BLM267

Chapter 7: Stacks

Data Structures Using C, Second Edition
Reema Thareja

- Introduction to Stacks
- Array Representation of Stacks
- Operations on a Stack
- Linked Representation of Stacks
- Operations on a Linked Stack
- Multiple Stacks
- Application of Stacks

Introduction to Stacks

- 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.

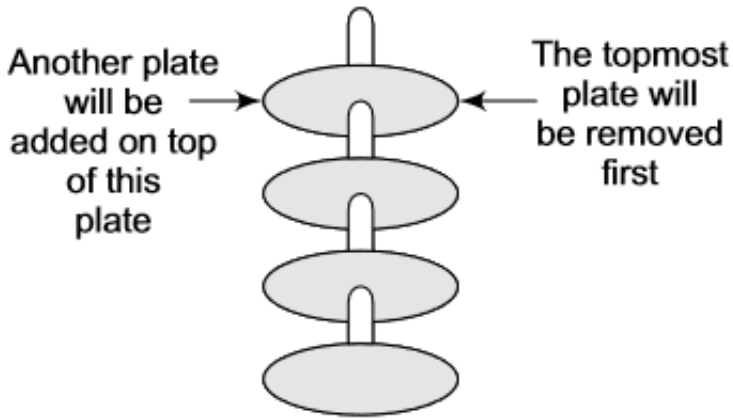

Figure 7.1 Stack of plates

Introduction to Stacks

- 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.
- 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.

Introduction to Stacks

- 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.

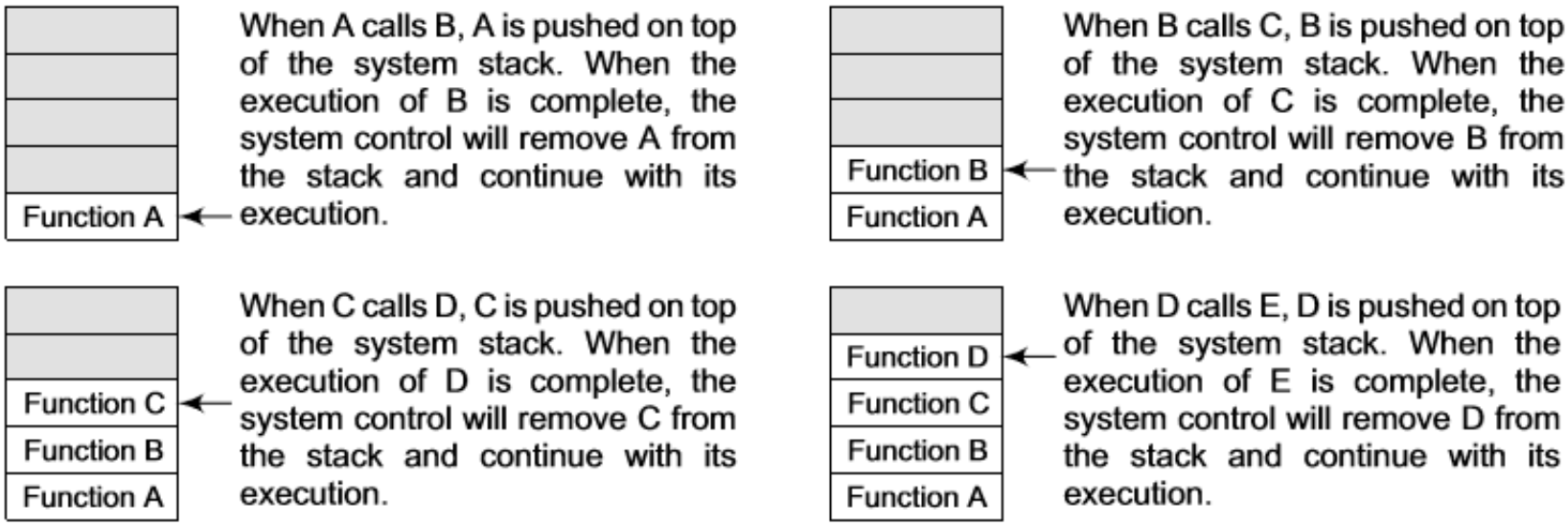

Figure 7.2 System stack in the case of function calls

Introduction to Stacks

- 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.

