# CHAPTER 3

# PROGRAMMING AND DATA STRUCTURES

**Syllabus:** Programming and data structures: programming in C; functions, recursion, parameter passing, scope, binding; abstract data types, arrays, stacks, queues, linked lists, trees, binary search trees, binary heaps.

## 3.1 INTRODUCTION

A computer is a machine that manipulates information. Computer science is the study of how information is organised, manipulated and can be utilised. Data structure is a way of organising data in computer's memory or disk so that it can be used efficiently. Examples of several common data structures are arrays, linked lists, queues, stacks, binary trees and hash tables. Different kinds of data structures are suited to different kinds of applications, and some are highly specialised to specific tasks. Different types of data structures are used in different types of applications, which are specialised to perform the specific task. Data structures provide a means to manage large amounts of data efficiently, such as large databases and Internet indexing services. Data structures are designed to manage a large amount of data to suit a specific purpose so that it can be accessed and worked in an appropriate manner. Usually, efficient data structures are a key in designing efficient algorithms. In programming, efficient data structures are used to design efficient programs, both in terms of time and memory. In the following text, the programming part is discussed followed by data structure.

## 3.2 BASIC TERMINOLOGY

This section defines the basic terms used in computer programming.

### 3.2.1 Programming Languages

A computer performs the tasks commanded by the user. The command is given through a sequence of lines of code that are understood by the computer. The art of

writing such a code (or a program) is called programming. According to famous programmer Niklaus Wirth,

PROGRAMS = DATA + ALGORITHMS

Every programming language has two important pillars- *syntax* (refers to grammatical representation) and *semantic* (refers to the meaning). A correct program should be correct from syntactic view as well as from semantic view. The different kinds of translators check only the syntactic representation and the programmer takes care of the semantic representation. There are different generations of programming languages:

1. **First Generation/1GL/(Machine level programming):** The program is written in a sequence of 0s and 1s (binary number system). Such a language is more closer to the machine and the programs are machine dependent and difficult to debug. The only advantage is that there is no need for translator software (as the program is already in machine code) and execution is very fast.
2. **Second Generation /2GL/(Assembly level programming):** The program is written using instructions based on mnemonics and hexadecimal number system. Such a language is also machine-dependent. *Assembler* is the translator used to convert assembly level code to machine level code. The programs more reader friendly so are easier to debug.
3. **Third Generation /3GL/(High-level programming):** The program is written using English-like statements and decimal number system. Such a language is more closer to the user and the programs are machine independent. There is a need of translator (compiler/interpreter) to convert the high- level code to machine level code. Such languages are user-friendly. Such languages are *file-oriented*.
4. **Fourth Generation /4GL/(Non-procedural programming):** Such languages allow the user to specify the result instead of the describing how the result is to be obtained. All kinds of query languages belong to this generation. Such languages are *database-oriented*.
5. **Fifth Generation /5GL/(Natural Language Programming):** They are similar to 4GL and eliminates the user or programmer to learn a specific language. The language used, resembles the human speech closely. Example- CLOUT, Q & A, HAL (Human Access Language).

### 3.2.2 Classification of High-Level Languages

Depending upon their usage, the high-level languages are classified as: procedural, non-procedural and problem oriented languages.

1. **Procedural Languages:** These are those languages which follow a certain procedure. These are three different types-:
   - *Algorithmic languages:* The programmer specifies the steps (algorithm) to be followed for accomplishing a particular task. Such languages have built-in expressions, functions and procedures. For example, C, COBOL, PASCAL.
   - *Object-oriented languages:* Instead of using functions, objects are used. The data and the functions are combined in defining an object. Some of the important features are: Abstraction, Encapsulation, Polymorphism and Inheritance. For example, C++, Java.
   - *Scripting languages:* It is a kind of language that combines different components together to perform a difficult task. For example, Visual Basic script, PERL.
2. **Non-Procedural Languages:** 4GL languages are defined as non-procedural languages. These are two different types-:
   - *Functional language:* The program consists of functions, such languages are used in the field of artificial intelligence. For example, LISt Processing (LISP).
   - *Logic-based language:* It is a set of rules that are defined and the answer is retrieved using if−then rules. For example, PROgramming in LOGic (PROLOG).
3. **Problem-Oriented Languages:** Especially designed to solve certain kind of problems. These are of two different types:
   - *Numerical problems:* These contains the built-in functions, used to solve mathematical problems. For example, Mathematica.
   - *Publishing:* It contains the built-in functions, used in publishing industry. For example, LATEX.

### 3.2.3 Procedural Programming and Object-Oriented Programming

#### *3.2.3.1 Procedural Programming*

It uses a list of instructions in a stepwise manner to computer. It relies on procedures. Procedures are also known as routines or subroutines. A procedure consists of a series of computational steps that has to be carried out. Procedural programming languages are also known as top-down languages.

#### *3.2.3.2 Object-Oriented Programming*

Object-oriented programming (OOP) is a problem-solving approach in which all computations are carried

out using objects. An object is an entity of a program. It is used to perform actions and interactions with other elements of the program. These are the fundamental units of OOP, for example, a person.

The other fundamental concepts of OOP are as follows:

1. **Class:** It is a blueprint for an object. It does not define any data, rather it will define the object of the class it consists and the operations performed on the object.
2. **Abstraction:** It is used to provide essential information in a program without presenting the details. For example, in a database system, only data is shown, all the information regarding creating, storing, etc., is hidden.
3. **Encapsulation:** It is used to place data and functions at the same place.
4. **Inheritance:** It is the process of forming a new class from an existing class. The new class is known as the derived class and the existing class is known as the base class. It uses the concept of reusability of code.
5. **Polymorphism:** It is used to provide different meanings or functions to the existing operators or functions.
6. **Overloading:** It is a branch of polymorphism. When the existing operator or function is made to operate on a new data type, then it is known as overloading.

#### *3.2.3.3 Structured Programming*

A kind of programming approach that revolutionized the programming concept, proposed by E. Dijkstra'. In 1968, Dijkstra submitted the famous article in ACM Journal, highlighting the drawbacks of using the GOTO statement. He advocated that program written without the use of GOTO statement (or JUMP statement) were more stable. Nowadays, more and more programmers are following such kind of approach in writing programs. Other notable attributes of structured programming are:

1. **Top-down analysis:** A large problem is subdivided into smaller sub-problems.
2. **Modularization:** Dividing the program into small independent modules.

### 3.2.4 Data Types

It constitutes the type of values a variable can take and a set of operations that can be applied to those values. Data types in programming language can be broadly classified as shown in Fig. 3.1.

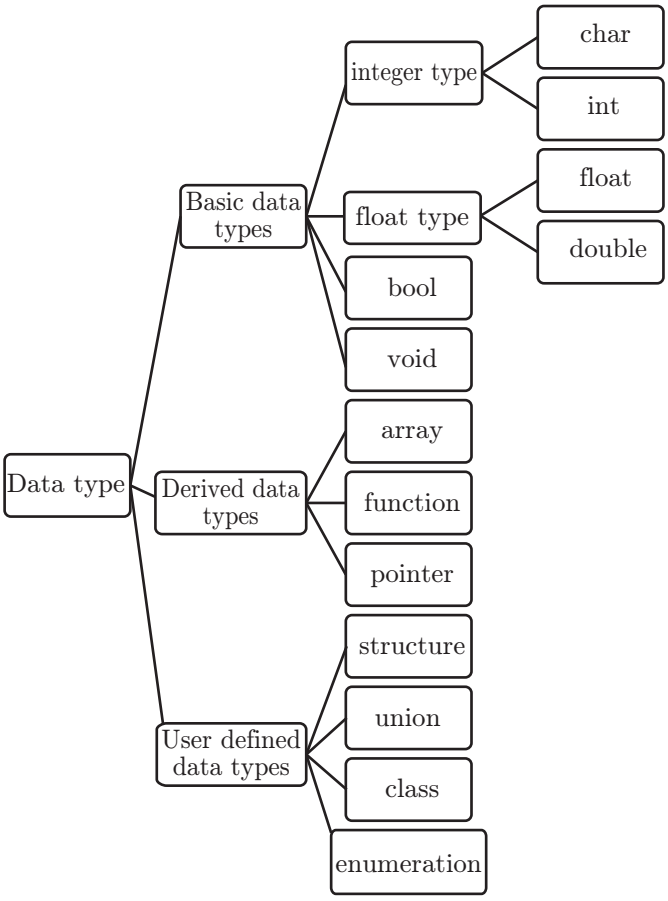



**Figure 3.1** Classification of data types in programming languages.

Table 3.1 gives detail about basic data types with its storage sizes and value ranges.

**Table 3.1** Basic data types, their storage size and value ranges

| Type | Storage Size | Value Range (Depends on Compiler Type) |
|---|---|---|
| char | 1 byte | −128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | −128 to 127 |
| int | 2 or 4 bytes | −32,768 to 32,767 or −2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |

(*Continued*)

**Table 3.1** | Continued

| Type | Storage Size | Value Range (Depends on Compiler Type) |
|---|---|---|
| short | 2 bytes | −32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | −2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |
| float | 4 bytes | 3.4E − 38 to 3.4E + 38 |
| double | 8 bytes | 1.7E − 308 to 1.7E + 308 |
| long double | 10 bytes | 3.4E − 4932 to 1.1E + 4932 |

C language is a weakly typed language so it allows implicit and explicit type conversions (Fig. 3.2). The compiler promotes each term in a binary expression to the highest precision operand.

### 3.2.5 Control Flow Statements

Programs consist of several statements and execution is not limited to a linear sequence of statements. During its execution, a program may repeat segments of code, or take decisions and branches. For that purpose, any programming language provides control flow statements that specifies the control flow of program or the order of execution of statements (Fig. 3.3).

#### *3.2.5.1 Conditional Statements*

These are used to execute a statement or a group of statements on the basis of certain conditions. These conditions are as follows:

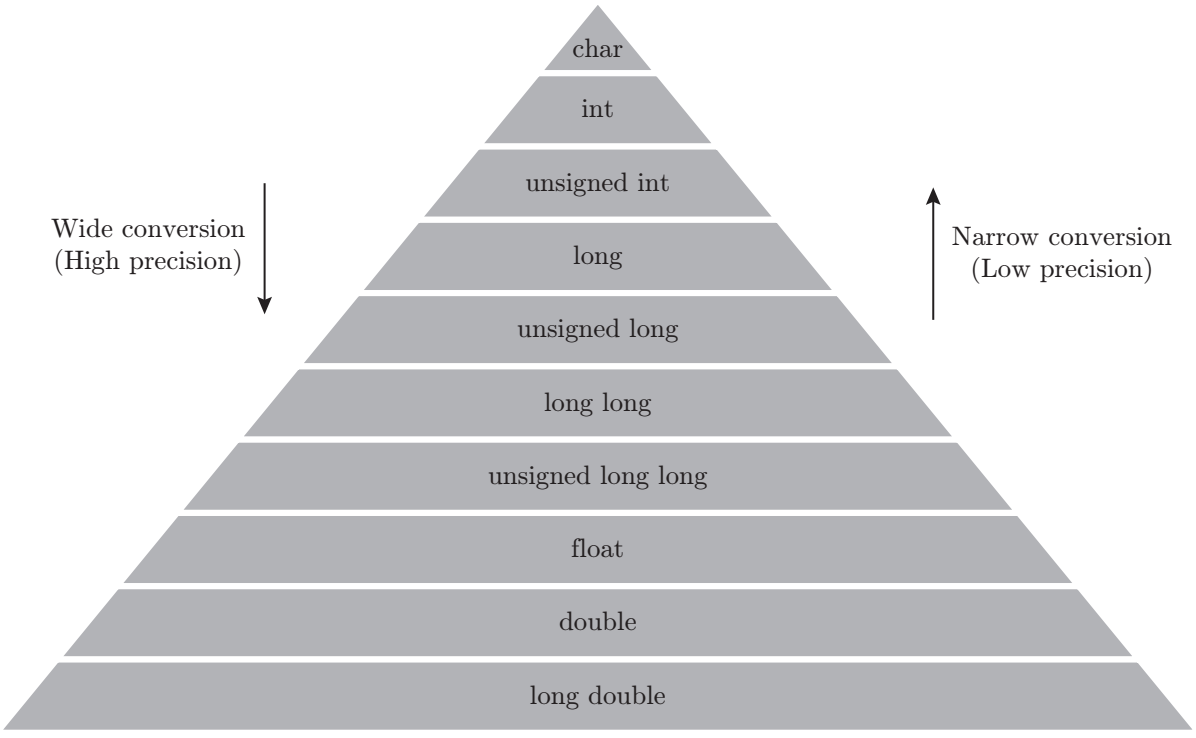



**Figure 3.2** | Type conversion in C.

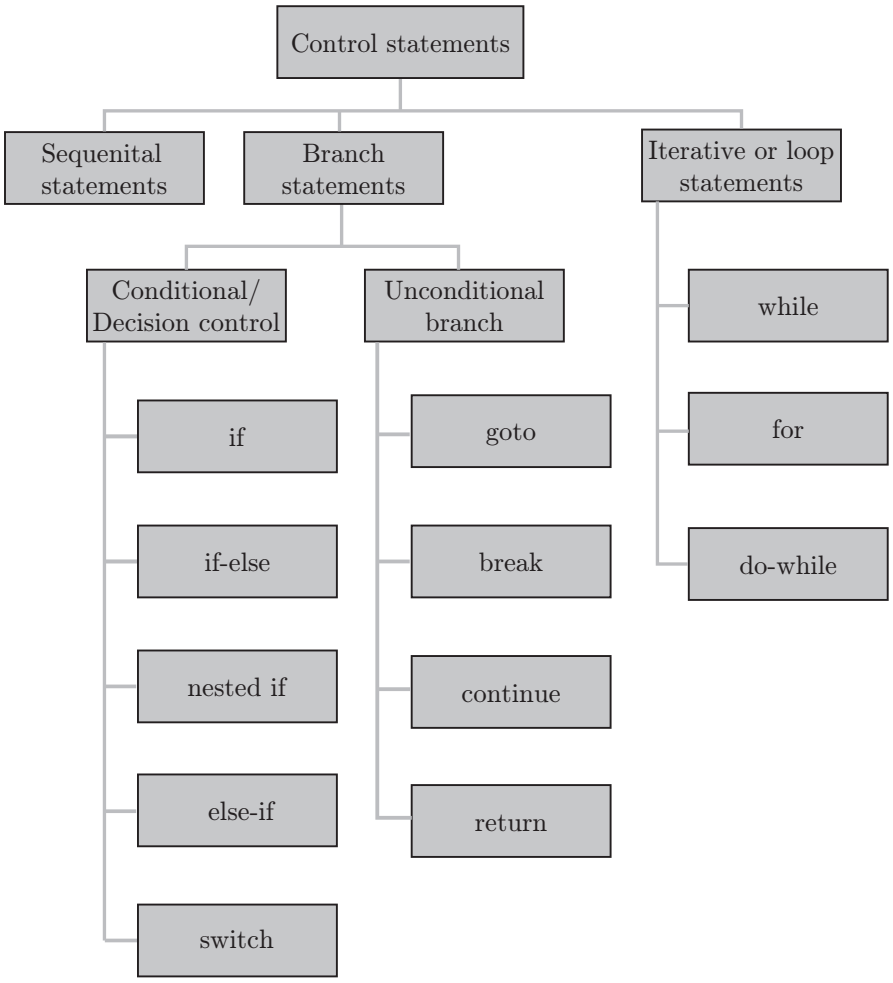



**Figure 3.3** | Control flow statements.

1. **If:** When the condition is satisfied then the statements inside the parenthesis {} will be executed.
2. **If-Else:** If the condition is true then the statements between if and else will be executed. If it is false then the statement after else will be executed. In nested if, one or more if-else block(s) lie within the parent if statement.
3. **Else-If:** If the condition is true then the statements between if and else if will be executed. If it is false then the condition in else if is checked, and if it is true it will be executed.
4. **Switch:** It is also known as matching case statements. It is used to test for equality against a list of values. It matches the value in variable with any of the case inside the switch. If it matches the case defined in the switch then that match will be executed. If none of the cases match the statement then default will be executed.

