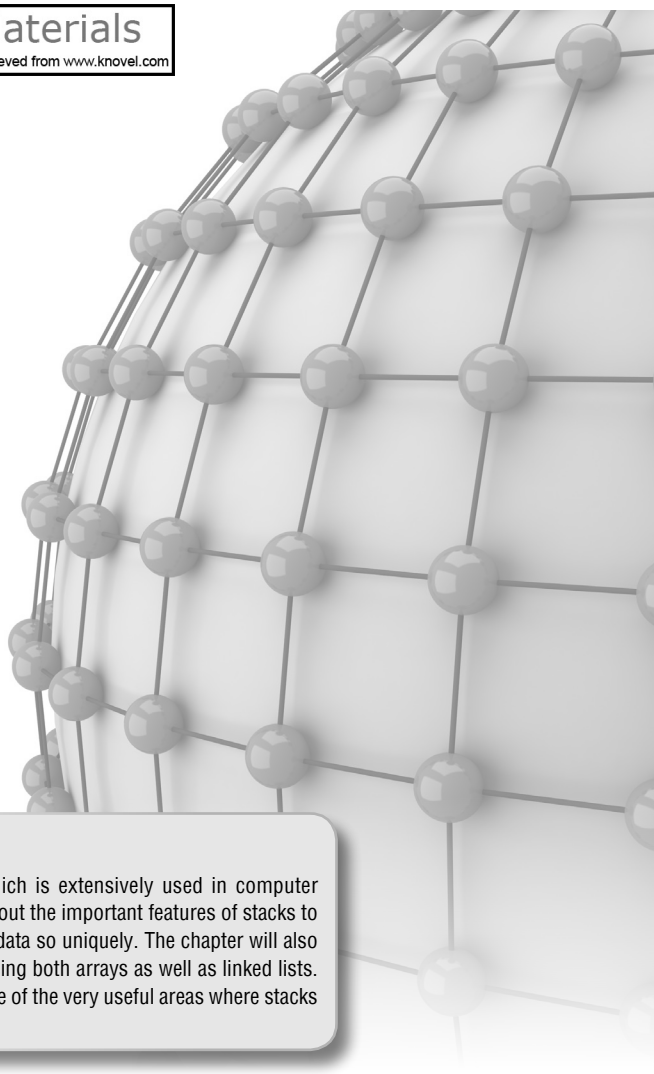
CHAPTER 7

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.

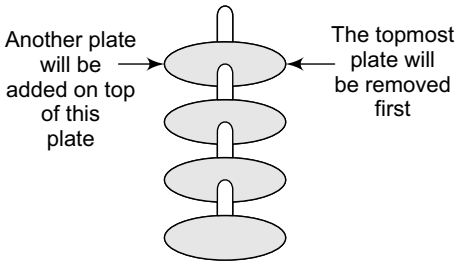


Figure 7.1 Stack of plates

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.

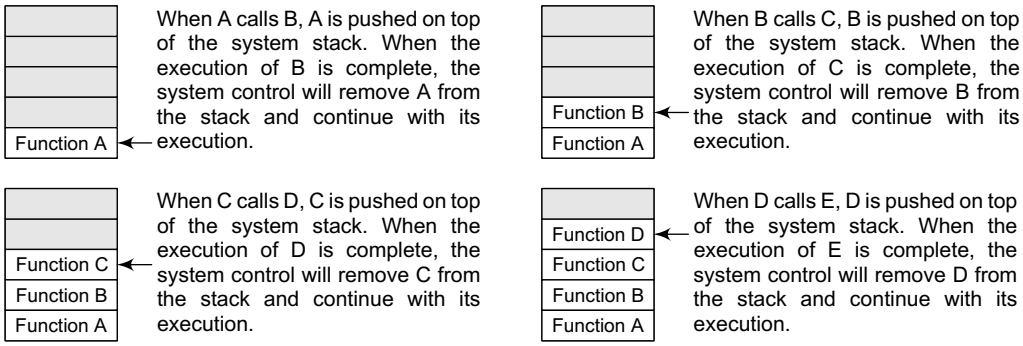


Figure 7.2 System stack in the case of function calls

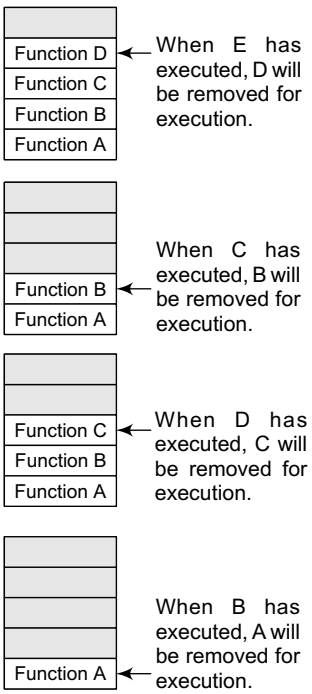


Figure 7.3 System stack when a called function returns to the calling function

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.