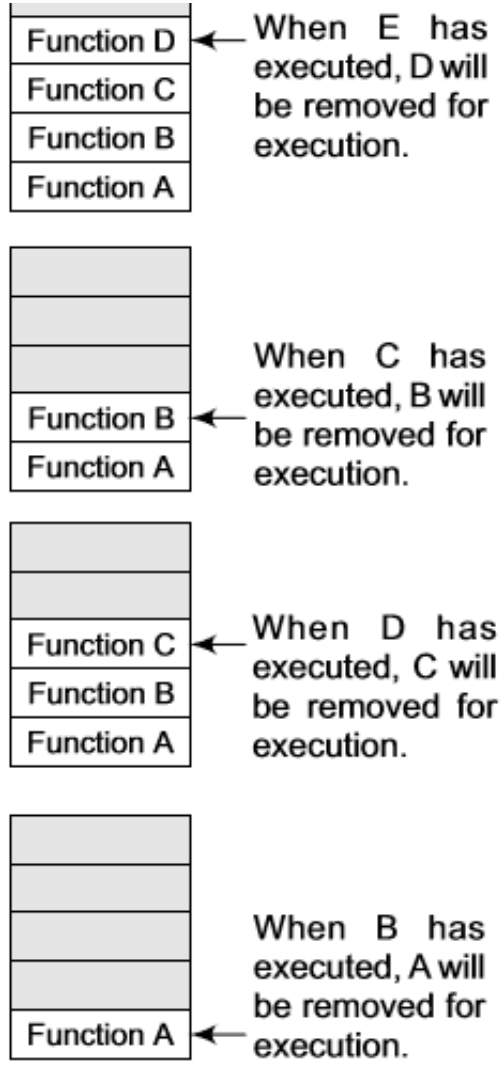

Figure 7.3 System stack when a called function returns

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.
- 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

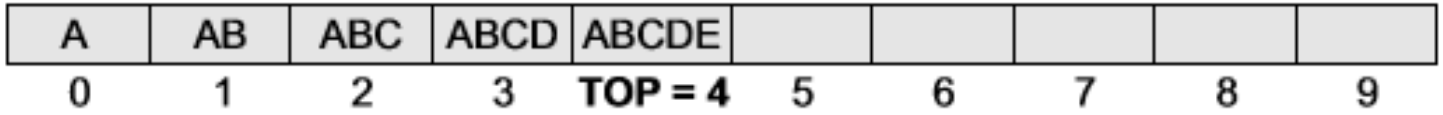

Figure 7.4 Stack

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.

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.

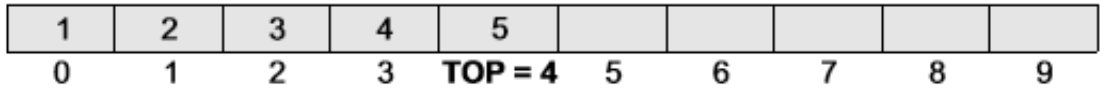

Figure 7.5 Stack

Operations on a 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 7.7 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.

