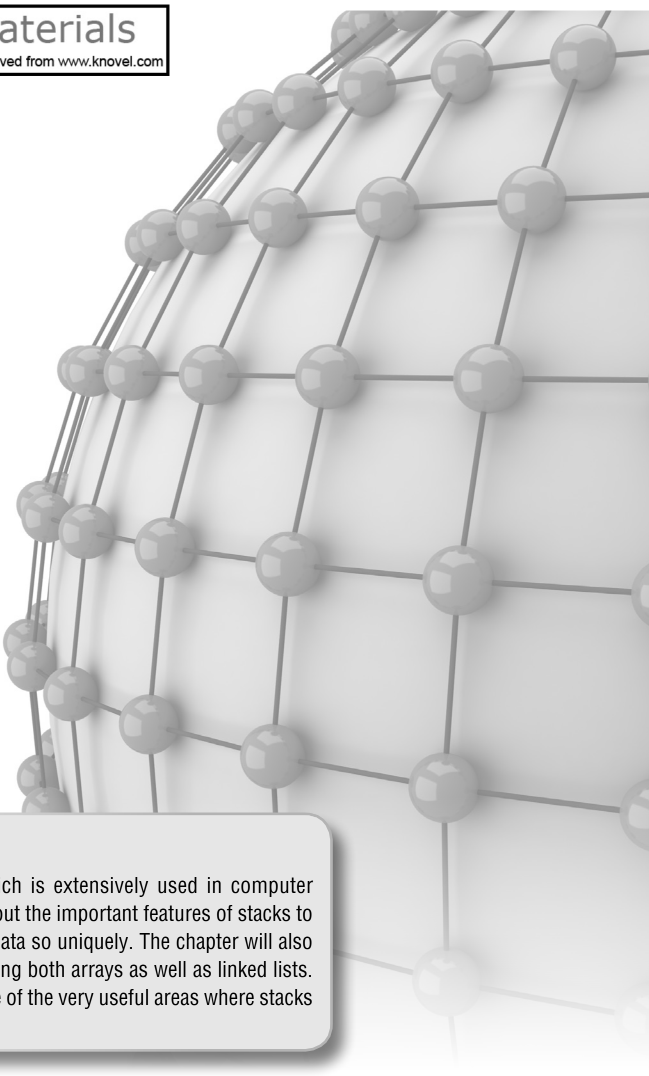

CHAPTER 7

Stacks

LEARNING OBJECTIVE

A stack is an important data structure which is extensively used in computer applications. In this chapter we will study about the important features of stacks to understand how and why they organize the data so uniquely. The chapter will also illustrate the implementation of stacks by using both arrays as well as linked lists. Finally, the chapter will discuss in detail some of the very useful areas where stacks are primarily used.

7.1 INTRODUCTION

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. 7.1. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

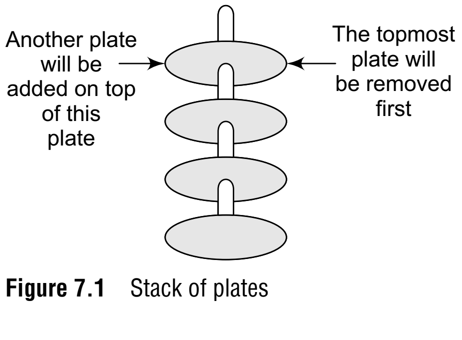
Another plate will be added on top of this plate
The topmost plate will be removed first

Figure 7.1 Stack of plates

Now the question is where do we need stacks in computer science? The answer is in function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.

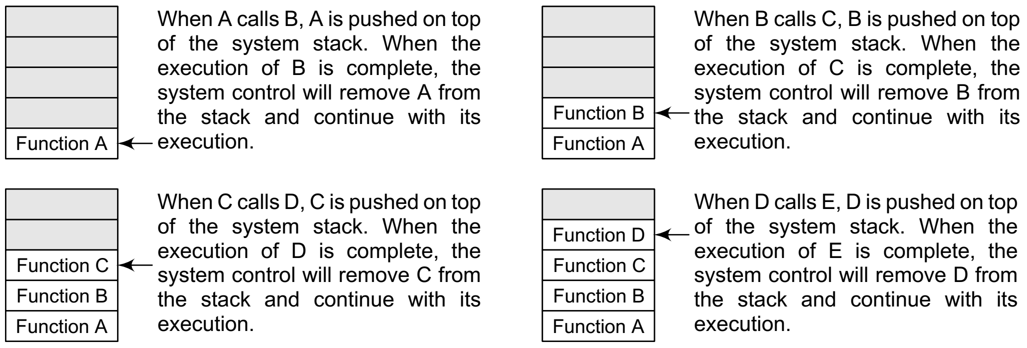
Function A
When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.
Function B
Function A
When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.
Function C
Function B
Function A
When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.
Function D
Function C
Function B
Function A
When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.