#### *3.2.5.2 Loops*

These are used to execute a block of code several number of times. The statements are executed sequentially. Loop executes a statement or a group of statements multiple times. When the execution of statements in the loop is unknown, then this concept is known as odd loop.

Loops are of the following three types:

1. **while loop:** It repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2. **for loop:** It executes a sequence of statements multiple times.
3. **do … while loop:** It is similar to while statement, except that it tests the condition at the end of the loop body.

Loops can be nested, that is, execution of one loop inside another loop.

There are various control statements used for changing execution from its normal sequence. These are as follows:

1. **break statement:** It terminates the loop or switch and transfers the execution to the statement immediately following the loop or switch.
2. **continue statement:** It skips the remainder of the loop body and immediately begins execution from the top.
3. **goto statement:** It transfers control to the labelled statement.
4. **return statement:** It transfers control from the function to the calling function.

## 3.2.6 Array

Array handles similar types of data. For example, storing marks of 100 students.

An array is a set of homogenous elements, which uses contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

### *3.2.6.1 Declaring an Array*

Array can be declared as follows:

```
type arrayName [arraySize];
```

where

1. `type` represents the type of element an array stores. For example, if array stores integer elements then type of array is 'int'.
2. `arrayName` represents the name given to the array. It can be any string.
3. `arraySize` represents the number of elements the array stores. It is showed in `[]`.

For example, an array of 10 integers can be defined as follows:

```
int arr[10]
```

Arrays are of the following two types:

1. One-dimensional arrays
2. Multi-dimensional arrays

#### Initialisation of One-Dimensional Array

Arrays can be initialised at declaration time in this source code as follows (Fig. 3.4):

```
int age[5] = {2, 4, 35, 5, 8}
```

| age[0] | age[1] | age[2] | age[3] | age[4] |
|---|---|---|---|---|
| 2 | 4 | 35 | 5 | 8 |

**Figure 3.4** Initialisation of one-dimensional array.

Note that the first element is numbered 0 and so on.

Here, the size of the array *age* is five times the size of int because there are five elements.

Starting address of an array is also known as the base address of the array. Suppose, the starting address of age[0] is 1000 and the size of int is 4 bytes. Then, the next address (address of age[1]) will be 1004, address of age[2] will be 1008 and so on.

### *3.2.6.2 Important Things to Remember in C Arrays*

Following are the important points for arrays:

1. Suppose you want to declare an array of 10 students. For example: arr[10]. Only array members from arr[0] to arr[9] are used. But, if element arr[10], arr[13], etc. are used, then the compiler may not show error but may cause fatal error during program execution.
2. The size of the array may not be defined during initialisation. For example,

   ```
   int age[] = {2, 4, 35, 5, 8};
   ```

   In this case, the compiler determines the size of the array by calculating the number of elements of an array.
3. If we do not provide value to any position in the array it will store 0 by default. For example,

   ```
   int age[5] = {2, 4, 35, 5};
   ```

   In this above case, first four places in the array will be filled by given values and the last will be filled by 0.

   ```
   int age[5]= { };
   ```

   Now array `age` is initialised with 0 at every position.

### *3.2.6.3 Address Calculation in an Array*

The address can be calculated as follows:

#### One-Dimensional Arrays

In one dimension, an array '*A*' is declared as follows:

```
A[lb... ...ub]
```

where lb is the lower bound of the array and ub is the upper bound of the array.

Suppose we want to calculate the *i*th element address, then

```
Address (arr[i]) = BA + (i – lb) * c;
```

where BA is the base address of array, lb is the lower bound of array and c is the size of each element.

### Two-Dimensional Arrays

In two dimensions, an array '*A*' is declared as follows:

```
A[lb1 ... ...ub1][lb2 ... ... ub2]
```

where $lb_1$ is the lower bound for row, $lb_2$ is the lower bound for column, $ub_1$ is the upper bound for row and $ub_2$ is the upper bound for column.

1. **Row Major Order:**

```
Address(arr[i][j]) = BA+[i-lb1)*Nc+
                     (j-lb2)]*c
```

2. **Column Major Order:**

```
Address(arr[i][j]) = BA+[j-lb2)*Nr+
                     (i-lb1)]*c
```

where BA is the base address, Nr is the number of rows = ($lb_2 - lb_1 + 1$) and Nc is the number of columns = ($ub_2 - ub_1 + 1$).

**Problem 3.1:** Consider the following array:

`a[1 ... ... 500]`

Base address = 1000

Size of each element = 3 bytes

Find address of `a[397]`.

**Solution:**

```
Address (a[397]) = BA + (i − lb) * c
                 = 1000 + (397 − 1) * 3
                 = 1000 + 396 * 3
                 = 2188
```

**Problem 3.2:** Consider the following array:

`a[50 ... ... 299][299 ... ... 500]`

Base address = 0

Size of each element = 5

Find the address of a[250][499] by row major order.

**Solution:**

```
Address(arr[i][j]) = BA+[i−lb1)*Nc
                     +(j−lb2)]*c
Address(a[250][499]) = BA+[(250−50)*201
                       +(499−299)]*5
                     = 0+(200*202+200)*5
                     = 203000
```

## 3.2.7 Function and Recursion

A function is a group of statements, which is used to perform a task and recursion is a programming technique, which is used to express operations in terms of themselves.

### *3.2.7.1 Function*

A function is a set of statements that together perform a task. Every C program has at least one function which is main(), and programs can have additional functions.

In other words, a function is a self-contained block, which takes some input and gives output after processing (Fig. 3.5). A function is known with various names such as a method or a sub-routine or a procedure, etc.

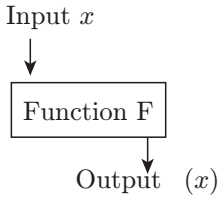



**Figure 3.5** A view of function.

**Example 3.1**

```
intavg(inta,intb,intc){
  float result;
  result = (a+b+c)/3;
  return result;
  }
```

### Parts of a Function

A function has the following parts:

1. Function prototype
2. Definition of a function
3. Function call
4. Actual and formal arguments
5. Return statement

These parts have been discussed in detail as follows:

1. **Function prototype:** Function prototype contains function declaration, which tells the compiler about a function name, return type, argument types and function calling. Function declaration is necessary before calling it. A full declaration includes the return type and the type of arguments. Function declaration will be as follows:

```
return_type function_name (parameter list);
```

For the above definition of function, the `avg()` function declaration is as follows:

```
intavg(int a, int b, int c);
```

In function declaration, the arguments are not important, only their type is required. So, the following declaration is also correct:

```
intavg(int, int, int);
```

2. **Definition of a function:** Function definition consists of the following parts:
   - *Function name*: This is the actual name of the function. It is unique. Function signature is constituted with the function name and parameter list together.
   - *Parameters*: At the time of function declaration, the parameters are passed into the function. The parameter list shows the type, value of parameter and number of arguments in the function. Parameters field is optional so that a function may contain no parameters.
   - *Function body*: The function body contains a collection of statements that defines what the task is performing by that function.
   - *Return type*: A function returns a value. The return type is that data type in which a function should return a value. Some functions perform the desired operations without returning a value. There are two possibilities, the function either returns a value or not. When the function does not return a value then the void data type used.

   In the above example, the function name is `avg`, the function has three integer-type parameters and the function calculates the average of those three numbers.
3. **Function call:** In function call, the control is transferred to the called function when the program calls the respective function. The name of the function must match with the function exactly with the name defined in the function prototype. A called function performs a defined task, and when its return statement is executed, it returns program control back to the main program. To call the function, the first step is to pass the arguments with function name, and then the return value stores in a specific manner.
4. **Actual and formal arguments:** A parameter is a special kind of variable used in a function to refer to the data provided as input to the function. These pieces of data are called arguments. Syntactically, we can pass any number of parameters to a function. Parameters are specified within a pair of parenthesis. These parameters are separated by commas (,). Parameter written in a function definition is called 'formal parameter'. Parameter written in a function call is called 'actual parameter'.
5. **Return statement:** When a return statement gets executed, then the function returns the value and transfers the control to the calling function. Execution continues in the calling function by following the remaining statements. A return statement can also return a value to the calling function.

### Types of C Functions

The following are two types of functions in C on the basis of whether it is defined by a user or not:

1. **Pre-defined function:** The function whose definition is already stored in the library of the respective language is called pre-defined function. These functions are provided in the system or particular language library. For example, in C language, printf(), scanf(), gets(), puts() are available in stdio.h header file. These functions are also called library functions or built-in functions.
2. **User-defined function:** User-defined functions are functions created by the user at the time of writing the program for modularity purpose. For example,

```
int max (int num1, int num2){
int result;
if(num1 > num2)
      result = num1;
else
      result = num2;
return result;
}
```

### Function Advantages

Function has the following advantages:

1. It provides the reusability of code in an easy manner.
2. It makes debugging and editing easier.
3. It reduces the size of a program.
4. It makes the logic of the program easier to design and understand.

## *3.2.8.2 Recursion*

When the function calls itself then it is called recursion. In C language, there is provision for the function to call itself. When using the recursion, it is necessary to have exit condition from the function; otherwise, the result will be infinite.

### Writing Recursive Functions

A recursive function has the following two parts:

1. **Base case:** It is the non-recursive solution and a stopping condition.
2. **Recursive case:** It is the recursive solution of the problem.

It is simply a specification of the general recursive function we have seen many times.

```
return_typefunction_name(Pass appropriate
  arguments)
{
 if a simple case, return the simple value
  /* base case or stopping condition*/
 else call function itself with simpler
  version of problem
}
```

**Example 3.2**

```
int sum(int n){
if(n==0)  /* base case or stopping
  condition */
return n;
else
returnn+sum(n-1); /*self-call to function
  sum() */
}
```

In this program, sum() function is called itself in else condition. If $n$ is equal to 0 then the function returns the value passing in the argument, but if $n$ is not equal to 0 then the function calls itself. Let $n = 5$ initially, then next time 4 is passed to the function and the value of the parameter is decreased by 1 until the condition is satisfied. At the end, when $n$ becomes 0, the value of $n$ is returned, which is from 5 to 1.

$$\begin{aligned}
&\text{sum}(5)\\
&= 5 + \text{sum}(4)\\
&= 5 + 4 + \text{sum}(3)\\
&= 5 + 4 + 3 + \text{sum}(2)\\
&= 5 + 4 + 3 + 2 + \text{sum}(1)\\
&= 5 + 4 + 3 + 2 + 1 + \text{sum}(0)\\
&= 5 + 4 + 3 + 2 + 1 + 0\\
&= 5 + 4 + 3 + 2 + 1\\
&= 5 + 4 + 3 + 3\\
&= 5 + 4 + 6\\
&= 5 + 10\\
&= 15
\end{aligned}$$

### Advantages of Recursion

Recursion has the following advantages:

1. Unnecessary calling of functions is avoided.
2. In recursion, the function calls anywhere in the program so it if flexible in nature.
3. In some situations, the recursion is compulsory in a programming. For example, to find the factorial of a given number.