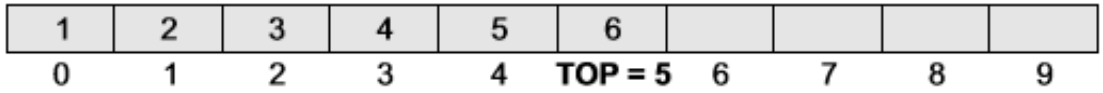

Figure 7.6 Stack after insertion

```
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

Operations on a Stack

- 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.

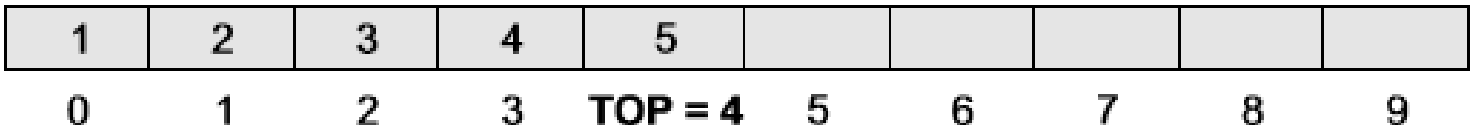

Figure 7.8 Stack

Operations on a Stack

- Pop Operation
- 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.
- 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.

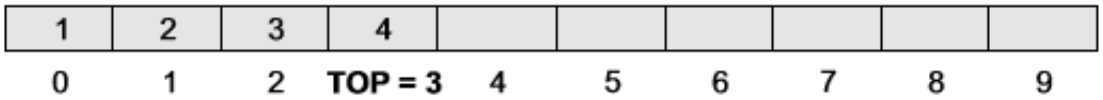

Figure 7.9 Stack after deletion

```
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

Operations on a Stack

- 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.
- Here, the Peek operation will return 5, as it is the value of f the stack.

```
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

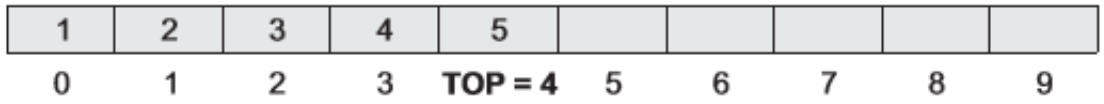

Figure 7.12 Stack

Operations on a Stack


```
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;
        }
}
```

Operations on a Stack

```
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]);
}
```

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).

Linked Representation of Stacks

- 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.

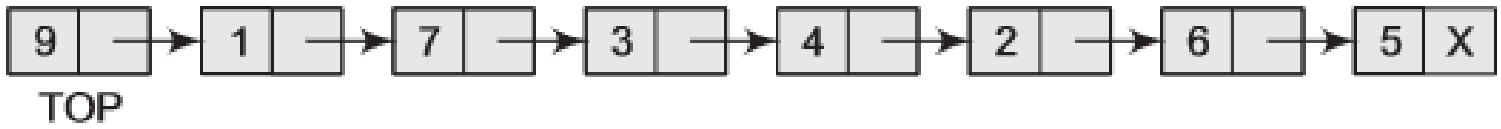

Figure 7.13 Linked stack

Operations on a Linked Stack

- 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.

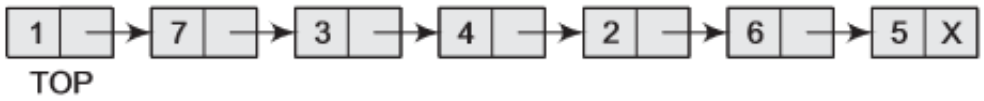

Figure 7.14 Linked stack

- 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

Operations on a Linked Stack

- 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 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.

```
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

Operations on a Linked Stack

- 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.

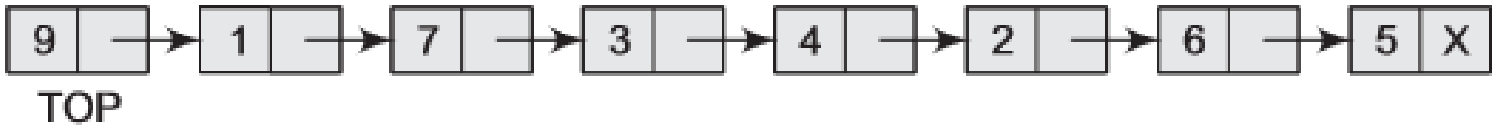

Figure 7.17 Linked stack

Operations on a Linked Stack

- Pop Operation
- 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.
- 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

```
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

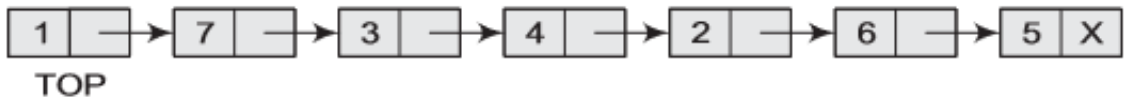

Figure 7.18 Linked stack after deletion of the topmost element

Operations on a Linked Stack


