UNIT 4 STACKS

Structure **Page Nos.**

4.0 INTRODUCTION

One of the most useful concepts in computer science is stack. In this unit, we shall examine this simple data structure and see why it plays such a prominent role in the area of programming. There are certain situations when we can insert or remove an item only at the beginning or the end of the list.

A stack is a linear structure in which items may be inserted or removed only at one end called the *top of the stack*. A stack may be seen in our daily life, for example, *Figure 4.1* depicts a stack of dishes. We can observe that any dish may

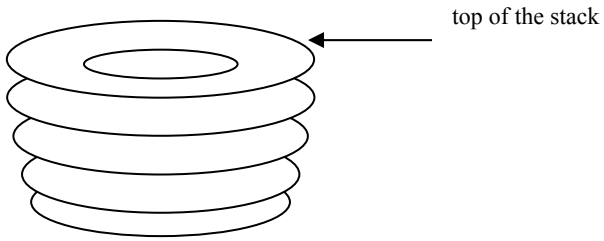


Figure 4.1: A stack of dishes

be added or removed only from the top of the stack. It concludes that the item added last will be the item removed first. Therefore, stacks are also called LIFO (Last In First Out) or FILO (First In Last Out) lists. We also call these lists as "piles" or "push-down list".

Generally, two operations are associated with the stacks named Push & Pop.

- *Push* is an operation used to insert an element at the top.
- *Pop* is an operation used to delete an element from the top

Example 4.1

Now we see the effects of push and pop operations on to an empty stack. *Figure 4.2(a)* shows (i) an empty stack; (ii) a list of the elements to be inserted on to stack;

and (iii) a variable top which helps us keep track of the location at which insertion or removal of the item would occur.

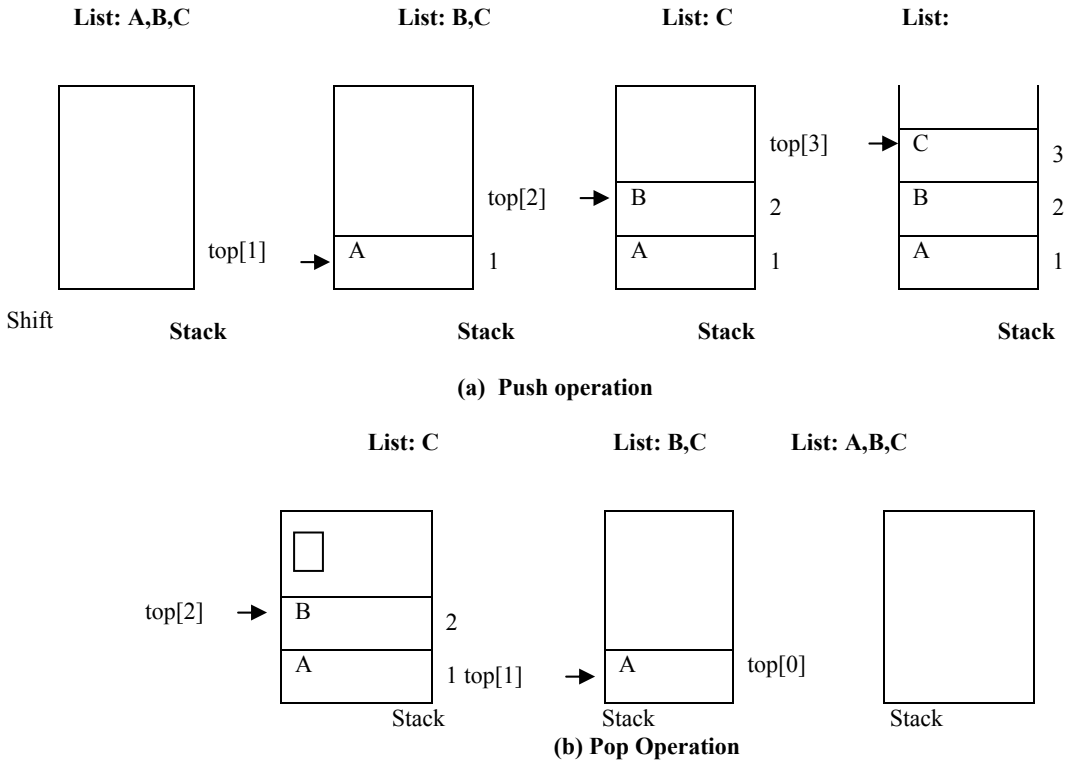


Figure 4.2: Demonstration of (a) Push operation, (b) Pop operation

Initially in *Figure 4.2(a)*, top contains 0, implies that the stack is empty. The list contains three elements, A, B &C. In *Figure 4.2(b),* we remove an element A from the list of elements, push it on to stack. The value of top becomes 1, pointing to the location of the stack at which A is stored.