## 3.2.8 Pointers

A pointer is a variable which stores the address of some other variables. The declaration of a pointer is done as follows:

```
Data type *variable_name;
```

**Example 3.3**

```
int *p; /* pointer to an integer */
int b = 20;
p = &b; /* address of variable b is stored
  in p */

printf ("Address of variable b is: %d", &b);
printf ("Address of variable b is: %d, p);
printf ("Value of variable b is: %d", *p);
  /* *p displays the value at stored address */
```

The operating systems may stop to access memory and cause the program to crash. To avoid such problem, pointers should always be initialized before use. These are initialized using free memory in the following way:

1. **Memory allocation to pointers:**
   ```
   int *p = malloc (sizeof (*p));
   ```
2. **De-allocating memory of pointer:**
   ```
   free (p);
   ```
3. **Pointer to pointer:**


```
int *p    /* pointer to an integer */
int **ptr  /* pointer to pointer */
int b = 20;
p = &val;   /* address of variable b is
  stored in p */
ptr = &p;   /* address of pointer is
  stored in ptr */

To print value of b:
*p = 20;
**ptr = 20;
```

### Pointer Arithmetic

1. **Incrementing a pointer:** Incrementing a pointer variable depends on data type of the pointer variable. It is generally used in array in which elements are stored at contiguous memory locations. For example,

   ```
   ptr ++ = ptr + 1 = ++ ptr = ptr + 1 *
     size_of(data type);
   (address + 1= ++ address = address++
     gives an address value)
   ```

```
#include<stdio.h>
int main(){
int *ptr=(int *)200; /*Assume integer
  takes 4 bytes*/
ptr=ptr+1;
printf("New Value of ptr : %u",ptr);
return 0;
}
```

**Output**

```
New Value of ptr: 204
```

2. **Decrementing a pointer:** Decrementing a pointer to a data variable will cause its value to be decremented by size of variable because memory required to store a variable (integer) varies from compiler to compiler.Example

```
ptr-- = ptr - 1 = --ptr = ptr - 1 *
  size_of(data type);
(address - 1= --address = address--
  gives an address value)
```

3. **Adding an integer value with pointer:** In C Programming, it is legal to add any integer number to pointer variable.

```
ptr + n = (ptr) + (n * size of data type);
```

```
#include<stdio.h>
int main()
{
int *ptr=(int *)100;
ptr=ptr+3;
printf("New Value of ptr : %u",ptr);
return 0;
}
```

**Output**

```
New Value of ptr : 112
```

4. **Subtracting an integer value from pointer:** In C programming, it is legal to subtract any integer number from pointer variable.

```
ptr - n = (ptr) - (n * size of data type)
```

```
#include<stdio.h>
int main()
{
int *ptr=(int *)100;
ptr=ptr - 3;
printf("New Value of ptr : %u",ptr);
return 0;
}
```

**Output**

```
New Value of ptr : 88
```

5. **Subtracting two pointers:** It means subtracting two pointers and it gives total number of two objects between them.

```
Actual Result = (ptr2 - ptr1) / Size
  of Data Type
```

```
Suppose the address of variable n is
  200 and integer takes 4 bytes.
```

```
#include<stdio.h>
int main()
{
    int n, *ptr1, *ptr2;
    ptr1 = &n;
    ptr2 = ptr1 + 2;
    printf("Difference is: %d",ptr2 -
      ptr1);
    return 0;
}
```

**Output**

```
Difference is 2
ptr1 = 200
ptr2 = 200 + 2 * 4 = 208
ptr2-ptr1 = (208-200)/4 = 2
Both pointers numerically differ by 8
  and technically by 2 objects
```

### *3.2.8.1 Dangling Pointer*

Dangling pointer arises when an object is deleted or de-allocated, without modifying the value of pointer, so that the pointer still points to the memory location of the de-allocated memory. In other words, a pointer pointing to a non-existing memory location is known as a dangling pointer. For example,

```
void main ()
  {
      int *p;
          {
               int i=10; //This is a Block.
               p=&i;//Variable i is local
                 so i's lifetime is only
                 within the block.
          }
      printf ("%d", *p); //p will point to
        de-allocated memory.
      *p=*p+100;
      printf ("%d",*p);
  }
```

This program is perfectly running and giving output as 100 200. Actually this should not have happened because in this program p points to a location **i** which has been deleted from the memory so **p** becomes a dangling pointer but still the program is running.

Sometimes, the dangling pointer constitutes an undefined behaviour as may be the reason that memory has not been overwritten by anything else. For example,

```
int *demoFun()//returns the address of
  variable i
  {
      int i=10;
      return &i;
  }
int main()
 {
  int *p;
  p=demoFun();   /*function returning
    address is assigning to pointer
    variable.*/
   printf("%d",*p);      //10
   *p=*p+20;
   printf("%d",*p);      // Garbage Value
 }
```

**Output**

```
- 10
Some Garbage Value
```

According to the program, variable **i** have been deleted after completion of `demoFun()` function but still the pointer is able to access the memory location of **i** because this memory is not overwritten by another variable. It must be the same reason that dangling pointer constitutes an undefined behaviour. But when we perform another operation, complier gives some garbage value because base pointer **p** does not get any data to that particular location.

### 3.2.9 Parameter Passing

There are various types of parameter-passing techniques in programming languages. The parameter passed is based on the program requirement. According to the parameter passing, the passing parameter depends on the way the calling function calls the called function.

The C programming language supports the following two types of parameter-passing techniques:

1. Call by value
2. Call by reference

#### *3.2.9.1 Call-by-Value Parameter-Passing Technique*

When we call a function, we pass the parameter to the called function and it is by default passed by the value of variable in the calling function. This is called 'call by value'.

**Example 3.4**

```
void exchange(int x, int y);
main()
{
int a = 10, b = 20;
printf("\na = %d b = %d", a, b);
exchange(a, b);
printf("\na = %d b = %d", a, b);
}

exchange(int x, int y)
{
int t;
t = x;
x = y;
y = t;
printf("\nx = %d y = %d", x, y);
}
```

Before calling the function exchange(), values of $a$ and $b$ are 10 and 20, respectively. After calling exchange(), $x$ and $y$ also receive 10 and 20, respectively. In exchange(), we are interchanging values of $x$ and $y$. So, the new values of $x$ and $y$ are 20 and 10, respectively. After successful execution of the function exchange(), control transfers to the main function, but in the main function, values of $a$ and $b$ remain unchanged, that is, 10 and 20, respectively.

So, the output of the above program is as follows:

```
a=10 b=20
x=20 y=10
a=10 b=20
```

In the call-by-value mechanism, only the formal argument (variables receiving value in called function from calling function, that is, $x$ and $y$ in the above program) gets changed and do not reflect in actual argument (variable in function call, that is, $a$ and $b$ in function call to exchange(a, b)).

#### *3.2.9.2 Call-by-Reference Parameter-Passing Technique*

In call by reference, the parameter is passing through the actual address of the value. The address of the variable never changes throughout the execution of the program. There is a possibility to access and modify the value of the variable by using a pointer. In case of any changes in the parameter inside the function, there is a change of the actual parameter. For example, the declaration of a variable is as follows:

`int x = 20;` which tells the compiler:

1. To allocate 2-byte memory location on stack frame of that particular function in which it is declared.
2. To hold value of type integer and also associate this address with name $x$.
3. This memory location is initialised with value 20.

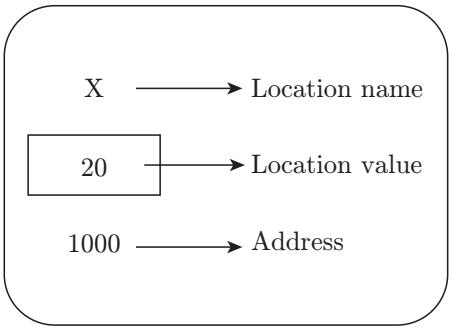



**Figure 3.6** | Memory map in call by reference.

The selection of address for a particular variable is compiler dependent and is always a whole number. In Fig. 3.6, the value 20 can be accessed by using variable name $x$ and address 1000. The symbol '&' is used to represent the 'address of'.

For example, expression &x in the above case returns address of variable $x$, which happens to be 1000.

An address is always positive, we use operator %u for unsigned integer. *' is a special operator in C used as 'value at address'. It is also called indirection operator.

For example, *(&x), which means value at address(&x), that is, value at address (1000). The example given below returns the address of a variable, the value of the variable and the value of variable using indirection operator.

**Example 3.5**

```
#include<stdio.h>
void main( )
{
int x = 20;
printf("\nAddress of x = %u", &x);
printf("\nValue of x = %d", x);
printf("\nValue of x = %d", *( &x ));
}
```

**Output**

```
Address of x = 1000
Value of x = 20
Value of x = 20
```

```
/* Call by Reference Parameter passing
  technique */
#include<stdio.h>
void exchange(a,b);
void main( )
{
int a = 10, b = 20;
printf("\na = %d b = %d", a, b);
exchange(&a, &b);
printf("\na = %d b = %d", a, b);
}
void exchange(int *x, int *y)
{
int t;
t = *x;
*x = *y;
*y = t;
}
```

**Output**

```
a = 10 b = 20
a = 20 b = 10
```

Consider the declaration of a variable. `int *x;` does not mean that $x$ is going to hold integer value, rather it simply means it is going to hold address (memory location) of integer type of variable. We are passing address of variables $a$, $b$ to function exchange(), so while receiving we are using two pointers to integer.

Generally, we use call-by-value mechanism if we do not need the changes made in calling function to reflect in the called function. But if we require the changes to reflect in the called function, we use the call-by-reference mechanism.

Also, if we want to return more than one value to the called function from the calling function, we go for call-by-reference value, which is not possible ordinarily.

**Problem 3.3:** Find the output of the following programs

(a)

```
#include<stdio.h>
void main()
{
   int a=3, *b;
   b = &a;
   printf("%d\n", a**b*a+*b);
}
```

(b)

```
#include<stdio.h>
int main()
{
    int a=10, *b, *c;
    b=&a; /* Assume address of a is 100
      and integer is 4 byte size */
    c=b;
    *b++=*c++;
    a++;
    printf("a=%d, b=%d, c=%d\n", a,
      b, c);
    return 0;
}
```

(c)

```
#include<stdio.h>
int main()
{
   char s[10] = "Point";
   char *const p=s;
   *p='J';
   printf("%s\n", s);
   return 0;
}
```

**Solution:**

The outputs of the programs are:

(a) `30` (b) `a=11, b=104, c=104` (c) `Point`

### 3.2.10 Structures and Unions

Structures are used to represent a record (e.g. library). A union is a special data type in C, which is used to store different data types in the same memory location. Table 3.2 shows the comparison of the structures and union.

**Table 3.2** Comparison between structure and union

| Structure | Union |
|---|---|
| 1. To define a structure, the '`struct`' keyword is used. | 1. To define union, the '`union`' keyword is used. |
| 2. When a structure variable is created, the compiler allocates memory for this variable equal to the sum of size of structure elements. | 2. When a union variable is created, the compiler allocates memory for that variable equal to the maximum size of union element. |
| 3. Each structure element within a structure is assigned to unique memory location. | 3. Individual members of the union share the memory allocated. |
| 4. More than one structure element can be initialised at once. | 4. Only one member can be initialised at a time. |
| 5. Individual members can be accessed at a time. | 5. Only one member can be accessed at a time. |

### 3.2.11 Enumerated Data Types

Enumerated data types assume values which are previously declared. For example,

```
enum month {jan, feb, mar, apr, may, jun,
  jul, aug, sep, oct, nov, dec};
enum month this_month;
this_month = apr;
```

### 3.2.12 Scoping

The scope is a part of a program where the identifier may have direct access, and beyond that scope, a variable cannot be accessed. In programming, variables can be declared in the following three ways:

1. Inside the main function
2. Outside the main function
3. Function parameter definition

#### *3.2.12.1 Local Variables*

A local variable is that variable which is declared in the body of the main function. It can be used only by the function in which it declares. Other function has no information about local variable. Formal parameter of a function is also treated as a local variable to that function. In the following example, num, sum, $r$ are local variables.

**Example 3.6**

```
#include<stdio.h>
int main(){
/* Local Variable Declaration*/
int num,sum=0,r;
printf("Enter a number: ");
scanf("%d",&num);
 while(num){
      r=num%10;
      num=num/10;
      sum=sum+r;
  }
  printf("Sum of digits of number: %d",sum);
 return 0;
}
```

Variables num, sum and $r$ are local to the function main(), and cannot be accessible outside the main function.

#### *3.2.12.2 Global Variables*

Global variables are those variables which are defined outside the body of the main function. The global variables will hold their value throughout the lifetime of the program. They can be accessed inside any of the functions defined for the program. Following is an example using local variable and global variable.

**Example 3.7**

```
#include<stdio.h>
/* Global Variable Declaration*/
int r;
int main(){
/* Local Variable Declaration*/
 int num,sum=0;
 printf("Enter a number: ");
  scanf("%d",&num);
  while(num){
      r=num%10;
      num=num/10;
      sum=sum+r;
  }
  printf("Sum of digits of number: %d",sum);
  return 0;
}
```

Variables num, sum are local to the function main(), and cannot be accessible outside the main function. But the variable *r* is declared outside the main as a global variable, it can be accessed by any function in this program.

If local variable and global variable have the same name, then preference will be given to the value of the local variable inside the function.

**Example 3.8**

```
#include<stdio.h>
/* Global Variable Declaration*/
int a=10;
int main(){
/* Local Variable Declaration*/
  int a=5,b=20,sum;
  sum= a+b;
  printf("Sum of number: %d",sum);
 return 0;
}
```

In the above example, variable '*a*' is declared as global and local both. And variable '*b*' is declared as local to the main function. When the sum of '*a*' and '*b*' are calculated by a compiler then priority is given to that '*a*' which is declared as local. So the program will print 25 as the sum of the variables.

#### *3.2.12.3 Storage Classes*

A storage class defines the scope and visibility of variables and functions in C program. The following four types of storage classes are available:

1. Register
2. Auto
3. Static
4. Extern

Storage, default value, scope and life of these four storage classes are given in Table 3.3.

**Table 3.3** Storage, default value, scope and life of the storage classes

| Class | Auto | Register | Static | Extern |
|---|---|---|---|---|
| Storage | Memory | CPU registers | Memory | Memory |
| Default | Garbage value | Garbage value | Zero | Zero |
| Scope | Local to block | Local to block | Local | Global |
| Life | Until block | Until block | Persists between different function calls | Till the end of program |

### 3.2.13 Binding

It is an association of an attribute with a program component in a program. For example, the data type of the value of a variable is an attribute, which is associated with the variable name. The binding time for an attribute is the time at which the binding occurs.

In C programming, the binding time for a variable type is when the program is compiled, but the value of the variable is not bound until the program executes.

#### *3.2.13.1 Early Binding*

In early binding, the linker copies the referenced module into the executable image of the program at compilation/link time. The referenced module is a part of the executable image.

1. **Advantages of early binding:**
   - It is simple and the best performer.
   - It is at least twice as fast as late binding.
   - The executable is stand alone.
2. **Disadvantages of early binding:**
   - The executable image is large.
   - If the referenced module changes then each executable must be re-linked.