Figure 7.2 System stack in the case of function calls

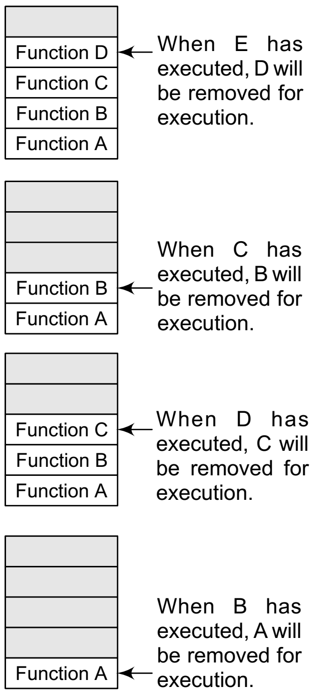
Function D
Function C
Function B
Function A
When E has executed, D will be removed for execution.
Function B
Function A
When C has executed, B will be removed for execution.
Function C
Function B
Function A
When D has executed, C will be removed for execution.
Function A
When B has executed, A will be removed for execution.

Figure 7.3 System stack when a called function returns to the calling function

In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used. Whenever a function calls another function, the calling function is pushed onto the top of the stack. This is because after the called function gets executed, the control is passed back to the calling function. Look at Fig. 7.2 which shows this concept.

Now when function `E` is executed, function `D` will be removed from the top of the stack and executed. Once function `D` gets completely executed, function `C` will be removed from the stack for execution. The whole procedure will be repeated until all the functions get executed. Let us look at the stack after each function is executed. This is shown in Fig. 7.3.

The system stack ensures a proper execution order of functions. Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.

Stacks can be implemented using either arrays or linked lists. In the following sections, we will discuss both array and linked list implementation of stacks.

7.2 ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called `TOP` associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called `MAX`, which is used to store the maximum number of elements that the stack can hold.

If `TOP = NULL`, then it indicates that the stack is empty and if `TOP = MAX-1`, then the stack is full. (You must be wondering why we have written `MAX-1`. It is because array indices start from 0.) Look at Fig. 7.4.

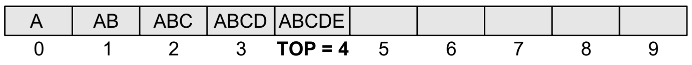

Figure 7.4 Stack

The stack in Fig. 7.4 shows that `TOP = 4`, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

7.3 OPERATIONS ON A STACK

A stack supports three basic operations: `push`, `pop`, and `peek`. The `push` operation adds an element to the top of the stack and the `pop` operation removes the element from the top of the stack. The `peek` operation returns the value of the topmost element of the stack.

7.3.1 Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if `TOP=MAX–1`, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an `OVERFLOW` message is printed. Consider the stack given in Fig. 7.5.

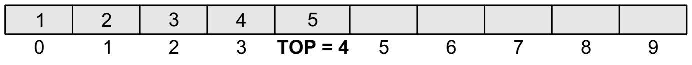

Figure 7.5 Stack

To insert an element with value 6, we first check if `TOP=MAX–1`. If the condition is false, then we increment the value of `TOP` and store the new element at the position given by `stack[TOP]`. Thus, the updated stack becomes as shown in Fig. 7.6.

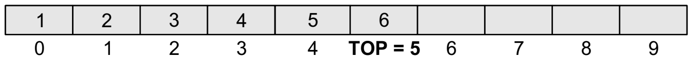

Figure 7.6 Stack after insertion

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Figure 7.7 Algorithm to insert an element in a stack

Figure 7.7 shows the algorithm to insert an element in a stack. In Step 1, we first check for the `OVERFLOW` condition. In Step 2, `TOP` is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by `TOP`.

7.3.2 Pop Operation

The `pop` operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if `TOP=NULL` because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an `UNDERFLOW` message is printed. Consider the stack given in Fig. 7.8.

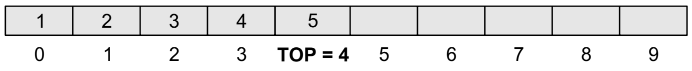

Figure 7.8 Stack

To delete the topmost element, we first check if `TOP=NULL`. If the condition is false, then we decrement the value pointed by `TOP`. Thus, the updated stack becomes as shown in Fig. 7.9.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Figure 7.10 Algorithm to delete an element from a stack

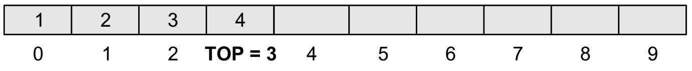

Figure 7.9 Stack after deletion

Figure 7.10 shows the algorithm to delete an element from a stack. In Step 1, we first check for the `UNDERFLOW` condition. In Step 2, the value of the location in the stack pointed by `TOP` is stored in `VAL`. In Step 3, `TOP` is decremented.