Similarly, we remove the elements B & C from the list one by one and push them on to the stack. Accordingly, the value of the **top** is incremented. *Figure 4.2(a)* explains the pushing of B and C on to stack. The **top** now contains value 3 and pointing to the location of the last inserted element C.

On the other hand, *Figure 4.2(b)* explains the working of pop operation. Since, only the top element can be removed from the stack, in *Figure 4.2(b),* we remove the top element C (we have no other choice). C goes to the list of elements and the value of the top is decremented by 1. The top now contains value 2, pointing to B (the top element of the stack). Similarly, in *Figure 4.2(b),* we remove the elements B and A from the stack one by one and add them to the list of elements. The value of top is decremented accordingly.

There is no upper limit on the number of items that may be kept in a stack. However, if a stack contains a single item and the stack is popped, the resulting stack is called empty stack. The pop operation cannot be applied to such stacks as there is no element to pop, whereas the push operation can be applied to any stack.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the concept of stack;
- implement the stack using arrays;
- implement the stack using linked lists;
- implement multiple stacks, and
- give some applications of stack.

4.2 ABSTRACT DATA TYPE-STACK

Conceptually, the **stack** abstract data type mimics the information kept in a pile on a desk. Informally, we first consider materials on a desk, where we may keep separate stacks for bills that need paying, magazines that we plan to read, and notes we have taken. We can perform several operations that involve a stack:

- start a new stack;
- place new information on the top of a stack;
- take the top item off of the stack;
- read the item on the top; and
- determine whether a stack is empty. (There may be nothing at the spot where the stack should be).

When discussing these operations, it is conventional to call the addition of an item to the top of the stack as a **push operation** and the deletion of an item from the top as a **pop operation**. (These terms are derived from the working of a spring-loaded rack containing a stack of cafeteria trays. Such a rack is loaded by pushing the trays down on to the springs as each diner removes a tray, the lessened weight on the springs causes the stack to pop up slightly).

4.3 IMPLEMENTATION OF STACK

Before programming a problem solution that uses a stack, we must decide how to represent a stack using the data structures that exist in our programming language. Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Each approach has its advantages and disadvantages. A stack is generally implemented with two basic operations – push and pop. **Push** means to insert an item on to stack. The push algorithm is illustrated in *Figure 4.3(a)*. Here, **tos** is a pointer which denotes the position of top most item in the stack. Stack is represented by the array **arr** and **MAXSTACK** represents the maximum possible number of elements in the stack. The pop algorithm is illustrated in *Figure 4.3(b).*

```
Step 1:  [Check for stack overflow]
                    if tos >=MAXSTACK
                    print “Stack overflow” and exit
Step 2:  [Increment the pointer value by one]
                    tos=tos+1
Step 3:  [Insert the item]
                    arr[tos]=value
Step 4:  Exit
```

Figure 4.3(a): Algorithm to push an item onto the stack

The pop operation removes the topmost item from the stack. After removal of top most value **tos** is decremented by 1.

```
Step 1: [Check whether the stack is empty]
                    if tos = 0
                    print "Stack underflow" and exit

Step 2: [Remove the top most item]
                    value=arr[tos]
                    tos=tos-1

Step 3: [Return the item of the stack]
                    return(value)
```