- If there is more than one executable using the module at the same time, then multiple copies of it are used in memory.

#### *3.2.13.2 Late Binding*

In late binding, the linker copies only a stub code for the referenced module into the executable image. It is also known as dynamic binding. The stub code loads the referenced module into memory at load/run time.

1. **Advantages of late binding:**
   - The size of the executable image is small.
   - On changing the referenced module, re-link of the executable is not required.
   - Most operating systems can share the referenced module.
2. **Disadvantages of late binding:**
   - The executable is dependent on the shared library at runtime.
   - Since the module is shared, it must be re-entrant and thread safe.

### 3.2.14 Abstract Data Types

An abstract data type is an object with a specification of the components. It is independent of implementation details.

Examples:

1. **Stack:** Operations such as push an item onto the stack, pop an item from the stack, ask if the stack is empty; implementation may be as array or linked list.
2. **Queue:** Operations such as add to the end of the queue, delete from the beginning of the queue, ask if the queue is empty; implementation may be as array or linked list or heap.
3. **Search structure:** Operations such as insert an item, ask if an item is in the structure and delete an item; implementation may be as array, linked list or tree.

## 3.3 STACK

A stack is an ordered collection of items in which the item is added and removed in order. In stack, the element is added on the top of the stack. Only one element, which is placed on the top, is removed from the top of the stack. The stack is called last-in first-out (LIFO) structure. Stack can be implemented by linked list and array.

The basic operations on stack are as follows

1. **PUSH:** To add an element to the stack.
2. **POP:** To remove an element from the stack.

### 3.3.1 PUSH Operation on Stack

When we try to add elements to the stack then the operation is called PUSH operation on stack (Fig. 3.7). The element is placed only on the top of the stack and then the stack position gets increased by 1.

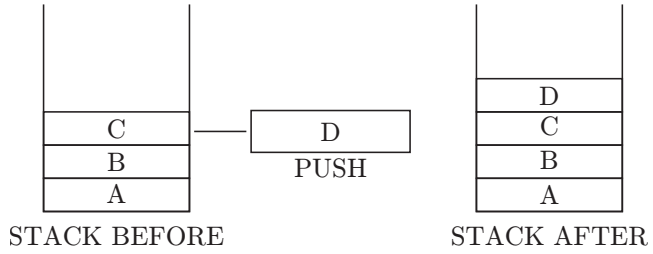



**Figure 3.7** | PUSH operation on stack.

#### *3.3.1.1 Code for PUSH Operation*

```
void push(int token)
{
char a;
if(top==MAX-1)
    {
printf("Stack full");  → is Full Condition
return;                   check
    }
do
    {
printf("\nEnter the token to be inserted:");
scanf("%d",&token);
top=top+1;
stack[top]=token;
printf("do you want to continue insertion
  Y/N");
       a=getch();
    }
while(a=='y');
}
```

### 3.3.2 POP Operation on Stack

Whenever we try to remove an element from the stack then the operation is called POP operation on stack (Fig. 3.8).

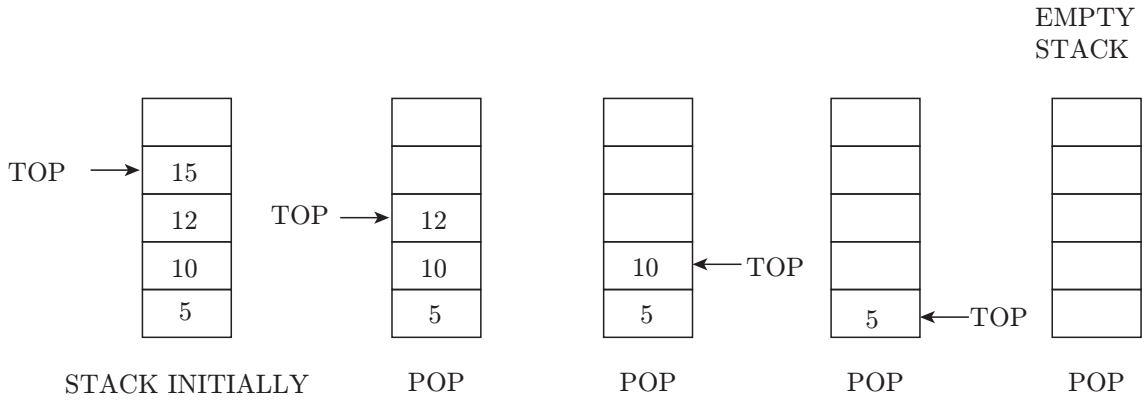



**Figure 3.8** | POP operation on stack.

#### *3.3.2.1 Code for POP Operation*

The following is the code of POP function in which we use the return statement to send the result back to the calling function

```
int pop()
{
int t;
if(top==-1)
     {
printf("Stack empty");
return -1;
     }
     t=stack[top];
top=top-1;
return t;
}
```

→ Is Empty Condition check

### 3.3.3 Application of Stack

Following are the applications on stack:

1. **Parsing:** Stacks are used by compilers to check the syntax of a program and for generating executable code.
2. **Reversing a list:** Stack can be used for reversing a list.
3. **Calling function:** When a function is called all local storage for the function is allocated on the stack and return address is also stored on the stack.
4. **Recursive function:** Stack can be used for implementing recursive functions.
5. **Expression conversion:**
   - *Infix*: An infix expression is one in which operators are located between their operands. This is how we are accustomed to writing expressions in standard mathematical notation.
   - *Postfix*: In postfix notation, the operator immediately follows its operands.
   - *Prefix*: In prefix notation, operands immediately follow operator.

#### Infix to Postfix Conversion

Algorithm for infix to postfix conversion:

1. You create a stack and an output string.
2. Then you read the infix string one token at a time, each token is either an identifier or an operator:
   - If the token is an identifier (e.g. a variable or a constant), then you append it onto the end of the output string.
   - If the token is an operator, then you compare its precedence with the precedence of the operator on the top of the stack.
     (a) If the operator on the top of the stack has lower precedence, then push the new operator onto the stack.
     (b) Else, pop the operator from the top of the stack and append it to the output string. Then push the new operator onto the stack.
     (c) Otherwise,
       (i) If the token is an open parenthesis, then push it onto the stack.
       (ii) If the token is a close parenthesis, then pop and append the operators from the stack until the open parenthesis is popped. Discard both parentheses.

**Problem 3.4:** Convert the given expression into postfix.

$$\text{Expression} = (A + B) * (C - D)$$

**Solution:**

| Expression | Stack | Output | Comment |
|---|---|---|---|
| $(A + B) * (C - D)$ | Empty | | Initial |
| $A + B) * (C - D)$ | ( | | Push |
| $+ B) * (C - D)$ | ( | $A$ | Print |
| $B) * (C - D)$ | (+ | $A$ | Push |
| $) * (C - D)$ | (+ | $AB$ | Print |
| $* (C - D)$ | | $AB+$ | Pop until ( and print |
| $(C - D)$ | * | $AB+$ | Push |
| $C - D)$ | *( | $AB+$ | Push |
| $- D)$ | *( | $AB + C$ | Print |
| $D)$ | *(− | $AB + C$ | Push |
| ) | *(− | $AB + CD$ | Print |
| End | * | $AB + CD-$ | Pop until ( |
| End | Empty | $AB + CD-$ | Pop every element |

### Infix to Prefix Conversion

Algorithm for infix to prefix conversion:

1. Reverse the given infix expression.
2. Make every '(' as ')' and every ')' as '('.
3. Apply algorithm infix to postfix on the expression that comes from step 2.
4. And finally reverse the output of step 3.

**Problem 3.5:** Convert the given expression into prefix:

Expression = $(A + B\hat{}C) * D + E\hat{}5$

**Solution:**

**Step 1:** Reverse the infix expression.

$5\hat{}E + D*)\ C\hat{}B + A\ ($

**Step 2:** Make every '(' as ')' and every ')' as '('

$5\hat{}E + D * (C\hat{}B + A)$

**Step 3:** Convert expression to postfix form.

| Expression | Stack | Output | Comment |
|---|---|---|---|
| $5\hat{}E + D * (C\hat{}B + A)$ | Empty | − | Initial |
| $\hat{}E + D * (C\hat{}B + A)$ | Empty | 5 | Print |
| $E + D * (C\hat{}B + A)$ | $\hat{}$ | 5 | Push |
| $+D * (C\hat{}B + A)$ | $\hat{}$ | $5E$ | Push |

(*Continued*)

**Table** Continued

| Expression | Stack | Output | Comment |
|---|---|---|---|
| $D * (C \wedge B + A)$ | + | $5E\wedge$ | Pop and push |
| $*(C \wedge B + A)$ | + | $5E \wedge D$ | Print |
| $(C \wedge B + A)$ | +* | $5E \wedge D$ | Push |
| $C \wedge B + A)$ | +*( | $5E \wedge D$ | Push |
| $\wedge B + A)$ | +*( | $5E \wedge DC$ | Print |
| $B + A)$ | +*($\wedge$ | $5E \wedge DC$ | Push |
| $+ A)$ | +*($\wedge$ | $5E \wedge DCB$ | Print |
| $A)$ | +*(+ | $5E \wedge DCB\wedge$ | Pop and push |
| ) | +*(+ | $5E \wedge DCB \wedge A$ | Print |
| End | +* | $5E \wedge DCB \wedge A+$ | Pop until '(' |
| End | Empty | $5E \wedge DCB \wedge A+*+$ | Pop every element |

**Step 4:** Reverse the expression.

$+*+A \wedge BCD \wedge E5$

## 3.4 QUEUE

Queue is a special abstract data type storage structure. The access to elements in a queue is restricted, unlike arrays. Enqueue and dequeue are two main operations used for the insertion of element in a queue and removal of an element from a queue, respectively. An item can be inserted at the end known as rear of the queue and removed from the end known as front of the queue. It is also called a first-in first-out (FIFO) list.

It has the following five properties:

1. **Capacity:** It stands for the maximum number of elements a queue can hold.
2. **Size:** It stands for the current size of the queue.
3. **Elements:** It is the array of elements.
4. **Front:** It is the index of the first element (the index at which the element has been removed).
5. **Rear:** It is the index of the last element (the index at which the element has been inserted).

### 3.4.1 Basic Operations on Queue

Queue has the following basic operations:

1. **Enqueue:** Inserts item $x$ at the rear of the queue $q$. Queue can be declared as follows:

```
voidEnqueue(queue, x)
{
  if(front==0&& rear==MAX-1)
{
    printf("\n Queue Overflow ");
}
  elseif(front==-1&&rear==-1)
  {
      front=rear=0;
      queue[rear]=x;

  }
  else
  {
      rear++;
      queue[rear]=x;
  }
}
```

2. **Dequeue:** Removes the front element from $q$ and returns its value. It can be declared as follows:

```
voidDequeue(queue)
{
  intx;
  if(front==-1)
  {
      printf("\n Underflow");
  }
  x=queue[front];
  if(front==rear)
{
     front=rear=-1;
}
else
  {
    if(front==MAX-1)
      front=0;
```

```
        else
          front++;
           printf("\n The deleted element
             is: %d", x);
    }

}
```

### 3.4.2 Types of Queue

Queue is of the following types:

1. **Simple queue or linear queue:** In this data structure, operations, such as, insertion and deletion occurs at the rear and front of the list, respectively (Fig. 3.9).

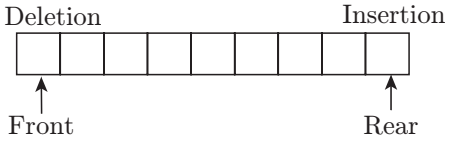



**Figure 3.9** | Simple queue.

2. **Circular queue:** A circular queue is a queue where all nodes are treated as circular (Fig. 3.10). The first node follows the last node.

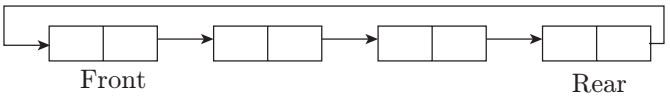



**Figure 3.10** | Circular queue.

3. **Priority queue:** It contains items which have some preset priority (Fig. 3.11). When an element has to be removed from a priority queue; the item with the highest priority is removed first.

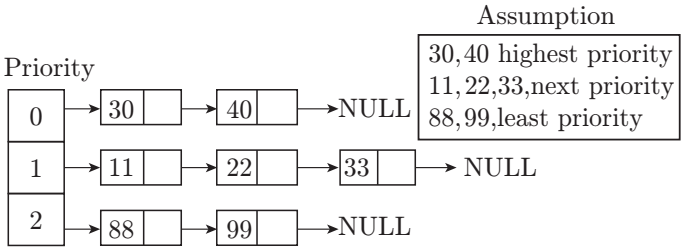



**Figure 3.11** | Priority queue.

4. **Dequeue (double-ended queue):** In this, insertion and deletion occur at both the ends, that is, front and rear of the queue (Fig. 3.12).

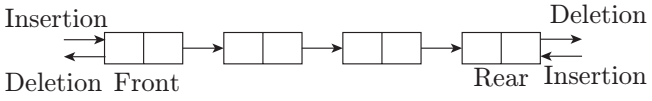



**Figure 3.12** | Double-ended queue.

### 3.4.3 Applications of Queues

Queues have many applications in computer systems:

1. Jobs in a single-processor computer
2. Print spooling
3. Information packets in computer networks

## 3.5 LINKED LIST

Linked list is a dynamic data structure whose length can be increased or decreased at runtime. Linked list basically consists of memory blocks that are located at random memory locations. Node of a linked list consists of two parts, one is value or data and the other is pointer to the next node.

### 3.5.1 Basic Operations

Basic operations of a singly-linked list are as follows:

1. **Create:** Creates an empty node of linked list.
2. **Insert:** Inserts a new element at the end of the list.
3. **Delete:** Deletes any node from the list.
4. **Find:** Finds any node in the list.
5. **Print:** Prints the list.