7.3.3 Peek Operation

`Peek` is an operation that returns the value of the topmost element of the stack without deleting it from the stack. The algorithm for `Peek` operation is given in Fig. 7.11.

However, the `Peek` operation first checks if the stack is empty, i.e., if `TOP = NULL`, then an appropriate message is printed, else the value is returned. Consider the stack given in Fig. 7.12.

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Figure 7.11 Algorithm for `Peek` operation

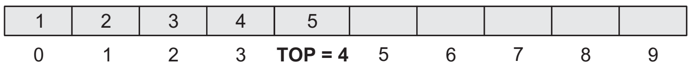

Figure 7.12 Stack

Here, the `Peek` operation will return 5, as it is the value of the topmost element of the stack.

PROGRAMMING EXAMPLE

1. Write a program to perform Push, Pop, and Peek operations on a stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 // Altering this value changes size of stack created

int st[MAX], top=-1;
void push(int st[], int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);

int main(int argc, char *argv[]) {
        int val, option;
        do
        {
                printf("\n *****MAIN MENU*****");
                printf("\n 1. PUSH");
                printf("\n 2. POP");
                printf("\n 3. PEEK");
                printf("\n 4. DISPLAY");
                printf("\n 5. EXIT");
                printf("\n Enter your option: ");
                scanf("%d", &option);
                switch(option)
                {
                case 1:
                        printf("\n Enter the number to be pushed on stack: ");
                        scanf("%d", &val);
                        push(st, val);
                        break;
                case 2:
                        val = pop(st);
                        if(val != -1)
                        printf("\n The value deleted from stack is: %d", val);
                        break;
                case 3:
                        val = peek(st);
                        if(val != -1)
                        printf("\n The value stored at top of stack is: %d", val);
                        break;
                case 4:
                        display(st);
                        break;
                }
        }while(option != 5);
        return 0;
}
void push(int st[], int val)
{
        if(top == MAX-1)
        {
                printf("\n STACK OVERFLOW");
        }
        else
        {
                top++;
                st[top] = val;
        }
}
int pop(int st[])
{
        int val;
        if(top == -1)
        {
                printf("\n STACK UNDERFLOW");
                return -1;
        }
        else
        {
                val = st[top];
                top--;
                return val;
        }
}
void display(int st[])
{
        int i;
        if(top == -1)
        printf("\n STACK IS EMPTY");
        else
        {
                for(i=top;i>=0;i--)
                printf("\n %d",st[i]);
                printf("\n"); // Added for formatting purposes
        }
}
int peek(int st[])
{
        if(top == -1)
        {
                printf("\n STACK IS EMPTY");
                return -1;
        }
        else
        return (st[top]);
}
```

Output
```
*****MAIN MENU*****
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 500
```

7.4 LINKED REPRESENTATION OF STACKS

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

The storage requirement of linked representation of the stack with `n` elements is `O(n)`, and the typical time requirement for the operations is `O(1)`.

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The `START` pointer of the linked list is used as `TOP`. All insertions and deletions are done at the node pointed by `TOP`. If `TOP = NULL`, then it indicates that the stack is empty.

The linked representation of a stack is shown in Fig. 7.13.

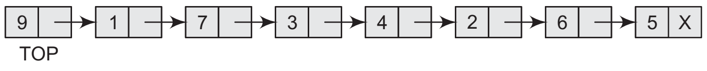

Figure 7.13 Linked stack

7.5 OPERATIONS ON A LINKED STACK

A linked stack supports all the three stack operations, that is, `push`, `pop`, and `peek`.

7.5.1 Push Operation

The `push` operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 7.14.

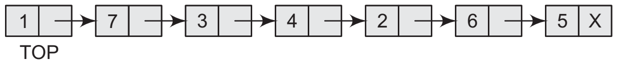

Figure 7.14 Linked stack

To insert an element with value 9, we first check if `TOP=NULL`. If this is the case, then we allocate memory for a new node, store the value in its `DATA` part and `NULL` in its `NEXT` part. The new node will then be called `TOP`. However, if `TOP!=NULL`, then we insert the new node at the beginning of the linked stack and name this new node as `TOP`. Thus, the updated stack becomes as shown in Fig. 7.15.

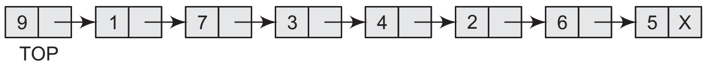

Figure 7.15 Linked stack after inserting a new node

Figure 7.16 shows the algorithm to push an element into a linked stack. In Step 1, memory is allocated for the new node. In Step 2, the `DATA` part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. This

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Figure 7.16 Algorithm to insert an element in a linked stack