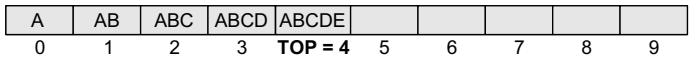


Figure 7.4 Stack

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.

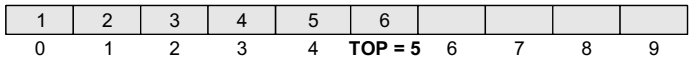


Figure 7.6 Stack after insertion

```
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

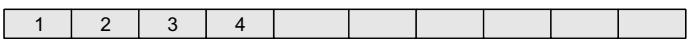


Figure 7.9 Stack after deletion

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.

```
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

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.

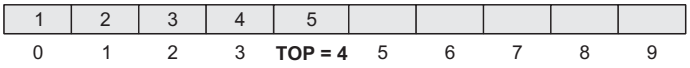


Figure 7.12 Stack

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.

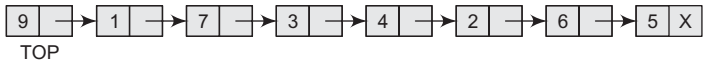


Figure 7.13 Linked stack

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.

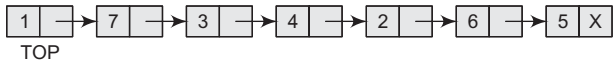


Figure 7.14 Linked stack

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.

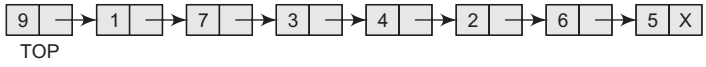


Figure 7.15 Linked stack after inserting a new node

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.

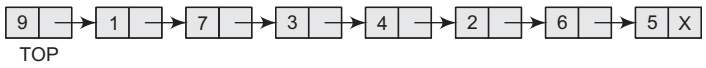


Figure 7.17 Linked stack

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

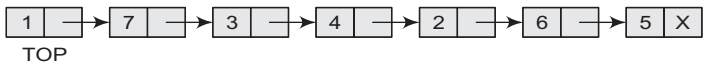


Figure 7.18 Linked stack after deletion of the topmost element

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

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]`. This is shown in Fig. 7.21.

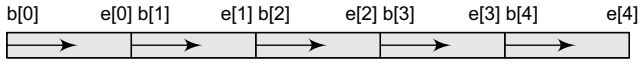


Figure 7.21 Multiple stacks

Programing Example

3. Write a program to implement multiple stacks.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int stack[MAX],topA=-1,topB=MAX;
void pushA(int val)
{
          if(topA==topB-1)
                    printf("\n OVERFLOW");
          else
          {
                    topA+= 1;
                    stack[topA] = val;
          }
}
int popA()
{
          int val;
          if(topA==-1)
          {
                    printf("\n UNDERFLOW");
                    val = -999;
          }
          else
          {
                    val = stack[topA];
                    topA--;
          }
          return val;
}
void display_stackA()
{
          int i;
          if(topA==-1)
                    printf("\n Stack A is Empty");
          else
          {
                    for(i=topA;i>=0;i--)
                         printf("\t %d",stack[i]);
          }
}
void pushB(int val)
{
          if(topB-1==topA)
                    printf("\n OVERFLOW");
          else
```