Figure 4.3(b): Algorithm to pop an element from the stack

4.3.1 Implementation of Stack Using Arrays

A Stack contains an ordered list of elements and an array is also used to store ordered list of elements. Hence, it would be very easy to manage a stack using an array. However, the problem with an array is that we are required to declare the size of the array before using it in a program. Therefore, the size of stack would be fixed. Though an array and a stack are totally different data structures, an array can be used to store the elements of a stack. We can declare the array with a maximum size large enough to manage a stack. Program 4.1 implements a stack using an array.

```
#include<stdio.h>

int choice, stack[10], top, element;

void menu();
void push();
void pop();
void showelements();

void main()
{ choice=element=1;
  top=0;
  menu();
}

void menu()
{
        printf("Enter one of the following options:\n");
        printf("PUSH 1\n POP 2\n SHOW ELEMENTS 3\n  EXIT 4\n");
        scanf("%d", &choice);
        if (choice==1)
        {
         push(); menu();
        }
          if (choice==2)
        {
         pop();menu();
```

```
        }
          if (choice==3)
                                {
                                    showelements(); menu();
                                }
        }


void push()
{
        if (top<=9)
        {
                printf("Enter the element to be pushed to stack:\n");
                scanf("%d", &element);
                stack[top]=element;
                ++top;

        }
        else
        {
        printf("Stack is full\n");
        }
        return;
}

void pop()
{
        if (top>0)
        {
                --top;
                element = stack[top];
                printf("Popped element:%d\n", element);
         }
        else
        {
        printf("Stack is empty\n");
        }
        return;

}

void showelements()
{
        if (top<=0)
                printf("Stack is empty\n");

        else
                for(int i=0; i<top; ++i)
                        printf("%d\n", stack[i]);


}
```

Program 4.1: Implementation of stack using arrays

Explanation

The size of the stack was declared as 10. So, stack cannot hold more than 10 elements. The main operations that can be performed on a stack are push and pop. How ever, in a program, we need to provide two more options , namely, *showelements* and *exit*. *showelements* will display the elements of the stack. In case, the user is not interested to perform any operation on the stack and would like to get out of the program, then s/he will select *exit* option. It will log the user out of the program. *choice* is a variable which will enable the user to select the option from the push, pop, showelements and exit operations. *top* points to the index of the free location in the stack to where the next element can be pushed. *element* is the variable which accepts the integer that has to be pushed to the stack or will hold the top element of the stack that has to be popped from the stack. The array *stack* can hold at most 10 elements. *push* and *pop* will perform the operations of pushing the element to the stack and popping the element from the stack respectively.

4.3.2 Implementation of Stack Using Linked Lists

In the last subsection, we have implemented a stack using an array. When a stack is implemented using arrays, it suffers from the basic limitation of an array – that is, its size cannot be increased or decreased once it is declared. As a result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list. In the case of a linked stack, we shall push and pop nodes from one end of a linked list.
The stack, as linked list is represented as a singly connected list. Each node in the linked list contains the data and a pointer that gives location of the next node in the list. Program 4.2 implements a stack using linked lists.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

/* Definition of the structure node */

typedef struct node
{
  int data;
  struct node *next;
} ;

/* Definition of push function */

void  push(node **tos,int item)

{
   node *temp;
   temp=(node*)malloc(sizeof(node));      /* create a new node dynamically */
   if(temp==NULL)                         /* If sufficient amount of memory is */
    {                                     /* not available, the function malloc will */
     printf("\nError: Insufficient Memory Space");       /* return NULL to temp */
     getch();
     return;
     }
    else                                  /* otherwise*/
    {
        temp->data=item;   /* put the item in the data portion of  node*/

         temp->next=*tos;                 /*insert this node at the front of the stack */
        *tos=temp;                        /* managed by linked list*/
```

```
        }
}                                       /*end of function push*/


        /* Definition of pop function */
int pop(node **tos)
{
    node *temp;
    temp=*tos;
    int item;
    if(*tos==NULL)
        return(NULL);
   else
        {
           *tos=(*tos)->next;                   /* To pop an element from stack*/
            item=temp->data;                     /* remove the front node of the */
            free(temp);                          /* stack managed by L.L*/
            return (item);
        }
}  /*end of function pop*/

/* Definition of display function */
void display(node *tos)

{
 node *temp=tos;
 if(temp==NULL)                 /* Check whether the stack is empty*/
   {
     printf("\nStack is empty");
     return;
  }
     else
        {
          while(temp!=NULL)
          {
             printf("\n%d",temp->data);    /* display all the values of the stack*/
             temp=temp->next;              /* from the front node to the last node*/
          }
        }

  }                                          /*end of function display*/


/* Definition of main function */
void main()