is done by checking if `TOP = NULL`. In case the `IF` statement evaluates to true, then `NULL` is stored in the `NEXT` part of the node and the new node is called `TOP`. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the `TOP` node) and termed as `TOP`.

7.5.2 Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if `TOP=NULL`, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an `UNDERFLOW` message is printed. Consider the stack shown in Fig. 7.17.

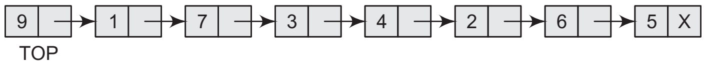

Figure 7.17 Linked stack

In case `TOP!=NULL`, then we will delete the node pointed by `TOP`, and make `TOP` point to the second element of the linked stack. Thus, the updated stack becomes as shown in Fig. 7.18.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

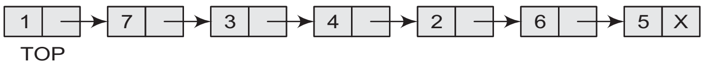

Figure 7.18 Linked stack after deletion of the topmost element

Figure 7.19 shows the algorithm to delete an element from a stack. In Step 1, we first check for the `UNDERFLOW` condition. In Step 2, we use a pointer `PTR` that points to `TOP`. In Step 3, `TOP` is made to point to the next node in sequence. In Step 4, the memory occupied by `PTR` is given back to the free pool.

Programming Example

2. Write a program to implement a linked stack.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
struct stack
{
        int data;
        struct stack *next;
};
struct stack *top = NULL;
struct stack *push(struct stack *, int);
struct stack *display(struct stack *);
struct stack *pop(struct stack *);
int peek(struct stack *);

int main(int argc, char *argv[]) {
        int val, option;
        do
        {
                printf("\n *****MAIN MENU*****");
                printf("\n 1. PUSH");
                printf("\n 2. POP");
                printf("\n 3. PEEK");
                printf("\n 4. DISPLAY");
                printf("\n 5. EXIT");
                printf("\n Enter your option: ");
                scanf("%d", &option);
                switch(option)
                {
                case 1:
                        printf("\n Enter the number to be pushed on stack: ");
                        scanf("%d", &val);
                        top = push(top, val);
                        break;
                case 2:
                        top = pop(top);
                        break;
                case 3:
                        val = peek(top);
                        if (val != -1)
                        printf("\n The value at the top of stack is: %d", val);
                        else
                        printf("\n STACK IS EMPTY");
                        break;
                case 4:
                        top = display(top);
                        break;
                }
        }while(option != 5);
        return 0;
}
struct stack *push(struct stack *top, int val)
{
        struct stack *ptr;
        ptr = (struct stack*)malloc(sizeof(struct stack));
        ptr -> data = val;
        if(top == NULL)
        {
                ptr -> next = NULL;
                top = ptr;
        }
        else
        {
                ptr -> next = top;
                top = ptr;
        }
        return top;
}
struct stack *display(struct stack *top)
{
        struct stack *ptr;
        ptr = top;
        if(top == NULL)
        printf("\n STACK IS EMPTY");
        else
        {
```

```
                    while(ptr != NULL)
                    {
                            printf("\n %d", ptr -> data);
                            ptr = ptr -> next;
                    }
            }
            return top;
    }
    struct stack *pop(struct stack *top)
    {
            struct stack *ptr;
            ptr = top;
            if(top == NULL)
            printf("\n STACK UNDERFLOW");
            else
            {
                    top = top -> next;
                    printf("\n The value being deleted is: %d", ptr -> data);
                    free(ptr);
            }
            return top;
    }
    int peek(struct stack *top)
    {
            if(top==NULL)
            return -1;
            else
            return top ->data;
    }
```

Output

```
    *****MAIN MENU*****
    1. PUSH
    2. POP
    3. Peek
    4. DISPLAY
    5. EXIT
    Enter your option : 1
    Enter the number to be pushed on stack : 100
```

7.6 MULTIPLE STACKS

While implementing a stack using an array, we had seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent `OVERFLOW` conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory. Thus, there lies a trade-off between the frequency of overflows and the space allocated.

So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size. Figure 7.20 illustrates this concept.

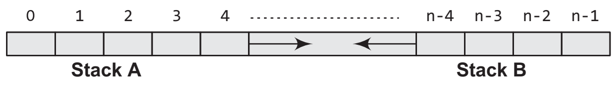

Figure 7.20 Multiple stacks

In Fig. 7.20, an array `STACK[n]` is used to represent two stacks, `Stack A` and `Stack B`. The value of `n` is such that the combined size of both the stacks will never exceed `n`. While operating on