```
    }
    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

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

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'.

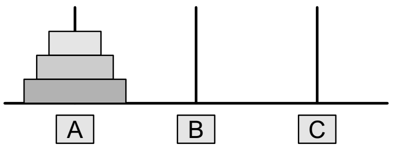

Figure 7.33 Tower of Hanoi

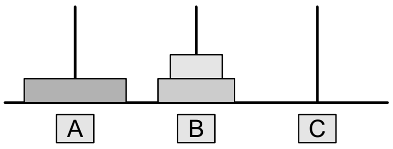

Figure 7.34 Move rings from A to B

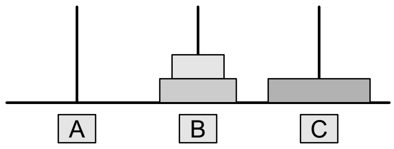

Figure 7.35 Move ring from A to C

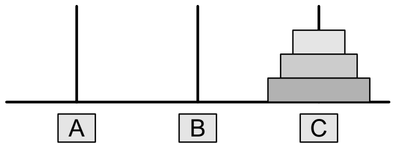

Figure 7.36 Move ring from B to C

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.

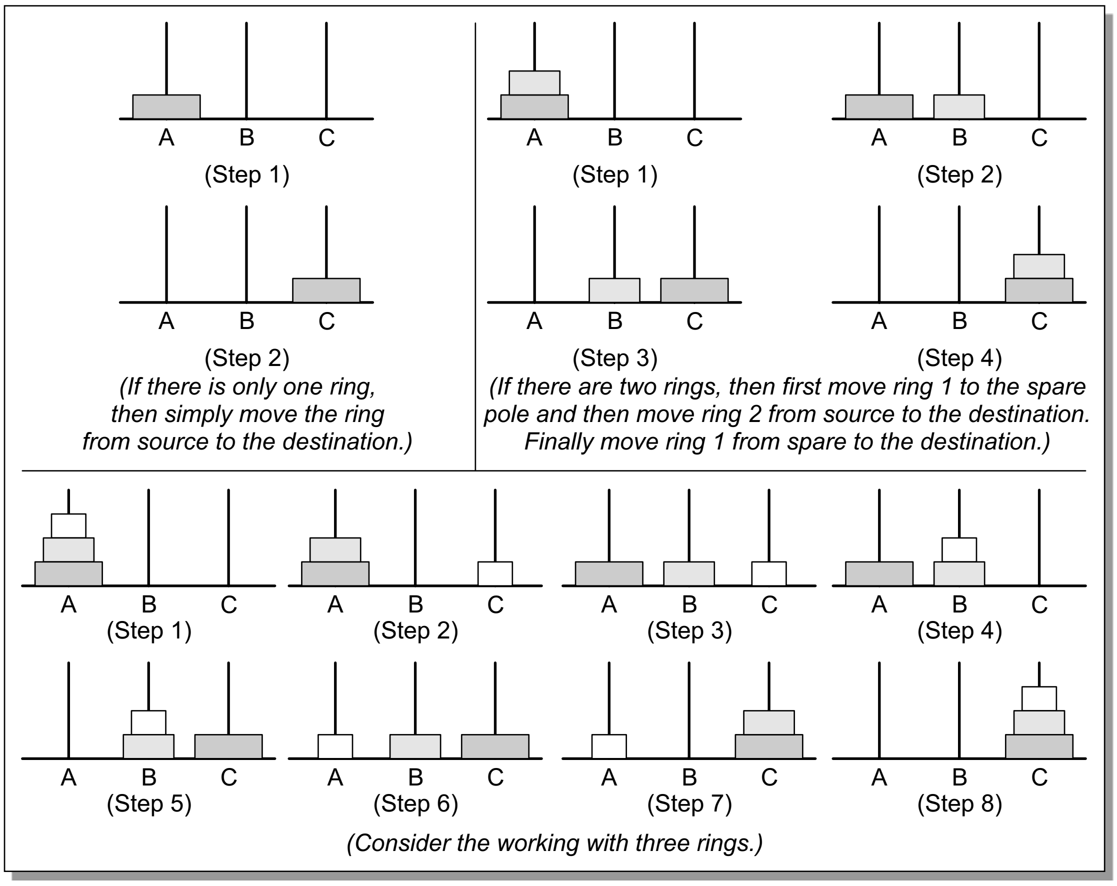
A B C
(Step 1)
A B C
(Step 2)
(If there is only one ring, then simply move the ring from source to the destination.)
A B C
(Step 1)
A B C
(Step 2)
A B C
(Step 3)
A B C
(Step 4)
(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)
A B C
(Step 1)
A B C
(Step 2)
A B C
(Step 3)
A B C
(Step 4)
A B C
(Step 5)
A B C
(Step 6)
A B C
(Step 7)
A B C
(Step 8)
(Consider the working with three rings.)

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.