Functions of these operations are listed as follows:

1. **Insert:** It takes the start node and data to be inserted as arguments. The new node is inserted at the end of the linked list, till it reaches the last node. Allocate the memory for the new node and put data in it. Store the address in the next field of the new node as NULL.
2. **Delete:** It takes the start node (as pointer) and data to be deleted as arguments. Reach the node for which the node next to it has to be deleted.
   - If that node points to NULL (i.e. pointer → next = NULL) then the element to be deleted is not present in the list.
   - Else, the pointer points to a node, whose next node has to be removed, declare a temporary node (temp), which points to the deleted node. Store the address of the node next to the temporary node in the next field of the node pointer (pointer → next = temp → next).

   Break the link of the node which is next to the pointer (which is also temp). Function free() will deallocate the memory.
3. **Find:** It takes the start node (as pointer) and data value of the node (key) to be found as arguments. The first node is dummy node, so start with the second node. Iterate through the entire linked list and search for the key. Until next field of the pointer is equal to NULL, check if pointer → data = key.

- If the condition holds, then the key is found.
- Else, move to the next node and search (pointer = pointer → next).
- If key is not found return 0, else return 1.

4. **Display:** Function takes the start node (as pointer) as an argument. If pointer = NULL, then there is no element in the list. Else, print the data value of the node (pointer → data) and move to the next node by recursively calling the print function with pointer → next sent as an argument.

### 3.5.2 Linked List Implementation

Linked list is implemented as follows:

1. **Node Structure of Linked List:**

```
structnode
{
    int data;
    structnode *next;
}*Start;
```

2. **Insert Node in Linked List:**

```
void insert(Start,x)
{
    structnode *temp,*New_Node;
    New_Node=(structnode *)
      malloc(sizeof (structnode));
    New_Node->data=num;
New_Node->next=NULL;
    temp=Start;
    if(temp==NULL)
    {
    Start=New_Node;
    exit(1)
    }
    else
    {
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    Start=New_Node;
    }
}
```

3. **Delete Node in Linked List:**

```
int delete(int num)
{
    structnode *temp, *prev;
    temp=Start;
    while(temp!=NULL)
    {
    if(temp->data==num)
    {
        if(temp==Start)
        {
        Start=temp->next;
        free(temp);
        return 1;
        }
        else
        {
        prev->next=temp->next;
        free(temp);
        return 1;
        }
    }
    else
    {
        prev=temp;
        temp=temp->next;
    }
    }
    return 0;
}
```

4. **Find a Node in Linked List:**

```
int find (Struct node *Start,x)
{
struct node *temp;
temp=start;
if(temp==NULL)
{
printf("List is empty");
exit();
}
else
{
while(temp->data!=x && temp->
  next!=NULL)
{
temp=temp->next;
}
if(temp->data==x)
{
printf(" Node is found at address
  %u",temp);
return 1;
}
else
{
printf(" Node is not found ");
return 0;
}
}}
```

5. **Display Linked List:**

```
void display(start)
{
    r=start;
    if(r==NULL)
    {
    return;
    }
```

```
    while(r!=NULL)
    {
    printf("\n%d ",r->data);
    r=r->next;
    }
    printf("\n");
}
```

## 3.6 TREES

A tree is a collection of nodes and edges. When we connect each node with the help of edges then it constructs a tree. The following has the properties of a tree:

1. Starting node is called the root node.
2. Except the root node every other node($n$) in a tree is surely connected by an edge from another node($y$), the other node($y$) is called the parent node of $n$. There is unique path from start node to end node.

### 3.6.1 Binary Tree

A binary tree is a tree data structure. In a binary tree, any node in the tree can have at most two children. Binary tree are used to construct binary search trees and binary heap. An example of a binary tree is shown in Fig. 3.13.

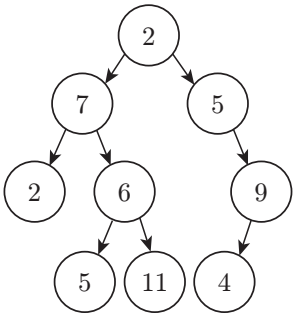



**Figure 3.13** | Binary tree.

1. **Vocabulary and definitions:** We will formally define a tree and its components.
2. **Node:** A node is an important part of a tree. It can have a name, which we call the 'key'. It may also have additional information.
3. **Edge:** An edge is another important part of a tree. It connects two nodes to show a relationship between them. Every node (except the root) is connected with exactly one incoming edge from another node. Each node may have several outgoing edges.
4. **Root:** The root of the tree does not have incoming edges.
5. **Path:** A path is an ordered list of nodes, which are connected by edges. For example, $2 \rightarrow 7 \rightarrow 6 \rightarrow 11$ is a path.
6. **Children:** It is a set of nodes ($S$), which have incoming edges from the same node. In Fig. 3.13, 2 and 6 are the children of node 7.
7. **Parent:** It is the parent of all the nodes that are connected to the outgoing edges. In Fig. 3.13, node 5 is parent of node 9.
8. **Sibling:** The nodes in the tree having the same parent are said to be siblings. The nodes 2 and 6 are siblings in the tree shown in Fig. 3.13.
9. **Subtree:** It is a set of nodes and edges comprised of a parent and all the descendants of that parent.
10. **Leaf node:** It is a node that has no children. For example, node 5 and node 11 are the leaf node in Fig. 3.13.
11. **Level:** The level of a node ($n$) is the number of edges on the path from the root node to ($n$). For example, the level of node 4 in Fig. 3.13 is three.
12. **Height:** It is equal to the maximum level of any node in the tree. The height of the tree in Fig. 3.13 is three.

### 3.6.2 Types of Binary Trees

Binary trees are of the following types:

1. **Strictly binary tree:** Figure 3.14 shows the Strictly binary tree. It has the following properties:
   - If in a tree, non-leaf nodes have exactly two children then that tree is called strictly binary tree.
   - In strictly binary tree, every leaf nodes have degree 0 and every non-leaf nodes have degree 2.
   - A strictly binary tree with $n$ leaf nodes has $(2n-1)$ total number of nodes. Thus, strictly binary tree always have odd number of nodes.

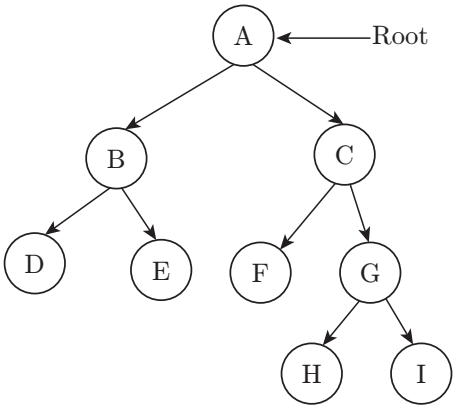



**Figure 3.14** | Strictly binary tree.

2. **Complete binary tree:** Figure 3.15 shows the strictly binary tree. It has the following properties:

- A complete binary tree of depth $d$ is strictly a binary tree having all the leaves at level $d$.
- The total number of nodes in a complete binary tree of depth $d$ equals $(2^{d+1} - 1)$.

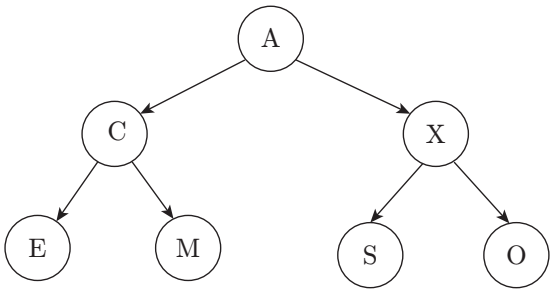



**Figure 3.15** | Complete binary tree.

3. **Almost complete binary tree:** A binary tree of depth $d$ is an almost complete binary tree (see Fig. 3.16) if:
   - Each leaf in the tree is at either level $d$ or level $d - 1$.
   - For any node $n_d$ in the tree with a right descendant at level $d$, all the left descendants of $n_d$ that are leaves are also at level $d$.
   - An almost complete binary tree with $N$ leaves that is not strictly binary has $2N$ nodes.

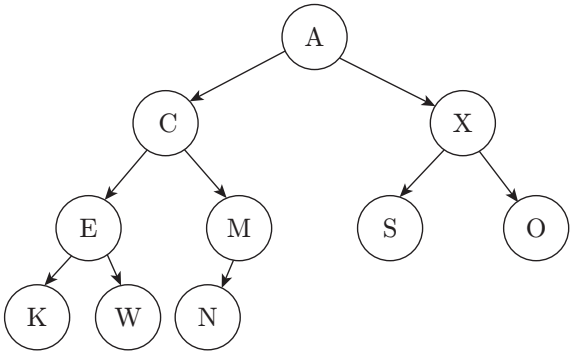



**Figure 3.16** | Almost complete binary tree.

### 3.6.3 Array Representation of Binary Trees

An array can be used to store a binary tree using the following mathematical relationships.

If root data is stored at index $n$, then left child is stored at index $2*n$ and right child is stored at index $2*n + 1$.

Figure 3.17 demonstrates the storage representation.

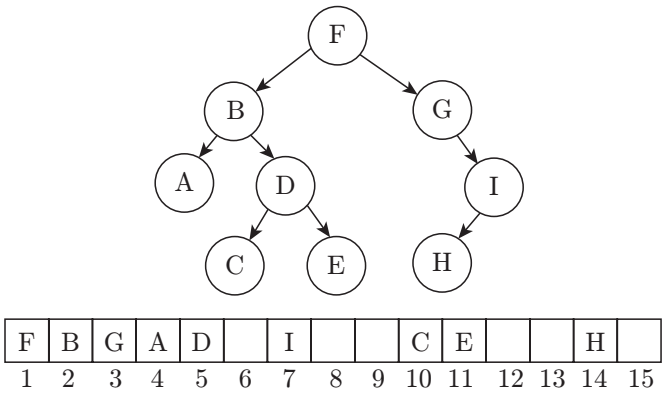



**Figure 3.17** | Storage representation.

### 3.6.4 Tree Applications

The following are common uses of a tree:

1. Manipulate hierarchical data
2. Make information easy to search
3. Manipulate sorted lists of data
4. As a workflow for compositing digital images for visual effects
5. Router algorithms

### 3.6.5 Binary Search Tree

Binary search tree is a node-based binary tree data structure with the following properties:

1. The left subtree contains the nodes with keys less than the node's key.
2. The right subtree contains the nodes with keys greater than the node's key.
3. Both the right and left subtree should also be binary search tree.
4. There should not be any duplicate nodes.

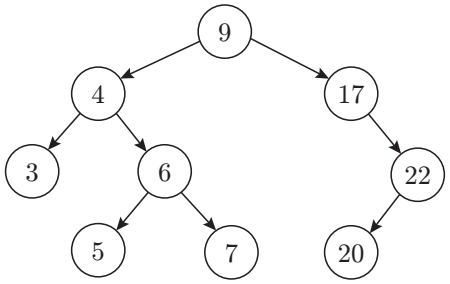
#### *3.6.5.1 Tree Traversal in Binary Search Tree*

There are three types of traversal techniques for tree:

1. **Preorder:**
   - Traverse the root
   - Traverse all the left external nodes starting with the left-most subtree
   - Traverse the right subtree starting at the left external node
2. **Postorder**
   - Traverse all the left external nodes starting with the left-most subtree
   - Traverse the right subtree starting at the left external node
   - Traverse the root

3. **Inorder**
   - Traverse the left-most subtree starting at the left external node
   - Traverse the root
   - Traverse the right subtree starting at the left external node

**Problem 3.6:** Consider the given tree and find the inorder, preorder and postorder of the tree.




**Solution:**

**Inorder Traversal:**

The inorder traversal of the above tree will output:

3, 4, 5, 6, 7, 9, 17, 20, 22

**Preorder Traversal:**

The preorder traversal of the above tree will output:

9, 4, 3, 6, 5, 7, 17, 22, 20

**Postorder Traversal:**

The postorder traversal of the above tree will output:

3, 5, 7, 6, 4, 20, 22, 17, 9

## 3.6.6 Graphs

A graph $G = (V, E)$ is composed of $V$ (set of vertices) and $E$ (set of edges), where edges connect the vertices. For example, see Fig. 3.18.

$V = \{a, b, c, d, e\}$ and $E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$

### *3.6.6.1 Basic Terminology*

Following are the basic terms used for graphs (see figure 3.19):

1. **Undirected graphs:** It consists of a set $V$ of vertices and a set $E$ of edges, each of them connecting to two different vertices.
2. **Directed graph:** In this, edges provide a connection from one vertex to another, but not necessarily in the opposite direction.
3. **Adjacent vertices:** Connected by an edge.
4. **Degree of a vertex:** It is defined as the number of adjacent vertices.
5. **Path:** It is defined as a sequence of vertices such that consecutive vertices are adjacent.
6. **Simple path:** Path with no repeated vertex.
7. **Cycle:** A simple path, where the first and last vertex are same.
8. **Connected graph:** When any two vertices are connected by some path.
9. **Subgraph:** When a subset of vertices and edges form a graph.
10. **Tree:** A connected graph without cycles.
11. **Balanced binary tree:** A binary tree in which no leaf is at much greater depth than any other leaf. Example of balanced binary search trees are red-black trees, AVL trees, etc. Examples of non-binary balanced search trees are B trees, B+ trees, etc.

### *3.6.6.2 Traversing a Graph*

There are two types of traversal techniques for graph:

1. **Breadth first traversal:** It is a level-by-level traversal of an ordered tree. It starts with vertex $I$, then it traverses to neighbours of vertex $i$, then neighbours of neighbours of vertex $i$ and so on. It uses a queue.

