```
                                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.