```
        {
                topB -= 1;
                stack[topB] = val;
        }
}
int popB()
{
        int val;
        if(topB==MAX)
        {
                printf("\n UNDERFLOW");
                val = -999;
        }
        else
        {
                val = stack[topB];
                topB++;
        }
}
void display_stackB()
{
        int i;
        if(topB==MAX)
                printf("\n Stack B is Empty");
        else
        {
                for(i=topB;i<MAX;i++)
                     printf("\t %d",stack[i]);
        }
}
void main()
{
        int option, val;
        clrscr();
        do
        {
                printf("\n *****MENU*****");
                printf("\n 1. PUSH IN STACK A");
                printf("\n 2. PUSH IN STACK B");
                printf("\n 3. POP FROM STACK A");
                printf("\n 4. POP FROM STACK B");
                printf("\n 5. DISPLAY STACK A");
                printf("\n 6. DISPLAY STACK B");
                printf("\n 7. EXIT");
                printf("\n Enter your choice");
                scanf("%d",&option);
                switch(option)
                {
                     case 1: printf("\n Enter the value to push on Stack A : ");
                             scanf("%d",&val);
                             pushA(val);
                             break;
                     case 2: printf("\n Enter the value to push on Stack B : ");
                             scanf("%d",&val);
                             pushB(val);
                             break;
                     case 3: val=popA();
                             if(val!=-999)
                                printf("\n The value popped from Stack A = %d",val);
                             break;
```

```
                              case 4: val=popB();
                                      if(val!=-999)
                                         printf("\n The value popped from Stack B = %d",val);
                                      break;
                              case 5: printf("\n The contents of Stack A are : \n");
                                      display_stackA();
                                      break;
                              case 6: printf("\n The contents of Stack B are : \n");
                                      display_stackB();
                                      break;
                           }
          }while(option!=7);
          getch();
   }
```

Output

```
*****MAIN MENU*****
1. PUSH IN STACK A
2. PUSH IN STACK B
3. POP FROM STACK A
4. POP FROM STACK B
5. DISPLAY STACK A
6. DISPLAY STACK B
7. EXIT
Enter your choice : 1
Enter the value to push on Stack A : 10
Enter the value to push on Stack A : 15
Enter your choice : 5
The content of Stack A are:
15      10
Enter your choice : 4
UNDERFLOW
Enter your choice : 7
```

7.7 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
- Recursion
- Tower of Hanoi

7.7.1 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.

PROGRAMMING EXAMPLE

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

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

```
#include <conio.h>
int stk[10];
int top=-1;
int pop();
void push(int);
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;
}
int pop()
{
        return(stk[top--]);
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements of the array : 1 2 3 4 5
The reversed array is : 5 4 3 2 1
```

7.7.2 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.

PROGRAMMING EXAMPLE

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();
void main()
{
        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");
}
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
```

7.7.3 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.

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`

`AB+C*` in the postfix notation

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`.

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.

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
```

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`.

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.

```
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

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[]);
```

```
void InfixtoPostfix(char source[], char target[]);
int getPriority(char);
int 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;
}
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[top])
> 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';
    }
    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

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

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.

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.

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 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]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Figure 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

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.

Programming Example

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]))
```

```
                        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));
}
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
```

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

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

Figure 7.25 Algorithm to convert an infix expression into 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`**

Programming Example

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);
```

```
        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';
}
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 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] == '%')
                {
                        while( (top!=-1) && (st[top]!= '(') && (getPriority(st[top])
```

```
    > 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';
        }
        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

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

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.

```
Step 1: Accept the prefix 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 expression

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

For example, consider the prefix expression `+ – 9 2 7 * 8 / 4 12`. Let us now apply the algorithm to evaluate this expression.

Programming Example

9. Write a program to evaluate a prefix expression.

```
#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;
```

```
}
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
```

7.7.4 Recursion

In this section we are going to discuss recursion which is an implicit application of the STACK ADT.

A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are

- *Base case,* in which the problem is simple enough to be solved directly without making any further calls to the same function.
- *Recursive case,* in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

To understand recursive functions, let us take an example of calculating factorial of a number. To calculate `n!`, we multiply the number with factorial of the number that is 1 less than that number. In other words, `n! = n × (n-1)!`

Let us say we need to find the value of `5!`

```
5! = 5 × 4 × 3 × 2 × 1
   = 120
```

This can be written as

```
5! = 5 × 4!, where 4!= 4 × 3!
```

Therefore,

```
5! = 5 × 4 × 3!
```

Similarly, we can also write,

```
5! = 5 × 4 × 3 × 2!
```

Expanding further

```
5! = 5 × 4 × 3 × 2 × 1!
```

We know, `1! = 1`

The series of problems and solutions can be given as shown in Fig. 7.27.

PROBLEM	SOLUTION
5!	5×4×3×2×1!
= 5×4!	= 5×4×3×2×1
= 5×4×3!	= 5×4×3×2
= 5×4×3×2!	= 5×4×6
= 5×4×3×2×1!	= 5×24
	= 120

Figure 7.27 Recursive factorial function

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the

factorial of a number. Every recursive function must have a base case and a recursive case. For the factorial function,

Programming Tip

Every recursive function must have at least one base case. Otherwise, the recursive function will generate an infinite sequence of calls, thereby resulting in an error condition known as an infinite stack.