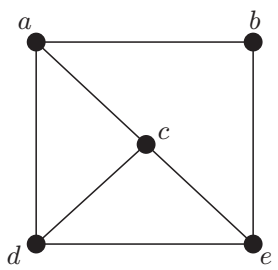



**Figure 3.18** | A graph $G = (V, E)$.

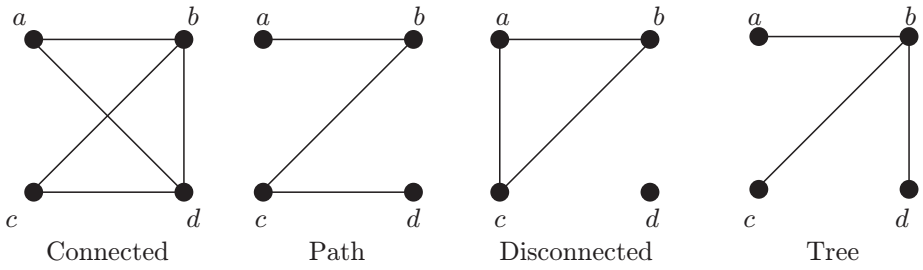
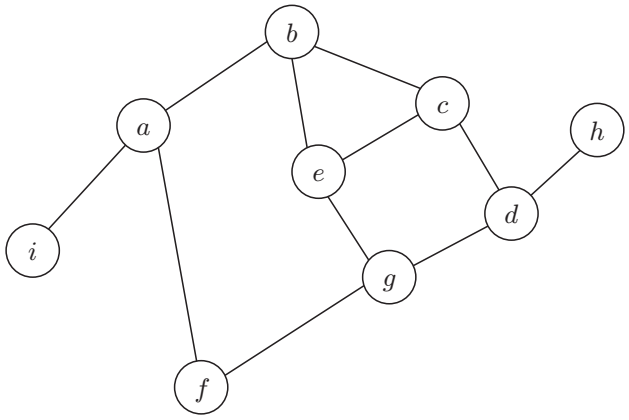
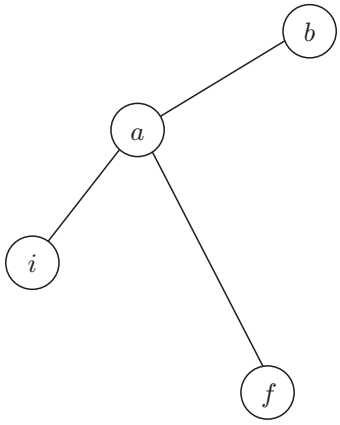



**Figure 3.19** | Examples showing connected, path, disconnected and tree graphs.

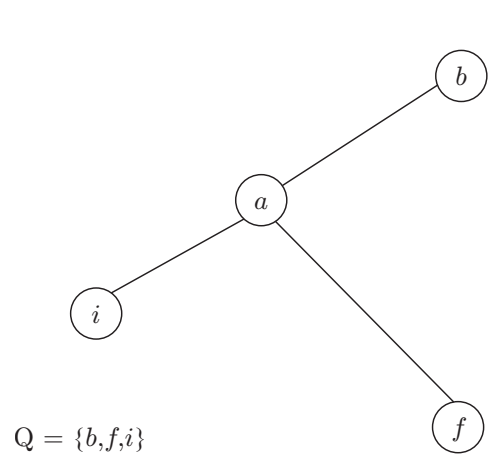
**Example 3.9**



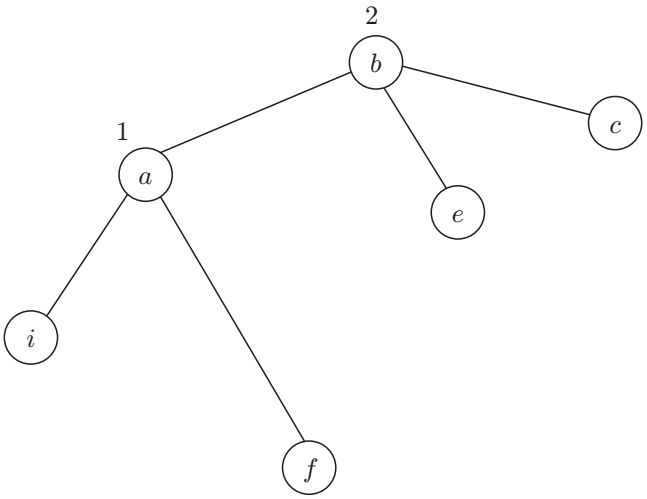

Q = {*a*}

(i)

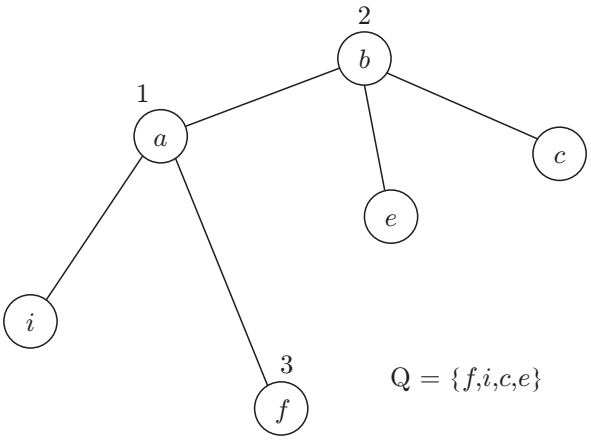



Q = {*b*,*f*,*i*}

(ii)

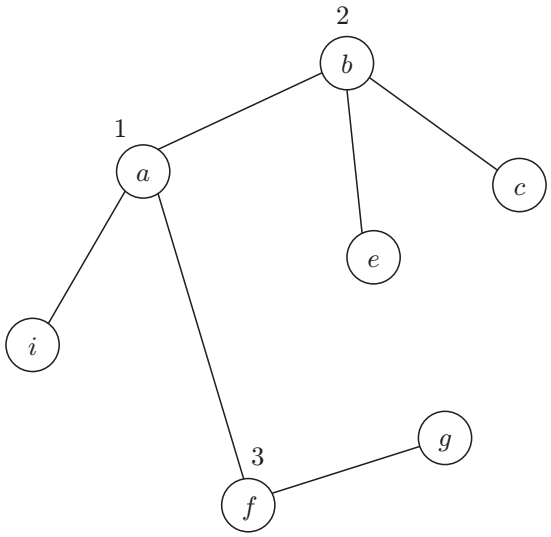



(iii)

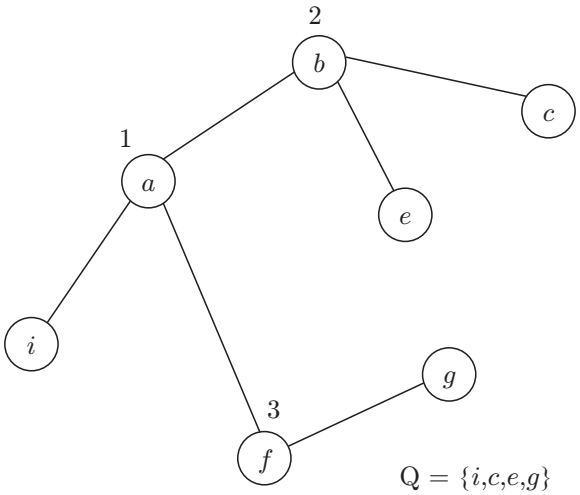



Q = {*f*,*i*,*c*,*e*}

(iv)

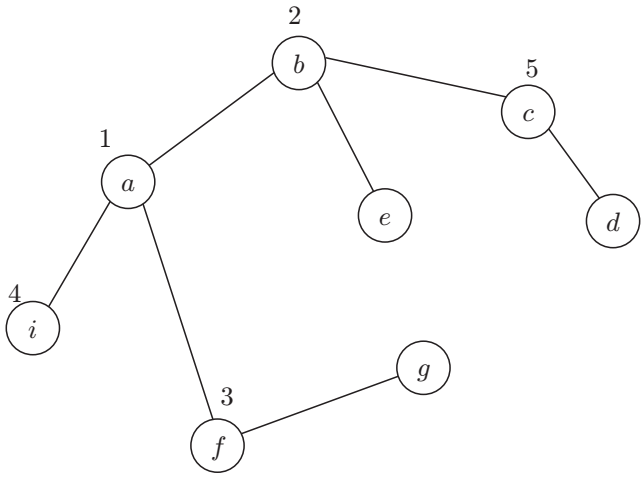



(v)

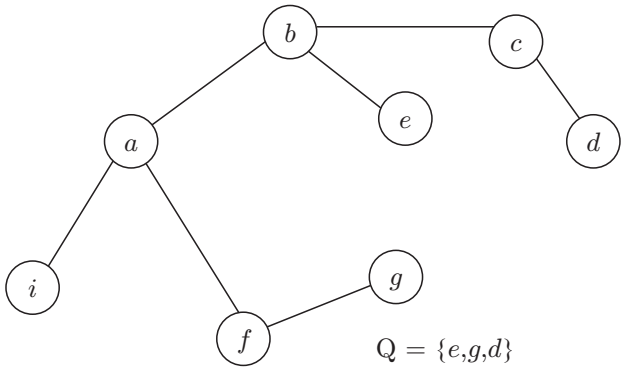



Q = {*i*,*c*,*e*,*g*}

(vi)

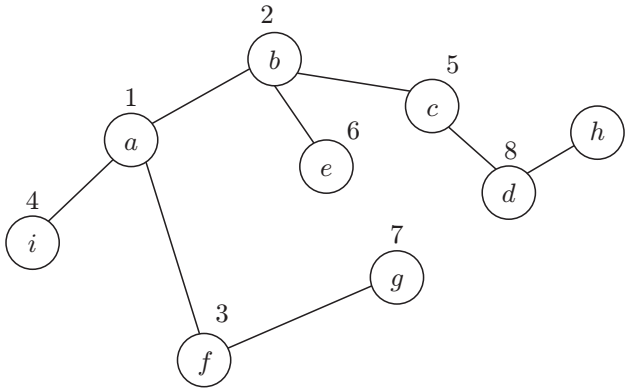



(vii)

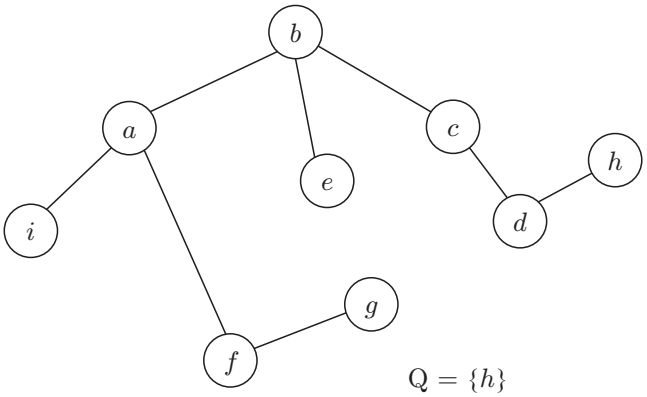



Q = {*e*,*g*,*d*}

(viii)

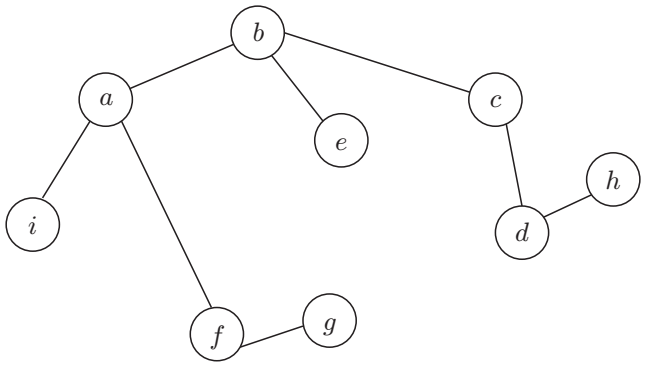



(ix)




Q = {*h*}

(x)

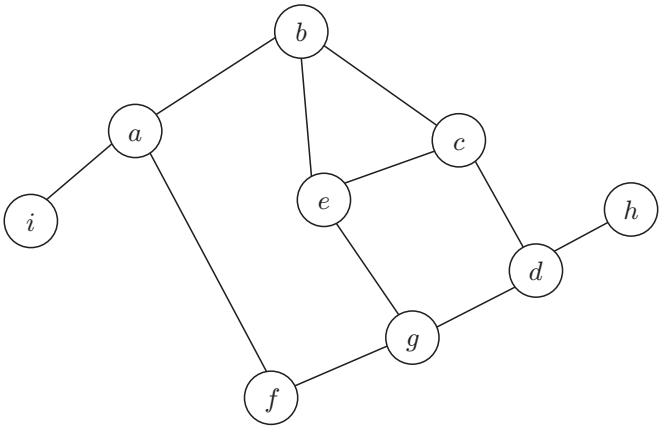


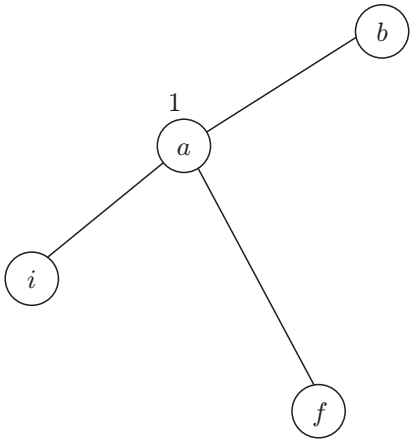

{*a*,*b*,*f*,*i*,*c*,*e*,*g*,*d*,*h*}

(xi)

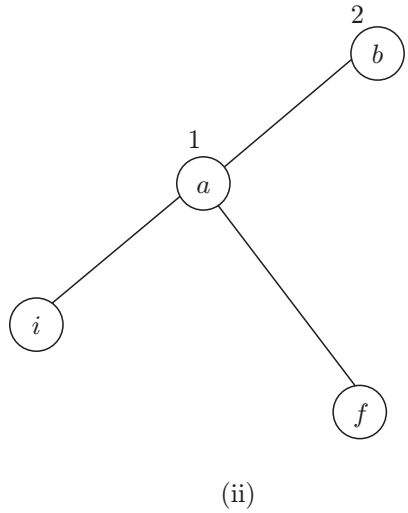
**2. Depth first traversal:** It is a preorder traversal of an ordered tree. It first traverses one subtree before returning to the current node and then traverses another subtree. It uses stack.