```
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;
}
```

Operations on a Linked Stack

```
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;
}
```

Operations on a Linked Stack

```
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;
}
```

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.

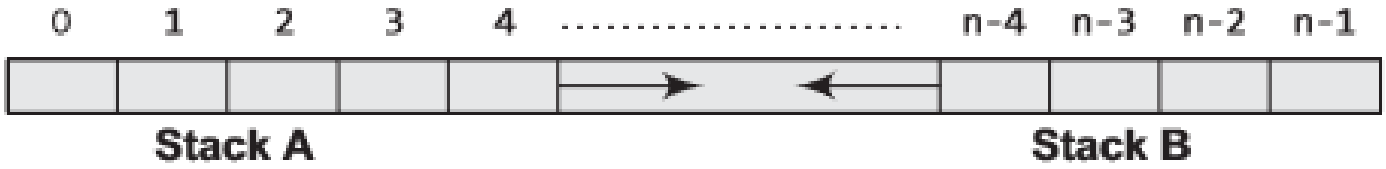

Figure 7.20 Multiple stacks

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 these stacks, it is important to note one thing—Stack A will grow from left to right, whereas Stack B will grow from right to left at the same time.
- Extending this concept to multiple stacks, a stack can also be used to represent n number of stacks in the same array.
- That is, if we have a STACK(n), then each stack I will be allocated an equal amount of space bounded by indices b(i) and e(i).
- Thi

Figure 7.21 Multiple stacks

Applications of Stacks

- In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution.
- The topics that will be discussed in this section include the following:
  - Reversing a list
  - Parentheses checker
  - Conversion of an infix expression into a postfix expression
  - Evaluation of a postfix expression
  - Conversion of an infix expression into a prefix expression
  - Evaluation of a prefix expression

Applications of Stacks

- Reversing a List
- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack.
- Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

Applications of Stacks

4. Write a program to reverse a list of given numbers.

```
#include <stdio.h>
int main()
{
        int val, n, i,
        arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array : ");
        for(i=0;i<n;i++)
                scanf("%d", &arr[i]);
        for(i=0;i<n;i++)
                push(arr[i]);
        for(i=0;i<n;i++)
        {
                val = pop();
                arr[i] = val;
        }
        printf("\n The reversed array is : ");
        for(i=0;i<n;i++)
                printf("\n %d", arr[i]);
        getche");
        return 0;
}
void push(int val)
{
        stk[++top] = val;
```

Applications of Stacks

- Implementing Parentheses Checker
- Stacks can be used to check the validity of parentheses in any algebraic expression.
- For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.
- For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid.
- Look at the program below which traverses an algebraic expression to check for its validity.

5. Write a program to check nesting of parentheses using a stack.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 10
int top = -1;
int stk[MAX];
void push(char);

char pop();
```

Applications of Stacks

```
{
        char exp[MAX],temp;
        int i, flag=1;
        clrscr();
        printf("Enter an expression : ");
        gets(exp);
        for(i=0;i<strlen(exp);i++)
        {
                if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
                        push(exp[i]);
                if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
                        if(top == -1)
                                flag=0;
                        else
                        {
                                temp=pop();
                                if(exp[i]==')' && (temp=='{' || temp=='['))
                                        flag=0;
                                if(exp[i]=='}' && (temp=='(' || temp=='['))
                                        flag=0;
                                if(exp[i]==']' && (temp=='(' || temp=='{'))
                                        flag=0;
                        }
        }
        if(top>=0)
                flag=0;
        if(flag==1)
                printf("\n Valid expression");
        else
                printf("\n Invalid expression");
```

Applications of Stacks

```
void push(char c)
{
        if(top == (MAX-1))
                printf("Stack Overflow\n");
        else
        {
                top=top+1;
                stk[top] = c;
        }
}
char pop()
{
        if(top == -1)
                printf("\n Stack Underflow");
        else
                return(stk[top--]);
}
```