- **Base case** is when `n = 1`, because if `n = 1`, the result will be `1` as `1! = 1`.
- **Recursive case** of the factorial function will call itself but with a smaller value of `n`, this case can be given as

  `factorial(n) = n × factorial (n-1)`

Look at the following program which calculates the factorial of a number recursively.

PROGRAMMING EXAMPLE

10. Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int);   // FUNCTION DECLARATION
int main()
{
        int num, val;
        printf("\n Enter the number: ");
        scanf("%d", &num);
        val = Fact(num);
        printf("\n Factorial of %d = %d", num, val);
        return 0;
}
int Fact(int n)
{
        if(n==1)
                return 1;
        else
            return (n * Fact(n-1));
}
```

Output

```
Enter the number : 5
Factorial of 5 = 120
```

From the above example, let us analyse the steps of a recursive program.

Step 1: Specify the base case which will stop the function from making a call to itself.

Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.

Step 3: Divide the problem into smaller or simpler sub-problems.

Step 4: Call the function from each sub-problem.

Step 5: Combine the results of the sub-problems.

Step 6: Return the result of the entire problem.

Greatest Common Divisor

The greatest common divisor of two numbers (integers) is the largest integer that divides both the numbers. We can find the GCD of two numbers recursively by using the *Euclid's algorithm* that states

$$\text{GCD}(a, b) = \begin{cases} b, & \text{if } b \text{ divides } a \\ \text{GCD}(b, a \bmod b), & \text{otherwise} \end{cases}$$

GCD can be implemented as a recursive function because if `b` does not divide `a`, then we call the same function (GCD) with another set of parameters that are smaller than the original ones.

Here we assume that `a > b`. However if `a < b`, then interchange `a` and `b` in the formula given above.

Working

Assume `a = 62` and `b = 8`

```
GCD(62, 8)
   rem = 62 % 8 = 6
   GCD(8, 6)
          rem = 8 % 6 = 2
          GCD(6, 2)
                  rem = 6 % 2 = 0
          Return 2
   Return 2
Return 2
```

PROGRAMMING EXAMPLE

11. Write a program to calculate the GCD of two numbers using recursive functions.

```
#include <stdio.h>
int GCD(int, int);
int main()
{
        int num1, num2, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &num1, &num2);
        res = GCD(num1, num2);
        printf("\n GCD of %d and %d = %d", num1, num2, res);
        return 0;
}

int GCD(int x, int y)
{
        int rem;
        rem = x%y;
        if(rem==0)
                return y;
        else
                return (GCD(y, rem));
}
```

Output

```
Enter the two numbers : 8 12
GCD of 8 and 12 = 4
```

Finding Exponents

We can also find exponent of a number using recursion. To find x^y, the base case would be when `y=0`, as we know that any number raised to the power `0` is `1`. Therefore, the general formula to find x^y can be given as

$$\text{EXP}(x, y) = \begin{cases} 1, & \text{if } y == 0 \\ x \times \text{EXP}(x^{y-1}), & \text{otherwise} \end{cases}$$

Working

```
exp_rec(2, 4) = 2 × exp_rec(2, 3)
       exp_rec(2, 3) = 2 × exp_rec(2, 2)
             exp_rec(2, 2) 2 × exp_rec(2, 1)
                    exp_rec(2, 1) = 2 × exp_rec(2, 0)
                           exp_rec(2, 0) = 1
                    exp_rec(2, 1) = 2 × 1 = 2
             exp_rec(2, 2) = 2 × 2 = 4
```

```
        exp_rec(2, 3) = 2 × 4 = 8
exp_rec(2, 4) = 2 × 8 = 16
```

Programming Example

12. Write a program to calculate `exp(x,y)` using recursive functions.