**Example 3.10**

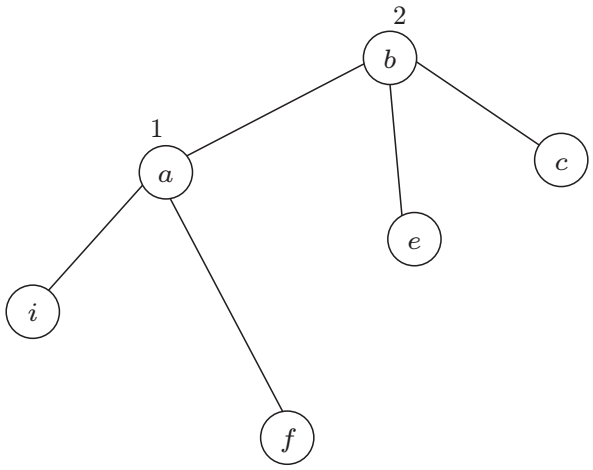


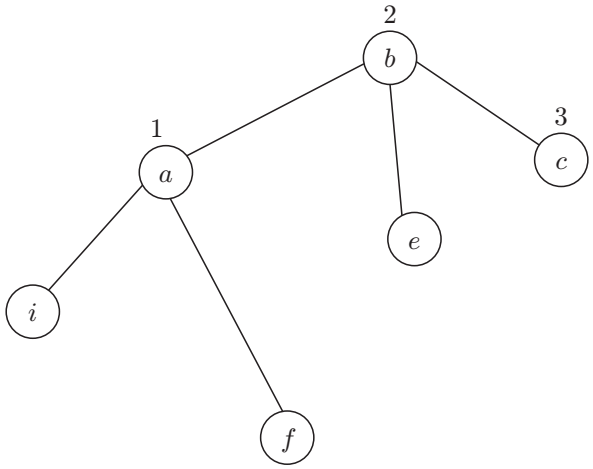

(i)



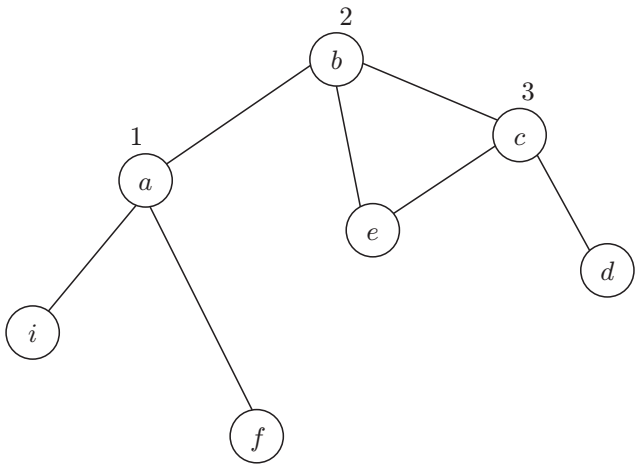

(ii)



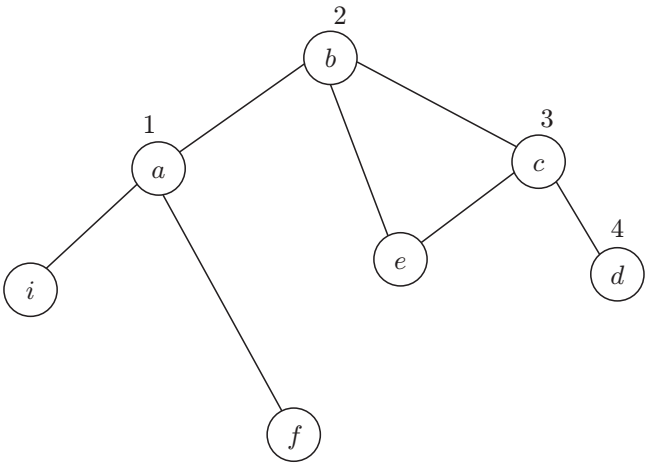

(iii)



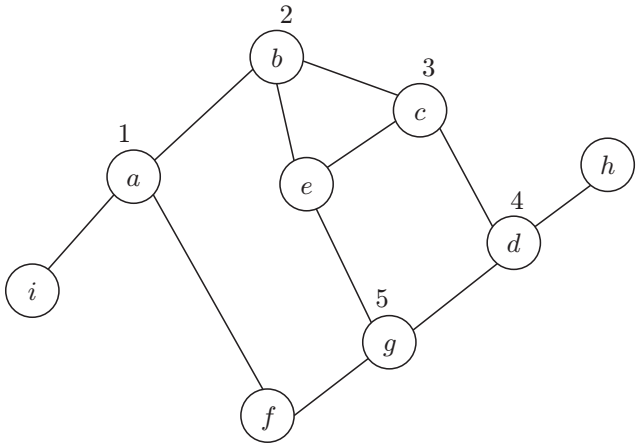

(iv)



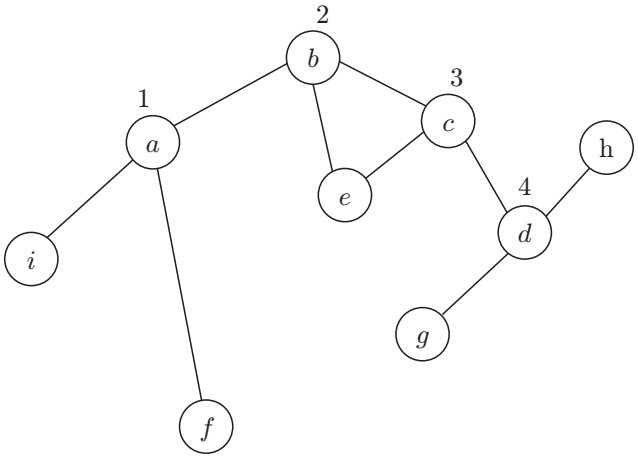

(v)



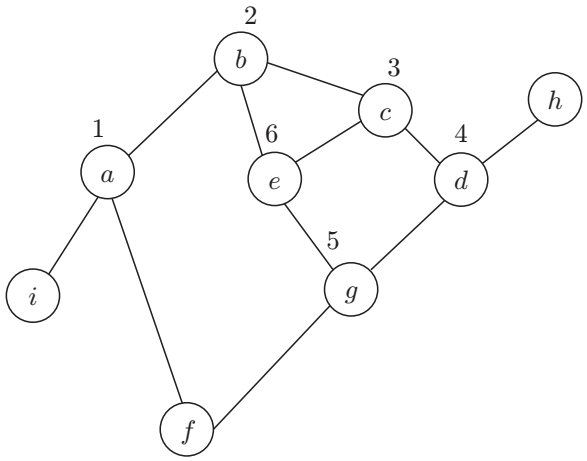

(vi)



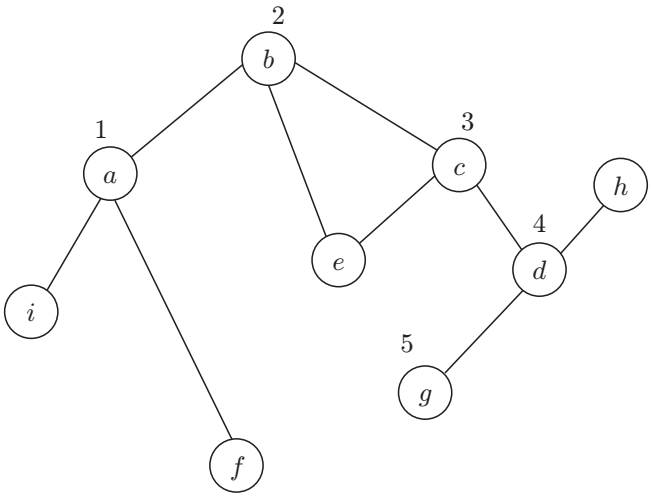

(ix)



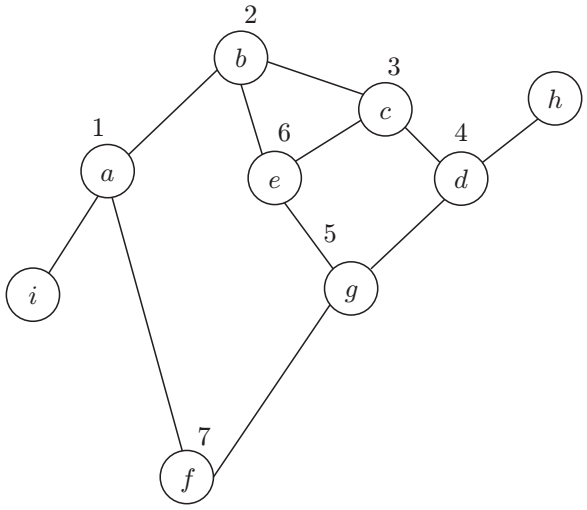

(vii)



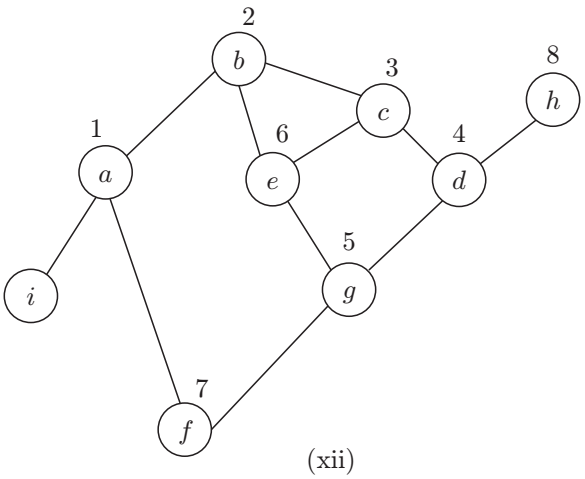

(x)



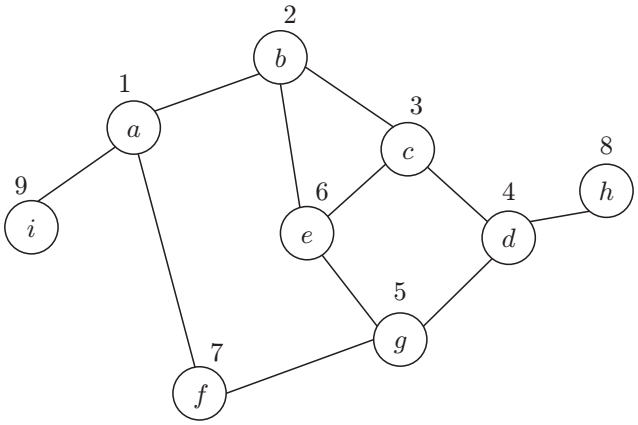

(viii)



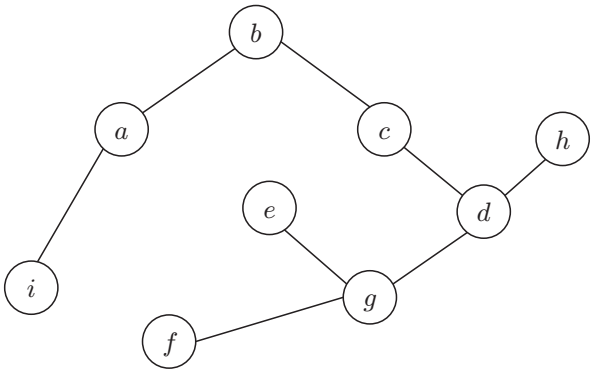

(xi)




(xii)




(xiii)




{a,b,c,d,g,e,f,h,i}

(ix)

### *3.6.6.3 Shortest Path*

For a given directed graph, where each edge has a non-negative weight or cost, the shortest path problem is to find a path of least total weight from the source to every other vertex in the graph (Fig. 3.20).

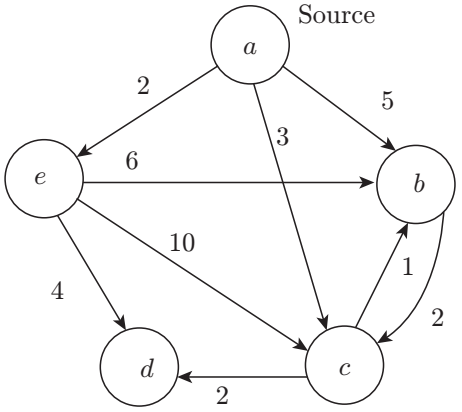
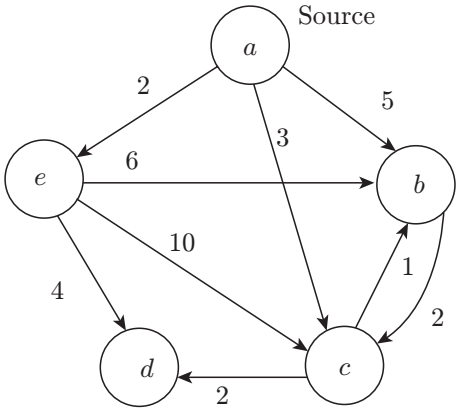


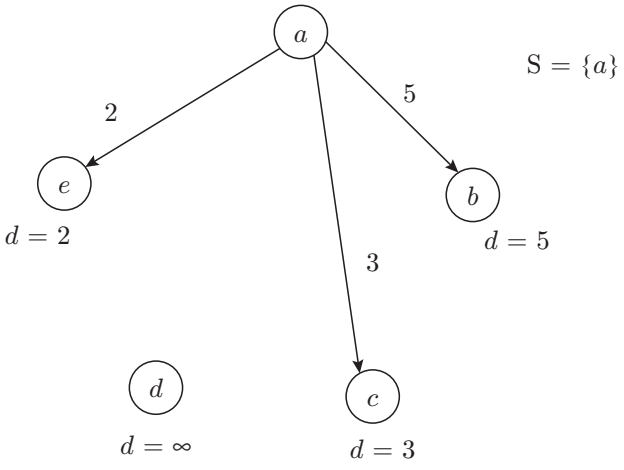

**Figure 3.20** | Graph for finding the shortest path.

To find the shortest path, an approach known as Greedy method is used. The method is as follows:

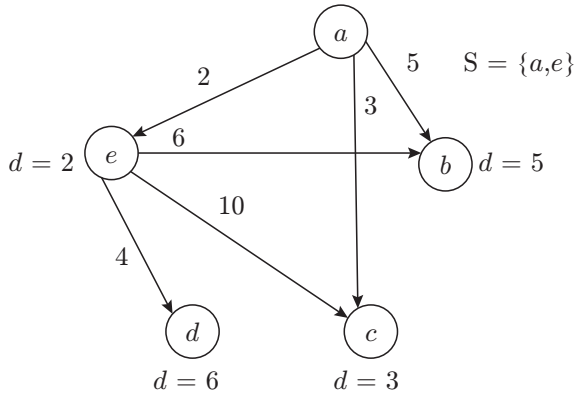
Choose the vertex with smallest weight and mark the path from source to the chosen vertex. Next, mark the same for other vertices.