Output

```
Enter an expression : (A + (B - C))
Valid Expression
```

Applications of Stacks

- Evaluation of Arithmetic Expressions
- Polish Notations
- Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions.
- But before learning about prefix and postfix notations, let us first see what an infix notation is.
- We all are familiar with the infix notation of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed in between the operands.
- For example, A+B; here, plus operator is placed between the two operands A and B.
- Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression.
- Information is needed about operator precedence and associativity rules, and brackets which override these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations

Applications of Stacks

- **Evaluation of Arithmetic Expressions**
- Postfix notation was developed by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher.
- His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.
- In postfix notation, as the name suggests, the operator is placed after the operands.
- For example, if an expression is written as A+B in infix notation, the same expression can be written as AB+ in postfix notation.
- The order of evaluation of a postfix expression is always from left to right.
- Even brackets cannot alter the order of evaluation.
- The expression (A + B) * C can be written as: AB+C* in the postfix notation.

Applications of Stacks

- Evaluation of Arithmetic Expressions
- A postfix operation does not even follow the rules of operator precedence.
- The operator which occurs first in the expression is operated first on the operands.
- For example, given a postfix notation AB+C*.
- While evaluation, addition will be performed prior to multiplication.
- Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them.
- In the example, AB+C*, + is applied on A and B, then * is applied on the result of addition and C.

Applications of Stacks

- Evaluation of Arithmetic Expressions
- Although a prefix notation is also evaluated from left to right, the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands.
- For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity, and even brackets cannot alter the order of evaluation.

Applications of Stacks

Example 7.1 Convert the following infix expressions into postfix expressions.

Solution

(a) (A-B) * (C+D)

[AB-] * [CD+]

AB-CD+*

(b) (A + B) / (C + D) - (D * E)

[AB+] / [CD+] - [DE*]

[AB+CD+/] - [DE*]

AB+CD+/DE*-

Example 7.2 Convert the following infix expressions into prefix expressions.

Solution

(a) (A + B) * C

(+AB)*C

*+ABC

(b) (A-B) * (C+D)

[-AB] * [+CD]

*-AB+CD

(c) (A + B) / (C + D) - (D * E)

[+AB] / [+CD] - [*DE]

[/+AB+CD] - [*DE]

-/+AB+CD*DE

Applications of Stacks

- Conversion of an Infix Expression into a Postfix Expression
- Let I be an algebraic expression written in infix notation.
- I may contain parentheses, operands, and operators.
- For simplicity of the algorithm we will use only +, –, *, /, % operators.
- The precedence of these operators can be given as follows:
- Higher priority *, /, %
- Lower priority +, –
- No doubt, the order of evaluation of these operators can be changed by making use of parentheses.
- For example, if we have an expression A + B * C, then first B * C will be done and the result will be added to A.
- But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

Applications of Stacks

- Conversion of an Infix Expression into a Postfix Expression
- The algorithm given below transforms an infix expression into postfix expression, as shown in Fig. 7.22.
- The algorithm accepts an infix expression that may contain operators, operands, and parentheses.
- For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (—) operators and that operators with same precedence are performed from left-to-right.
- The algorithm uses a stack to temporarily hold operators.
- The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack.
- The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression.
- The algorithm is repeated until the stack is empty.

Applications of Stacks

- Conversion of an Infix Expression into a Postfix Expression

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Figure 7.22 Algorithm to convert an infix notation to postfix notation

Applications of Stacks

- Conversion of an Infix Expression into a Postfix Expression

Solution

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

Example 7.3 Convert the following infix expression into postfix expression using the algorithm given in Fig. 7.22.