```
#include <stdio.h>
int exp_rec(int, int);
int main()
{
        int num1, num2, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &num1, &num2);
        res = exp_rec(num1, num2);
        printf ("\n RESULT = %d", res);
        return 0;
}
int exp_rec(int x, int y)
{
        if(y==0)
                return 1;
        else
                return (x * exp_rec(x, y-1));
}
```

Output

```
Enter the two numbers : 3 4
RESULT = 81
```

The Fibonacci Series

The Fibonacci series can be given as

```
0 1 1 2 3 5 8 13 21 34 55 ……
```

That is, the third term of the series is the sum of the first and second terms. Similarly, fourth term is the sum of second and third terms, and so on. Now we will design a recursive solution to find the `nth` term of the Fibonacci series. The general formula to do so can be given as

As per the formula, `FIB(0) =0` and `FIB(1) = 1`. So we have two base cases. This is necessary because every problem is divided into two smaller problems.

$$\text{FIB}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{FIB}(n-1) + \text{FIB}(n-2), & \text{otherwise} \end{cases}$$

Programming Example

13. Write a program to print the Fibonacci series using recursion.

```
#include <stdio.h>
int Fibonacci(int);
int main()
{
        int n, i = 0, res;
        printf("Enter the number of terms\n");
        scanf("%d",&n);
        printf("Fibonacci series\n");
        for(i = 0; i < n; i++ )
        {
                res = Fibonacci(i);
```

```
                printf("%d\t",res);
        }
        return 0;
}
int Fibonacci(int n)
{
        if ( n == 0 )
                return 0;
        else if ( n == 1 )
                return 1;
        else
                return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Output

```
Enter the number of terms
Fibonacci series
   0    1       1       2       3
```

Types of Recursion

Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. Any recursive function can be characterized based on:

- whether the function calls itself directly or indirectly (*direct or indirect recursion*),
- whether any operation is pending at each recursive call (*tail-recursive* or not), and
- the structure of the calling pattern (*linear or tree-recursive*).

In this section, we will read about all these types of recursions.

```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

Figure 7.28 Direct recursion

Direct Recursion

A function is said to be *directly* recursiveif it explicitly calls itself. For example, consider the code shown in Fig. 7.28. Here, the function `Func()` calls itself for all positive values of `n`, so it is said to be a directly recursive function.

```
int Funcl (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
     return Func1(x-1);
}
```

Figure 7.29 Indirect recursion

Indirect Recursion

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other (Fig. 7.29).

```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

Figure 7.30 Non-tail recursion

Tail Recursion

A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller. When the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

In Fig. 7.30, the factorial function that we have written is a non-tail-recursive function, because there is a pending operation of multiplication to be performed on return from each recursive call.

```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

Figure 7.31 Tail recursion

Whenever there is a pending operation to be performed, the function becomes non-tail recursive. In such a non-tail recursive function, information about each pending operation must be stored, so the amount of information directly depends on the number of calls.

However, the same factorial function can be written in a tail-recursive manner as shown Fig. 7.31.

In the code, `Fact1` function preserves the syntax of `Fact(n)`. Here the recursion occurs in the `Fact1` function and not in `Fact` function. Carefully observe that `Fact1` has no pending operation to be performed on return from recursive calls. The value computed by the recursive call is simply returned without any modification. So in this case, the amount of information to be stored on the system stack is constant (only the values of `n` and `res` need to be stored) and is independent of the number of recursive calls.

Converting Recursive Functions to Tail Recursive

A non-tail recursive function can be converted into a tail-recursive function by using an *auxiliary parameter* as we did in case of the Factorial function. The auxiliary parameter is used to form the result. When we use such a parameter, the pending operation is incorporated into the auxiliary parameter so that the recursive call no longer has a pending operation. We generally use an auxiliary function while using the auxiliary parameter. This is done to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

Linear and Tree Recursion