{
  int item, ch;
  char choice='y';
  node *p=NULL;
  do
   {
        clrscr();
        printf("\t\t\t\t*****MENU*****");
```

```
        printf("\n\t\t\t1. To PUSH an element");
        printf("\n\t\t\t2. To POP an element");
        printf("\n\t\t\t3. To DISPLAY the elements of stack");
        printf("\n\t\t\t4. Exit");
        printf("\n\n\n\t\t\tEnter your choice:-");
        scanf("%d",&ch);
        switch(ch)
                {
                case 1:
                        printf("\n Enter an element which you want to push ");
                        scanf("%d",&item);
                        push(&p,item);
                        break;
                case 2:
                        item=pop(&p);
                        if(item!=NULL);
                        printf("\n Detected item is%d",item);
                        break;
                case 3:
                        printf("\nThe elements of stack are");
                        display(p);
                        break;
                case 4:
                        exit(0);

                }       /*switch closed */
                printf("\n\n\t\t Do you want to run it again y/n");
                scanf("%c",&choice);
    } while(choice=='y');

}
                                                /*end of function main*/
```

Program 4.2: Implementation of Stack using Linked Lists

Similarly, as we did in the implementation of stack using arrays, to know the working of this program, we executed it thrice and pushed 3 elements (10, 20, 30). Then we call the function display in the next run to see the elements in the stack.

Explanation

Initially, we defined a structure called *node*. Each node contains two portions, data and a pointer that keeps the address of the next node in the list. The *Push* function will insert a node at the front of the linked list, whereas *pop* function will delete the node from the front of the linked list. There is no need to declare the size of the stack in advance as we have done in the program where in we implemented the stack using arrays since we create nodes dynamically as well as delete them dynamically. The function *display* will print the elements of the stack.

☞ Check Your Progress 1

1) State True or False.

   (a) Stacks are sometimes called FIFO lists.

   (b) Stack allows Push and Pop from both ends.

   (c) TOS (top of the stack) gives the bottom most element in the stack.

2) Comment on the following.

   (a) Why is the linked list representation of the stack better than the array representation of the stack?

   (b) Discuss the underflow and overflow problem in stacks.

4.4 ALGORITHMIC IMPLEMENTATION OF MULTIPLE STACKS

So far, now we have been concerned only with the representation of a single stack. What happens when a data representation is needed for several stacks? Let us see an array X whose dimension is m. For convenience, we shall assume that the indexes of the array commence from 1 and end at m. If we have only 2 stacks to implement in the same array X, then the solution is simple.

Suppose A and B are two stacks. We can define an array stack A with n_1 elements and an array stack B with n_2 elements. Overflow may occur when either stack A contains more than n_1 elements or stack B contains more than n_2 elements.

Suppose, instead of that, we define a single array stack with $n = n_1 + n_2$ elements for stack A and B together. See the *Figure 4.4* below. Let the stack A "grow" to the right, and stack B "grow" to the left. In this case, overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements. It does not matter how many elements individually are there in each stack.

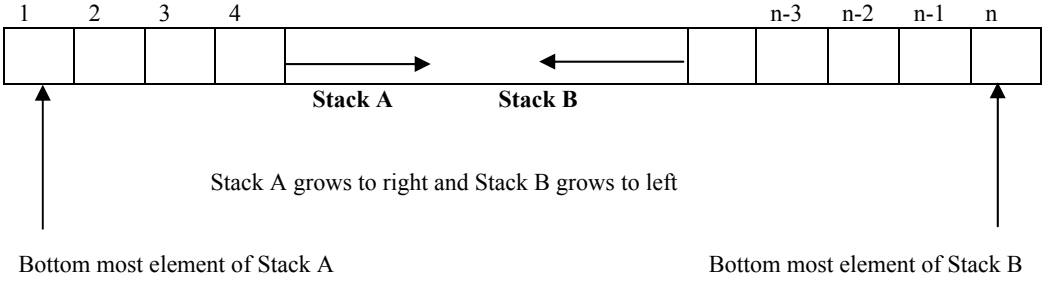
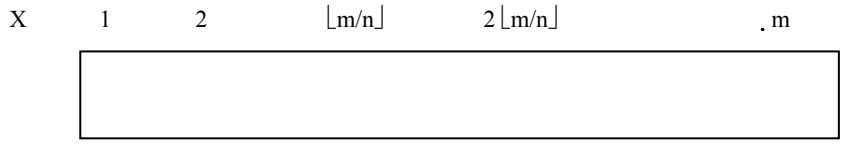


Figure 4.4: Implementation of multiple stacks using arrays

But, in the case of more than 2 stacks, we cannot represent these in the same way because a one-dimensional array has only two fixed points X(1) and X(m) and each stack requires a fixed point for its bottom most element. When more than two stacks, say **n**, are to be represented sequentially, we can initially divide the available memory X(1:m) into **n** segments. If the sizes of the stacks are known, then, we can allocate the segments to them in proportion to the expected sizes of the various stacks. If the sizes of the stacks are not known, then, X(1:m) may be divided into equal segments. For each stack i, we shall use BM (i) to represent a position one less than the position in X for the bottom most element of that stack. TM(i), $1 \leq i \leq n$ will point to the topmost element of stack i. We shall use the boundary condition BM (i) = TM (i) iff the i^{th} stack is empty (refer to *Figure 4.5*). If we grow the i^{th} stack in lower memory indexes than the $i+1^{st}$ stack, then, with roughly equal initial segments we have
BM (i) = TM (i) = $\lfloor m/n \rfloor (i - 1)$, $1 \leq i \leq n$, as the initial values of BM (i) and TM (i).



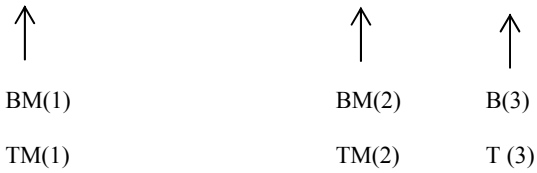


Figure 4.5: Initial configuration for n stacks in X(1:m)

All stacks are empty and memory is divided into roughly equal segments.

Figure 4.6 depicts an algorithm to add an element to the i^{th} stack. *Figure 4.7* depicts an algorithm to delete an element from the i^{th} stack.

