3.6 SUMMARY

- A **stack** is defined as a linear list in which insertions and deletions take place at the same end. This end is called as the **top** of the stack and the other end is called as the **bottom** of the stack.
- Stack operates on *Last-In-First-Out* (**LIFO**) manner.
- The push operation inserts an element into a stack and pop operation removes an element from the top of the stack.
- To push and pop from the top of the stack, a pointer is maintained called as **top**. When stack top is equal to array size we say that an **overflow** has occurred. Similarly, when the stack is empty, initiating a pop leads to **underflow**.
- The chapter covered the details about the implementation issues of stack in C.
- The stack applications that have been dealt with algorithm and C code were:
  1. Parentheses matching
  2. Postfix expression evaluation
  3. Infix to postfix conversion
  4. Infix to prefix conversion.
- A double stack is new kind of stack useful for many applications like compiler design, etc. A double stack is one where using a single array we operate two stack. The first one grows from left to right and the second grows from right to left.

3.7 EXERCISES

3.1 Explain the working of a Stack. Write C functions for **Push** and **Pop** operations of a stack implemented using arrays. Your functions should take care of *overflow* and *underflow* conditions.

3.2 Write a pseudo-code that splits a given stack in to two different stacks using only push and pop operations. The location at which the stack has to be splitted is given as input and is relative to TOP.

3.3 Assume that s1 and s2 are two stacks. Combine these two stacks by placing all elements of the second stack on top of the first stack maintaining the same order as it was.

3.4 Write a C program to evaluate a given valid suffix expression. Whenever your program encounters an invalid operator, ask the user to re-enter the expression.

3.5 The text assumed a valid suffix expression in Program 3.5. Design an algorithm to validate the input suffix expression.

3.6 Modify the Program 3.5, assuming that the evaluation is to be done for a logical expression. The return value is either true or false.

3.7 Explain clearly the method to convert an infix expression to postfix form. Write a C function that does this. However, the input expression, now may contain even unary operators + and -.

3.8 Develop C functions to convert a postfix expression to infix and a prefix expression to infix expression.

3.9 Under what circumstances the overflow and underflow occur in a double stack? Discuss the merits and demerits of a double stack.

3.10 Show the stack contents after each operation for the following sequence of push and pop operations, assuming MAX = 5:

(1) Push(10)
(2) Push(20)
(3) Push(30)
(4) Pop()
(5) Push(40)
(6) Push(50)
(7) Push(60)
(8) Pop()
(9) Pop()
(10) Pop()

3.11 Assuming a double stack, show the contents of the stack after each operation as follows:

(1) Push(10,2)
(2) Push(20,1)
(3) Push(30,1)
(4) Pop(2)
(5) Push(40,1)
(6) Push(50,1)
(7) Push(60,2)
(8) Pop(2)
(9) Pop(2)
(10) Pop(1)

Chapte 4

Recursion

4.1 INTRODUCTION

Recursion is a programming technique used to solve problems more naturally. Quite often this technique is confused with the iterative method that use a `for` loop or a `while` loop or a `do-while` loop. This chapter attempts to remove this ambiguity and explains the basic concepts of recursion.

We shall solve some of the classic problems using recursive technique. Few of them are given below:

- Factorial
- Fibonacci
- GCD
- Towers of Hanoi problem
- Character string based problems

Remember to grasp the concepts of recursion, as it will be used in almost all the chapters that follow.

4.2 DEFINITION

Recursion is a process by which we define some thing in terms of itself. In Latin, **re** - means **back** and **currere** means **to run**. A procedure or function that is run over and over for a definite number of times is recursion. In other words, when a function is

The boxes *a* to *d* are the recursive calls to `fact()` until the value of *n* becomes 0. Since 0! = 1, by definition, no more recursive calls are initiated. The function does not go back to the calling routine from box-*d*. Instead, the control goes to the previous calling sequence i.e., box-*c* carrying the value obtained in box-*d* (because of return statement). In Figure 4.1(b) it is shown that value 1 is carried to box-*c* and then 2 to box-*b* and then 2 to box-*b* and 6 to box-*a* where 4 * 6 = 24 is returned to the calling program.

4.4 EXAMPLE – 2 FIBONACCI NUMBERS

Another commonly used (and some times abused!) example of a recursive function is the generation of **Fibonacci numbers**. These numbers were originated because of *leonardo Fibonacci* in 1202. These numbers are useful in finding the solution for the rabbits problem. *How many pairs of rabbits can be produced from a single pair in a year?* You can assume that no rabbits die in this one year.

The recurrence relation for the *n*th Fibonacci number can be written as,

$$fib(n) = \begin{cases} 1, \text{if } n = 0 \\ 1, \text{if } n = 1 \\ fib(n-1) + fib(n-2), \text{if } n > 1 \end{cases} \quad \ldots.(4.2)$$

When $n > 1$ the *n*th Fibonacci number is calculated by adding the previous two number. Hence,

1, 1, 2, 3, 5, 8,13, . . .

is an infinite Fibonacci series. The 6th Fibonacci number 5 is obtained by adding the previous two numbers 2 and 3, that is,

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(2) &= fib(1) + fib(0) \\ &= 1 + 1 = 2 \\ fib(3) &= fib(2) + fib(1) \\ &= 2 + 1 = 3 \\ &\vdots \quad \vdots \quad \vdots \\ &\vdots \quad \vdots \quad \vdots \\ fib\,(n) &= fib(n\text{-}1) + fib(n\text{-}2) \end{aligned}$$

This suggests that we can solve this problem of finding the *n*th Fibonacci number using recursive method - just like factorial. The algorithm appears in Algorithm 4.2.

The terminating condition for this problem is when either $n = 0$ or 1, unlike factorial which uses just one condition.