```
int Fibonacci(int num)
{
  if(num == 0)
    return 0;
  else if (num == 1)
    return 1;
    else
    return (Fibonacci(num - 1) + Fibonacci(num – 2));
}
```

Observe the series of function calls. When the function returns, the pending operations in turn calls the function

Fibonacci(7) = Fibonacci(6) + Fibonacci(5)
Fibonacci(6) = Fibonacci(5) + Fibonacci(4)
Fibonacci(5) = Fibonacci(4) + Fibonacci(3)
Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
Fibonacci(2) = Fibonacci(1) + Fibonacci(0)

Now we have, Fibonacci(2) = 1 + 0 = 1

Fibonacci(3) = 1 + 1 = 2
Fibonacci(4) = 2 + 1 = 3
Fibonacci(5) = 3 + 2 = 5
Fibonacci(6) = 3 + 5 = 8
Fibonacci(7) = 5 + 8 = 13

Figure 7.32 Tree recursion

Recursive functions can also be characterized depending on the way in which the recursion grows in a linear fashion or forming a tree structure (Fig. 7.32).

In simple words, a recursive function is said to be *linearly* recursive when the pending operation (if any) does not make another recursive call to the function. For example, observe the last line of recursive factorial function. The factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another recursive call to `Fact`.

On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function. For example, the `Fibonacci` function in which the pending operations recursively call the `Fib onacci` function.

Tower of Hanoi

The tower of Hanoi is one of the main applications of recursion. It says, 'if you can solve `n-1` cases, then you can easily solve the `nth` case'.

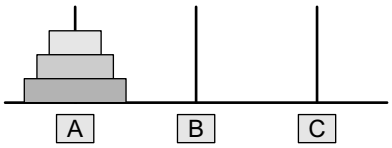


Figure 7.33 Tower of Hanoi

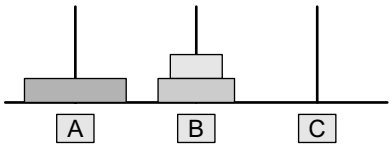


Figure 7.34 Move rings from A to B

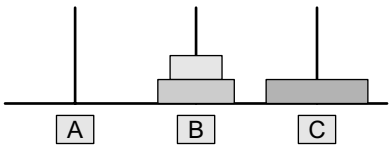


Figure 7.35 Move ring from A to C

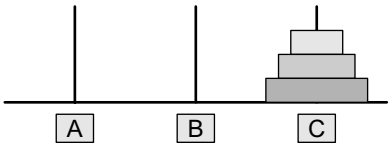


Figure 7.36 Move ring from B to C

Look at Fig. 7.33 which shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

We will be doing this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings `(n-1 rings)` from the source pole to the spare pole. We move the first two rings from pole A to B as shown in Fig. 7.34.

Now that `n-1` rings have been removed from pole A, the `nth` ring can be easily moved from the source pole (A) to the destination pole (C). Figure 7.35 shows this step.

The final step is to move the `n-1` rings from the spare pole (B) to the destination pole (C). This is shown in Fig. 7.36.

To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

Base case: `if n=1`

- Move the ring from A to C using B as spare

Recursive case:

- Move `n – 1` rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move `n – 1` rings from B to C using A as spare

The following code implements the solution of the Tower of Hanoi problem.