(a) A – (B / C + (D % E * F) / G)* H

(b) A – (B / C + (D % E * F) / G)* H)

Programming Example

6. Write a program to convert an infix expression into its equivalent postfix notation.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char st[MAX];
int top=-1;
void push(char st[], char);
char pop(char st[]);
```

Applications of Stacks

```
 InfixtoPostfix(char source[], char target[]);
getPriority(char);
main()

     char infix[100], postfix[100];
     clrscr();
     printf("\n Enter any infix expression : ");
     gets(infix);
     strcpy(postfix, "");
     InfixtoPostfix(infix, postfix);
     printf("\n The corresponding postfix expression is : '
     puts(postfix);
     getch();
     return 0;
```

Applications of Stacks


```
void InfixtoPostfix(char source[], char target[])
{
        int i=0, j=0;
        char temp;
        strcpy(target, "");
        while(source[i]!='\0')
        {
                if(source[i]=='(')
                {
                        push(st, source[i]);
                        i++;
                }
                else if(source[i] == ')')
                {
                        while((top!=-1) && (st[top]!='('))
                        {
                                target[j] = pop(st);
                                j++;
                        }
                        if(top==-1)
                        {
                                printf("\n INCORRECT EXPRESSION");
                                exit(1);
                        }
                        temp = pop(st);//remove left parenthesis
                        i++;
                }
                else if(isdigit(source[i]) || isalpha(source[i]))
                {
                        target[j] = source[i];
                        j++;
                        i++;
                }
                else if (source[i] == '+' || source[i] == '-' || source[i] == '*'
source[i] == '/' || source[i] == '%')
                {
                        while( (top!=-1) && (st[top]!= '(') && (getPriority(st[to
> getPriority(source[i])))
                        {
                                target[j] = pop(st);
                                j++;
                        }
                        push(st, source[i]);
                        i++;
                }
                else
```

```
                        exit(1);
                }
        }
        while((top!=-1) && (st[top]!='('))
        {
                target[j] = pop(st);
                j++;
        }
        target[j]='\0';
}
int getPriority(char op)
{
        if(op=='/' || op == '*' || op=='%')
                return 1;
        else if(op=='+' || op=='-')
                return 0;
}
void push(char st[], char val)
{
        if(top==MAX-1)
                printf("\n STACK OVERFLOW");
        else
        {
                top++;
                st[top]=val;
        }
}
char pop(char st[])
{
        char val=' ';
        if(top==-1)
                printf("\n STACK UNDERFLOW");
        else
        {
                val=st[top];
                top--;
        }
        return val;
}
```

Output

Applications of Stacks

- Evaluation of a Postfix Expression
- The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation.
- That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.
- Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.

Applications of Stacks

- Evaluation of a Postfix Expression
- Using stacks, any postfix expression can be evaluated very easily.
- Every character of the postfix expression is scanned from left to right.
- If the character encountered is an operand, it is pushed on to the stack.
- However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values.
- The result is then pushed on to the stack. Let us look at Fig. 7.23 which shows the algorithm to evaluate a postfix expression.

Applications of Stacks

- Evaluation of a Postfix Expression
- Let us now take an example that makes use of this algorithm.
- Consider the infix expression given as 9 – ((3 * 4) + 8) / 4. Evaluate the expression.
- The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation.
- Look at Table 7.1, which shows the procedure.

Applications of Stacks

- **Evaluation of a Postfix Expression**

```
ep 1: Add a ")" at the end of the
      postfix expression
ep 2: Scan every character of the
      postfix expression and repeat
      Steps 3 and 4 until ")"is encountered
ep 3: IF an operand is encountered,
      push it on the stack
      IF an operator O is encountered, then
      a. Pop the top two elements from the
         stack as A and B as A and B
      b. Evaluate B O A, where A is the
         topmost element and B
         is the element below A.
      c. Push the result of evaluation
         on the stack
      [END OF IF]
ep 4: SET RESULT equal to the topmost element
      of the stack
ep 5: EXIT
```

ıre 7.23 Algorithm to evaluate a postfix expression

Table 7.1 Evaluation of a postfix expression

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Applications of Stacks

7. Write a program to evaluate a postfix expression.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=-1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
        float val;
        char exp[100];
        clrscr();
        printf("\n Enter any postfix expression : ");
        gets(exp);
        val = evaluatePostfixExp(exp);
        printf("\n Value of the postfix expression = %.2f", val);
        getch();
        return 0;
}
float evaluatePostfixExp(char exp[])
{
        int i=0;
        float op1, op2, value;
        while(exp[i] != '\0')
        {
                if(isdigit(exp[i]))
```

