements such a state is called as **empty stack**.



(a) Empty Stack **(b) Push A** **(c) Push B**

Fig. 3.2 Stack operations

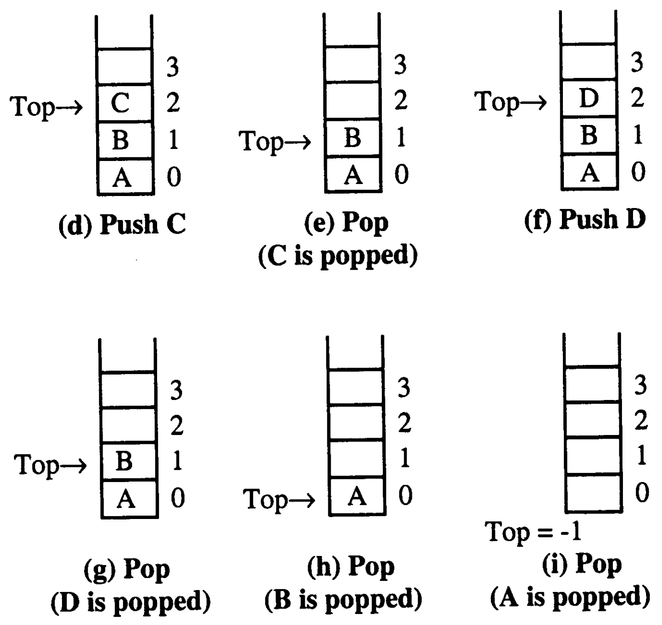


Fig 3.2 Stack operations (contd)

Initially the stack is empty and is shown in Figure 3.2(a). Figures 3.2(b), (c) and (d) show how *Top* is updated and stack grows upwards with elements A, B and C. Now, the element C is at the *Top*. Figure 3.2(e) shows, when a *pop* operation is performed, element C is removed from stack and *Top* points to B. This suggests that *Top* always points to the filled, top most element. We add D so that contents of stack is A, B and D (from bottom to top). Last three Figures (g), (h) and (i) show series of *pop* operations in which D, B and A are all removed and finally the stack is empty.

An ADT specification can now be written for a stack and is shown in Figure 3.3.

```
ADT Stack {
    specification:
         Linear list of elements. Top is a pointer to
         point to the top of the stack.
    Operations:
         Push(x) - Insert an element to the stack
                   pointed by Top
         Pop() - Removes an element pointed by Top.
         IsFull() - Returns true when the stack is full.
         IsEmpty() - Returns true when the stack is
                     empty.
}
```

Fig. 3.3 ADT for a Stack

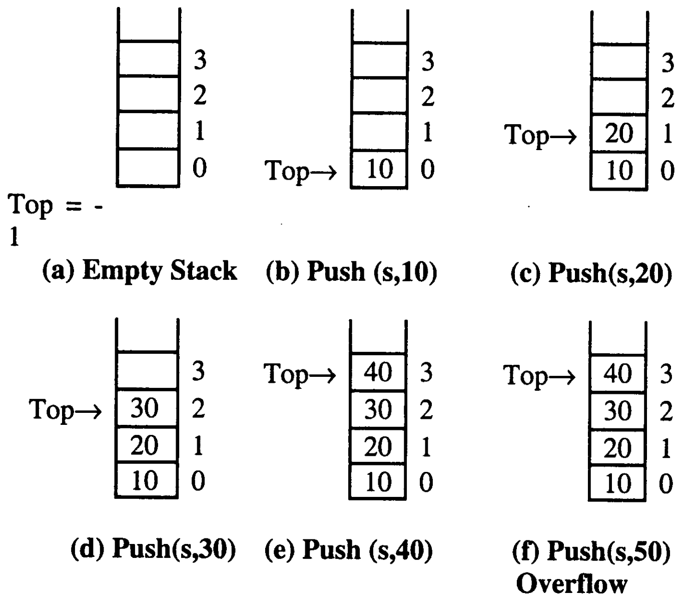


Fig. 3.4 Push() snapshot.

3.3.3 Implementing Pop() Function

An element may be popped (or removed) only when stack is not empty. Therefore, we first check stack status (whether stack is empty or not) using a function `Empty()`. When the stack is not empty, simply retrieve the element pointed by `top` and store it in a temporary variable called `temp`. This is shown in Program 3.2.

Program 3.2
Pop fucntion

```
int Pop (Stack ps)
{
    int temp;
    if (Empty(ps))   /* underflow */
        return(-1);

    temp = ps->items[ps->top]; /* retrieve the element */
    --(ps->top);
    return(temp);
}
```