```
#include <stdio.h>
int main()
{
      int n;
      printf("\n Enter the number of rings: ");
      scanf("%d", &n);
      move(n,'A', 'C', 'B');
      return 0;
}
void move(int n, char source, char dest, char spare)
{
      if (n==1)
            printf("\n Move from %c to %c",source,dest);
      else
      {
            move(n-1,source,spare,dest);
            move(1,source,dest,spare);
            move(n-1,spare,dest,source);
      }
}
```

Let us look at the Tower of Hanoi problem in detail using the program given above. Figure 7.37 on the next page explains the working of the program using one, then two, and finally three rings.

Recursion versus Iteration

Recursion is more of a top-down approach to problem solving in which the original problem is divided into smaller sub-problems. On the contrary, iteration follows a bottom-up approach that begins with what is known and then constructing the solution step by step.

Recursion is an excellent way of solving complex problems especially when the problem can be defined in recursive terms. For such problems, a recursive code can be written and modified in a much simpler and clearer manner.

However, recursive solutions are not always the best solutions. In some cases, recursive programs may require substantial amount of run-time overhead. Therefore, when implementing a recursive solution, there is a trade-off involved between the time spent in constructing and maintaining the program and the cost incurred in running-time and memory space required for the execution of the program.

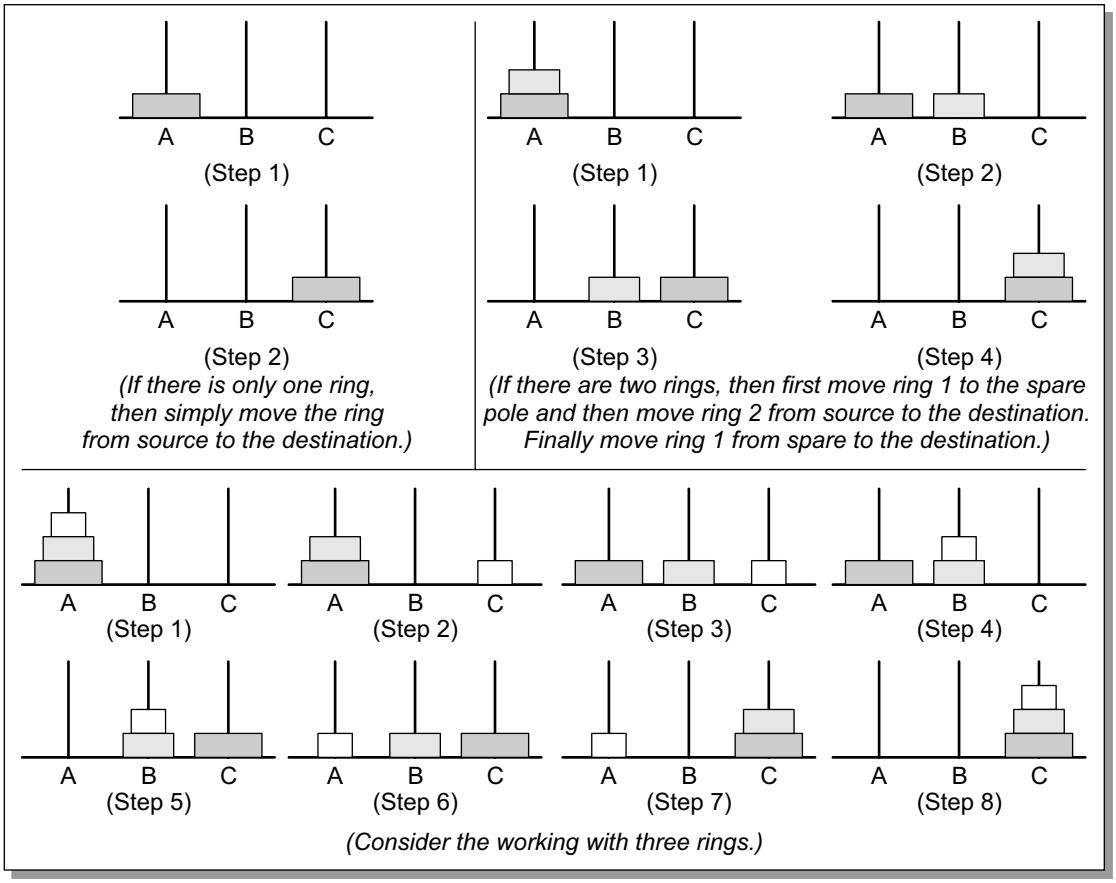


Figure 7.37 Working of Tower of Hanoi with one, two, and three rings

Whenever a recursive function is called, some amount of overhead in the form of a run time stack is always involved. Before jumping to the function with a smaller parameter, the original parameters, the local variables, and the return address of the calling function are all stored on the system stack. Therefore, while using recursion a lot of time is needed to first push all the information on the stack when the function is called and then again in retrieving the information stored on the stack once the control passes back to the calling function.

To conclude, one must use recursion only to find solution to a problem for which no obvious iterative solution is known. To summarize the concept of recursion, let us briefly discuss the pros and cons of recursion.

The advantages of using a recursive program include the following:

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Recursion follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

The drawbacks/disadvantages of using a recursive program include the following:

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly while using global variables.

The advantages of recursion pay off for the extra overhead involved in terms of time and space required.

Points to Remember

- A stack is a linear data structure in which elements 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 is inserted last is the first one to be taken out.
- In the computer's memory, stacks can be implemented using either linked lists or single arrays.
- The storage requirement of linked representation of stack with `n` elements is `O(n)` and the typical time requirement for operations is `O(1)`.
- Infix, prefix, and postfix notations are three different but equivalent notations of writing algebraic expressions.
- In postfix notation, operators are placed after the operands, whereas in prefix notation, operators are placed before the operands.
- Postfix notations are evaluated using stacks. Every character of the postfix expression is scanned from left to right. If the character is an operand, it is pushed onto the stack. Else, if it is an operator, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed onto the stack.
- Multiple stacks means to have more than one stack in the same array of sufficient size.
- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. They are implmented using system stack.

Exercises

Review Questions

1. What do you understand by stack overflow and underflow?
2. Differentiate between an array and a stack.
3. How does a stack implemented using a linked list differ from a stack implemented using an array?
4. Differentiate between `peek()` and `pop()` functions.
5. Why are parentheses not required in postfix/prefix expressions?
6. Explain how stacks are used in a non-recursive program?