Applications of Stacks

```
                        push(st, (float)(exp[i]-'0'));
                else
                {
                        op2 = pop(st);
                        op1 = pop(st);
                        switch(exp[i])
                        {
                                case '+':
                                        value = op1 + op2;
                                        break;
                                case '-':
                                        value = op1 - op2;
                                        break;
                                case '/':
                                        value = op1 / op2;
                                        break;
                                case '*':
                                        value = op1 * op2;
                                        break;
                                case '%':
                                        value = (int)op1 % (int)op2;
                                        break;
                        }
                        push(st, value);
                }
                i++;
        }
        return(pop(st));
}
```

Applications of Stacks

```
void push(float st[], float val)
{
        if(top==MAX-1)
                printf("\n STACK OVERFLOW");
        else
        {
                top++;
                st[top]=val;
        }
}
float pop(float st[])
{
        float val=-1;
        if(top==-1)
                printf("\n STACK UNDERFLOW");
        else
        {
                val=st[top];
                top--;
        }
        return val;
}
```

Output

```
Enter any postfix expression : 9 3 4 * 8 + 4 / -
Value of the postfix expression = 4.00
```

Applications of Stacks

- **Conversion of an Infix Expression into a Prefix Expression**
- There are two algorithms to convert an infix expression into its equivalent prefix expression.
- The first algorithm is given in Fig. 7.24, while the second algorithm is shown in Fig. 7.25.

```
Step 1: Scan each character in the infix
        expression. For this, repeat Steps
        2-8 until the end of infix expression
Step 2: Push the operator into the operator stack,
        operand into the operand stack, and
        ignore all the left parentheses until
        a right parenthesis is encountered
Step 3: Pop operand 2 from operand stack
Step 4: Pop operand 1 from operand stack
Step 5: Pop operator from operator stack
Step 6: Concatenate operator and operand 1
Step 7: Concatenate result with operand 2
Step 8: Push result into the operand stack
Step 9: END
```

Figure 7.24 Algorithm to convert an infix expression into prefix expression

```
tep 1: Reverse the infix string. Note that
       while reversing the string you must
       interchange left and right parentheses.
tep 2: Obtain the postfix expression of the
       infix expression obtained in Step 1.
tep 3: Reverse the postfix expression to get
       the prefix expression
```

ure 7.25 Algorithm to convert an infix expression into prefix expression

Applications of Stacks

- **Conversion of an Infix Expression into a Prefix Expression**
- The corresponding prefix expression is obtained in the operand stack. For example, given an infix expression (A – B / C) * (A / K – L)
- Step 1:Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses. (L – K / A) * (C / B – A)
- Step 2:Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.
  The expression is: (L – K / A) * (C / B – A)
  Therefore,
  (L – (K A /)) * ((C B /) – A)
  = (LKA/–) * (CB/A–) = L K A / – C B/A – *
- Step 3:Reverse the postfix expression to get the prefix expression Therefore,
  the prefix expression is * – A / B C – /A K L

Applications of Stacks

8. Write a program to convert an infix expression to a prefix expression.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
char st[MAX];
int top=-1;
void reverse(char str[]);
void push(char st[], char);
char pop(char st[]);
void InfixtoPostfix(char source[], char target[]);
int getPriority(char);
char infix[100], postfix[100], temp[100];
int main()
{
        clrscr();
        printf("\\n Enter any infix expression : ");
        gets(infix);
        reverse(infix);
        strcpy(postfix, "");
        InfixtoPostfix(temp, postfix);
        printf("\n The corresponding postfix expression is : ");
        puts(postfix);
        strcpy(temp,"");
        reverse(postfix);
```

Applications of Stacks

```
        printf("\n The prefix expression is : \n");
        puts(temp);
        getch();
        return 0;
}
void reverse(char str[])
{
        int len, i=0, j=0;
        len=strlen(str);
        j=len-1;
        while(j>= 0)
        {
                if (str[j] == '(')
                        temp[i] = ')';
                else    if ( str[j] == ')')
                        temp[i] = '(';
                else
                        temp[i] = str[j];
                i++,    j--;
        }
        temp[i] = '\0';
}
```

Applications of Stacks