```
    ADD(i,e)
Step1: if TM (i)=BM (i+1)
        Print "Stack is full" and exit
Step2: [Increment the pointer value by one]
            TM (i)← TM (i)+1
            X(TM (i))← e
Step3: Exit
```

Figure 4.6: Algorithm to add an element to i^{th} stack

//delete the topmost elements of stack i.

```
DELETE(i,e)
Step1: if TM (i)=BM (i)
        Print "Stack is empty" and exit
Step2: [remove the topmost item]
            e← X(TM (i))
            TM (i)←TM(i)-1
Step3: Exit
```

Figure 4.7: Algorithm to delete an element from i^{th} stack

4.5 APPLICATIONS

Stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. Polish notations are evaluated by stacks. Conversions of different notations (Prefix, Postfix, Infix) into one another are performed using stacks. Stacks are widely used inside computer when recursive functions are called. The computer evaluates an arithmetic expression written in infix notation in two steps. First, it converts the infix expression to postfix expression and then it evaluates the postfix expression. In each step, stack is used to accomplish the task.

4.6 SUMMARY

In this unit, we have studied how the stacks are implemented using arrays and using liked list. Also, the advantages and disadvantages of using these two schemes were discussed. For example, when a stack is implemented using arrays, it suffers from the basic limitations of an array (fixed memory). To overcome this problem, stacks are implemented using linked lists. This unit also introduced learners to the concepts of multiple stacks. The problems associated with the implementation of multiple stacks are also covered.

☞ Check Your Progress 2

1) Multiple stacks can be implemented using _____.
2) _____ are evaluated by stacks.
3) Stack is used whenever a _____ function is called.

4.7 SOLUTIONS / ANSWERS

Check Your Progress 1

1) (a) False (b) False
(c) False

Check Your Progress 2

1) Arrays or Pointers
2) Postfix expressions
3) Recursive

4.8 FURTHER READINGS

1. *Data Structures Using C and C++*, Yedidyah Langsam,Moshe J. Augenstein,Aaron M Tenenbaum, Second Edition,PHI publications.

2. *Data Structures,* Seymour Lipschutz, Schaum's Outline series, Mc GrawHill.

Reference Websites

http://www.cs.queensu.ca