7. What do you understand by a multiple stack? How is it useful?
8. Explain the terms infix expression, prefix expression, and postfix expression. Convert the following infix expressions to their postfix equivalents:
   (a) A – B + C (b) A * B + C / D
   (c) (A – B) + C * D / E – C
   (d) (A * B) + (C / D) – (D + E)
   (e) ((A – B) + D / ((E + F) * G))
   (f) (A – 2 * (B + C) / D * E) + F
   (g) 14 / 7 * 3 – 4 + 9 / 2
9. Convert the following infix expressions to their postfix equivalents:
   (a) A – B + C (b) A * B + C / D
   (c) (A – B) + C * D / E – C
   (d) (A * B) + (C / D) – (D + E)
   (e) ((A – B) + D / ((E + F) * G))
   (f) (A – 2 * (B + C) / D * E) + F
   (g) 14 / 7 * 3 – 4 + 9 / 2
10. Find the infix equivalents of the following postfix equivalents:

(a) A B + C * D – (b) ABC * + D –

11. Give the infix expression of the following prefix expressions.
(a) * – + A B C D (b) + – a * B C D

12. Convert the expression given below into its corresponding postfix expression and then evaluate it. Also write a program to evaluate a postfix expression.
10 + ((7 – 5) + 10)/2

13. Write a function that accepts two stacks. Copy the contents of first stack in the second stack. Note that the order of elements must be preserved.
(*Hint: use a temporary stack*)

14. Draw the stack structure in each case when the following operations are performed on an empty stack.
(a) Add A, B, C, D, E, F (b) Delete two letters
(c) Add G (d) Add H
(e) Delete four letters (f) Add I

15. Differentiate between an iterative function and a recursive function. Which one will you prefer to use and in what circumstances?

16. Explain the Tower of Hanoi problem.

Programming Exercises

1. Write a program to implement a stack using a linked list.

2. Write a program to convert the expression "a+b" into "ab+".

3. Write a program to convert the expression "a+b" into "+ab".

4. Write a program to implement a stack that stores names of students in the class.

5. Write a program to input two stacks and compare their contents.

6. Write a program to compute `F(x, y)`, where
`F(x, y)` = `F(x-y, y)` + 1 if y≤x
And `F(x, y)` = `0` if `x<y`

7. Write a program to compute `F(n, r)` where `F(n, r)` can be recursively defined as:
`F(n, r)` = `F(n-1, r)` + `F(n-1, r-1)`

8. Write a program to compute `Lambda(n)` for all positive values of `n` where `Lambda(n)` can be recursively defined as:
`Lambda(n)` = `Lambda(n/2)` + `1 if n>1`
and `Lambda(n)` = `0` if `n =1`

9. Write a program to compute `F(M, N)` where `F(M, N)` can be recursively defined as:
`F(M,N)` = `1` if `M=0` or `M≥N≥1`
and `F(M,N)` = `F(M-1,N)` + `F(M-1, N-1)`, otherwise

10. Write a program to reverse a string using recursion.

Multiple-choice Questions

1. Stack is a
(a) LIFO (b) FIFO (c) FILO (d) LILO

2. Which function places an element on the stack?
(a) `Pop()` (b) `Push()`
(c) `Peek()` (d) `isEmpty()`

3. Disks piled up one above the other represent a
(a) Stack (b) Queue
(c) Linked List (d) Array

4. Reverse Polish notation is the other name of
(a) Infix expression (b) Prefix expression
(c) Postfix expression (d) Algebraic expression

True or False

1. `Pop()` is used to add an element on the top of the stack.

2. Postfix operation does not follow the rules of operator precedence.

3. Recursion follows a divide-and-conquer technique to solve problems.

4. Using a recursive function takes more memory and time to execute.

5. Recursion is more of a bottom-up approach to problem solving.

6. *An indirect* recursive function if it contains a call to another function which ultimately calls it.

7. The peek operation displays the topmost value and deletes it from the stack.

8. In a stack, the element that was inserted last is the first one to be taken out.

9. Underflow occurs when TOP = MAX-1.

10. The storage requirement of linked representation of the stack with n elements is O(n).

11. A push operation on linked stack can be performed in O(n) time.

12. Overflow can never occur in case of multiple stacks.

Fill in the Blanks

1. _____ is used to convert an infix expression into a postfix expression.

2. _____ is used in a non-recursive implementation of a recursive algorithm.

3. The storage requirement of a linked stack with n elements is _____.

4. Underflow takes when _____.

5. The order of evaluation of a postfix expression is from _____.

6. Whenever there is a pending operation to be performed, the function becomes _____ recursive.

7. A function is said to be _____ recursive if it explicitly calls itself.