```
        char temp;
        strcpy(target, "");
        while(source[i]!= '\0')
        {
                if(source[i]=='(')
                {
                        push(st, source[i]);
                        i++;
                }
                else if(source[i] == ')')
                {
                        while((top!=-1) && (st[top]!='('))
                        {
                                target[j] = pop(st);
                                j++;
                        }
                        if(top==-1)
                        {
                                printf("\n INCORRECT EXPRESSION");
                                exit(1);
                        }
                        temp = pop(st); //remove left parentheses
                        i++;
                }
                else if(isdigit(source[i]) || isalpha(source[i]))
                {
                        target[j] = source[i];
                        j++;
                        i++;
                }
                else if( source[i] == '+' || source[i] == '-' || source[i] == '*' ||
source[i] == '/' || source[i] == '%')
```

Applications of Stacks

```
> getPriority(source[i])))
                        {
                                target[j] = pop(st);
                                j++;
                        }
                        push(st, source[i]);
                        i++;
                }
                else
                {
                        printf("\n INCORRECT ELEMENT IN EXPRESSION");
                        exit(1);
                }
        }
        while((top!=-1) && (st[top]!='('))
        {
                target[j] = pop(st);
                j++;
        }
        target[j]='\0';
}
```

Applications of Stacks

```
{
        if(op=='/' || op == '*' || op=='%')
                return 1;
        else if(op=='+' || op=='-')
                return 0;
}
void push(char st[], char val)
{
        if(top==MAX-1)
                printf("\n STACK OVERFLOW");
        else
        {
                top++;
                st[top] = val;
        }
 }
char pop(char st[])
{
        char val=' ';
        if(top==-1)
                printf("\n STACK UNDERFLOW");
        else
        {
                val=st[top];
                top--;
        }
        return val;
}
```

Output

```
Enter any infix expression : A+B-C*D
The corresponding postfix expression is : AB+CD*-
```

Applications of Stacks

- **Evaluation of a Prefix Expression**
- There are a number of techniques for evaluating a prefix expression.
- The simplest way of evaluation of a prefix expression is given in Fig. 7.26.
- For example, consider the prefix expression + – 9 2 7 * 8 / 4 12. Let us now apply the algorithm to evaluate this expression.

```
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
        (a) Scan the prefix expression
            from right, one character at a
            time.
        (b) If the scanned character is an
            operand, push it on the
            operand stack.
        (c) If the scanned character is an
            operator, then
              (i) Pop two values from the
                  operand stack
             (ii) Apply the operator on
                  the popped operands
            (iii) Push the result on the
                  operand stack
Step 3: END
```

Figure 7.26 Algorithm for evaluation of a prefix

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
–	24, 5
+	29

Applications of Stacks

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int stk[10];
int top=-1;
int pop();
void push(int);
int main()
{
    char prefix[10];
    int len, val, i, opr1, opr2, res;
    clrscr();
    printf("\n Enter the prefix expression : ");
    gets(prefix);
    len = strlen(prefix);
      for(i=len-1;i>=0;i--)
      {
    switch(get_type(prefix[i]))
            {
                case 0:
                     val = prefix[i] - '0';
                     push(val);
                     break;
                case 1:
                     opr1 = pop();
                     opr2 = pop();
                     switch(prefix[i])
                     {
                          case '+':
                               res = opr1 + opr2;
                               break;
                          case '-':
                               res = opr1 - opr2;
                               break;
                          case '*':
                               res = opr1 * opr2;
                               break;
                          case '/':
                               res = opr1 / opr2;
                               break;
                     }
                     push(res);
            }
      }
      printf("\n RESULT = %d", stk[0]);
      getche();
      return 0;
}
void push(int val)
{
      stk[++top] = val;
```

Applications of Stacks

```
}
int pop()
{
        return(stk[top--]);
}
int get_type(char c)
{
        if(c == '+' || c == '-' || c == '*' || c == '/')
                return 1;
        else return 0;
}
```

Output

```
Enter the prefix expression : +-927
RESULT = 14
```