**Example 3.11**

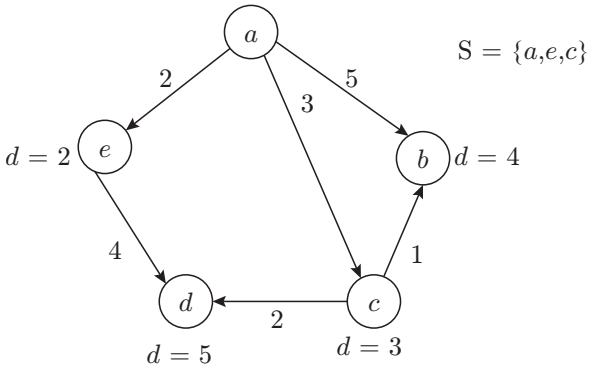


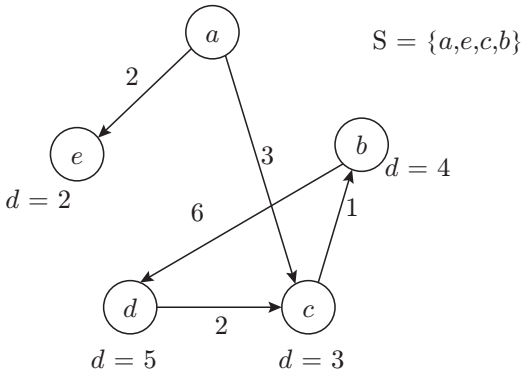

(i)



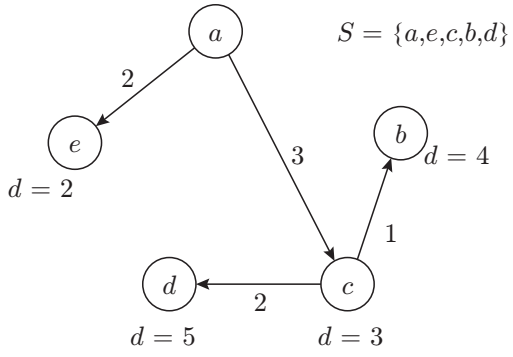

(ii)




(iii)




(iv)




(v)

### *3.6.6.4 Spanning Tree*

A spanning tree of a graph $G$ is a tree which contains all the vertices of the graph $G$. See Fig. 3.21.

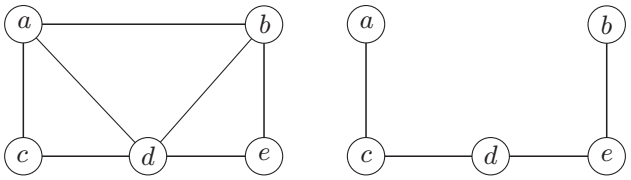



**Figure 3.21** | Spanning tree of a graph.

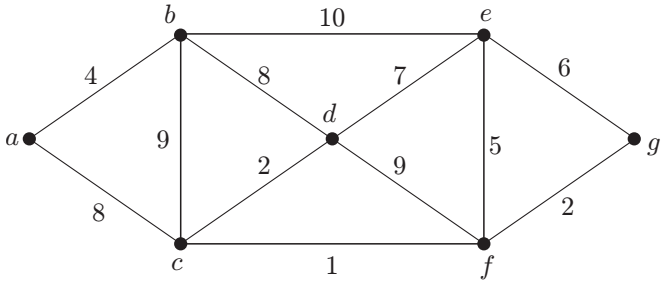
### Minimum Spanning Tree

Minimum spanning tree in an undirected connected weighted graph is a spanning tree with minimum weight.

Prim's algorithm:

1. Begin with all the vertices.
2. Choose and draw any vertex.
3. Find the edge of least weight joining a drawn vertex to a vertex not currently drawn. Draw the weighted edge and the corresponding new vertex.
4. Repeat step 3 until all the vertices are connected, then STOP.

**Example 3.12**



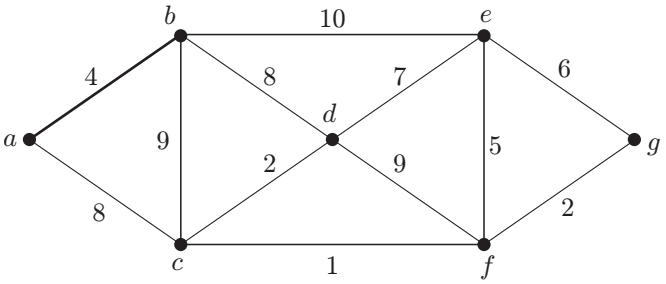

(i)

**Step 1**

$U = \{a\},\ V - U = \{b,\ c,\ d,\ e,\ f,\ g\}$

| Closest | | |
|---|---|---|
| **U** | **V − U** | **Low Cost** |
| $b$ | $A$ | 4 |
| $c$ | $A$ | 8 |
| $d$ | $A$ | $\infty$ |
| $e$ | $A$ | $\infty$ |
| $f$ | $A$ | $\infty$ |
| $g$ | $A$ | $\infty$ |

Select vertex $b$ to include in $U$.



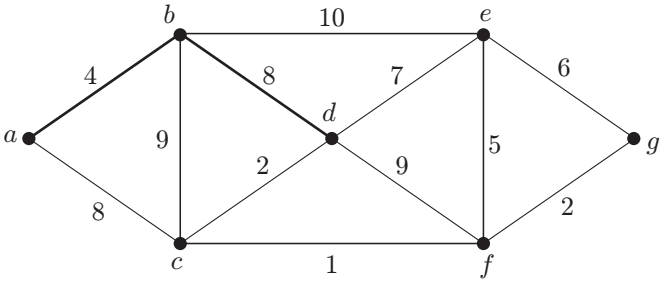

(ii)

**Step 2**

$U = \{a, b\},\ V - U = \{c, d, e, f, g\}$

| Closest | | |
|---|---|---|
| **U** | **V – U** | **Low Cost** |
| $C$ | $A$ | 8 |
| $D$ | $B$ | 8 |
| $E$ | $B$ | 10 |
| $F$ | $A$ | $\infty$ |
| $G$ | $A$ | $\infty$ |

Select vertex $d$ to include in $U$.



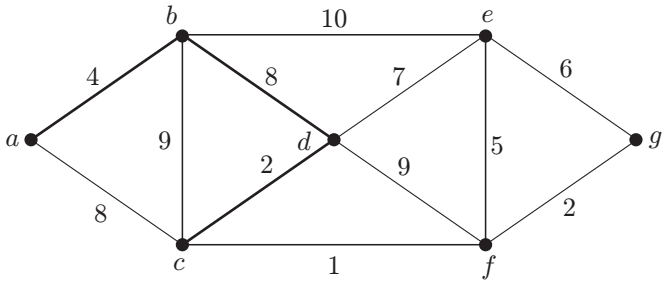

(iii)

**Step 3**

$U = \{a, b, d\},\ V - U = \{c, e, f, g\}$

| Closest | | |
|---|---|---|
| **U** | **V – U** | **Low Cost** |
| $c$ | $D$ | 2 |
| $e$ | $D$ | 7 |
| $f$ | $D$ | 9 |
| $g$ | $A$ | $\infty$ |

Select vertex $c$ to include in $U$.



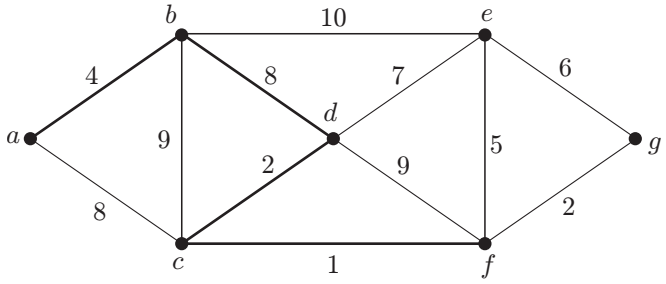

(iv)

**Step 4**

$U = \{a, b, c, d\},\ V - U = \{e, f, g\}$

| Closest | | |
|---|---|---|
| **U** | **V – U** | **Low Cost** |
| $e$ | $D$ | 7 |
| $f$ | $C$ | 1 |
| $g$ | $A$ | $\infty$ |

Select vertex $f$ to include in $U$.



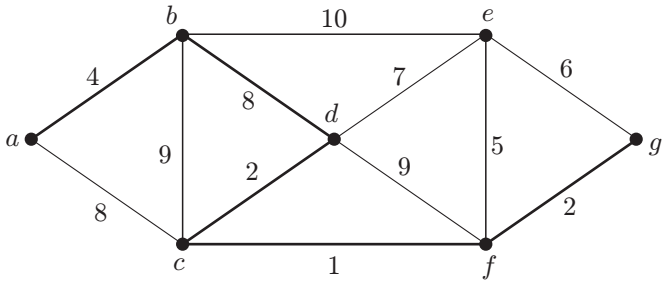

(v)

**Step 5**

$U = \{a, b, c, d, f\},\ V - U = \{e, g\}$

| Closest | | |
|---|---|---|
| **U** | **V – U** | **Low Cost** |
| $e$ | $F$ | 5 |
| $g$ | $F$ | 2 |

Select vertex $g$ to include in $U$.



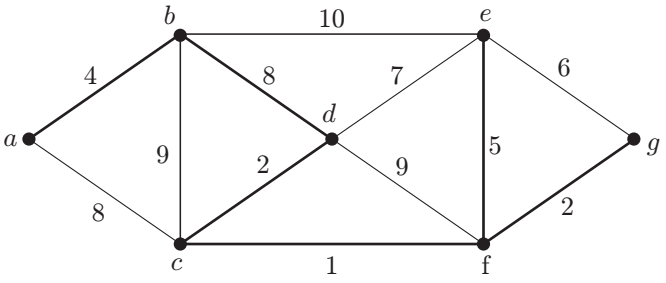

(vi)

**Step 6**

$U = \{a, b, c, d, f, g\}$, $V - U = \{e\}$

| Closest | | |
|---|---|---|
| **U** | **V – U** | **Low Cost** |
| *e* | *F* | 5 |

Select vertex $e$ to include in $U$.




(vii)

**Step 7**

$U = \{a, b, c, d, e, f, g\}$, $V - U = \{\}$

MST Complete

### *3.6.6.5 AVL Tree*

AVL tree is a binary tree which satisfies the height balance property (Fig. 3.22). It has a time complexity of $O(\log(n))$.

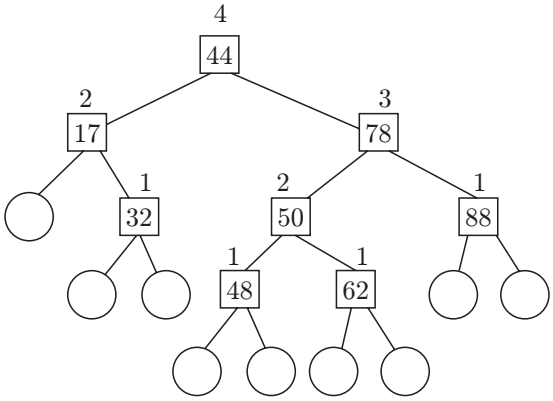



**Figure 3.22** | AVL tree.

AVL has the following properties:

1. **Balance Factor of a Node:** It is the difference in heights of its two subtrees ($h_R - h_L$).
2. **Balanced Node:** Node with $|BF| \leq 1$; if $|BF| > 1$, the node is unbalanced.
3. **Balance Factor of Binary Trees:** It corresponds to the balance factor of its root node.
   - Tree is 'left-heavy' if BF $\leq -1$
   - Tree is 'equal-height' if BF $= 0$
   - Tree is 'right-heavy' if BF $\geq +1$
4. **Balance Factor of AVL Trees:** BF of each node in an AVL tree can only $\in \{-1, 0, 1\}$.

#### Rotation

It is the restructuring of the tree which maintains the binary search tree property.

1. **Types of Rotation:**
   - *Single-Left rotation*: Refer Fig. 3.23.

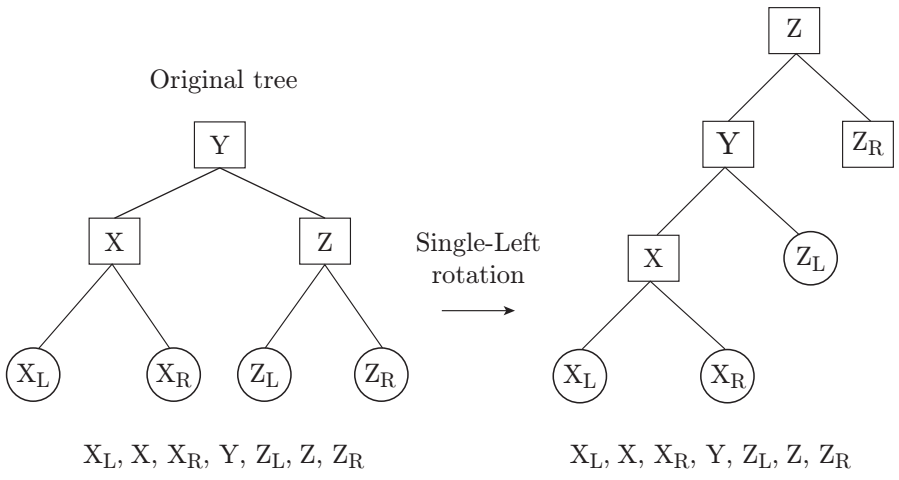



**Figure 3.23** | Single-Left rotation.

- *Single-Right rotation*: Refer Fig. 3.24.

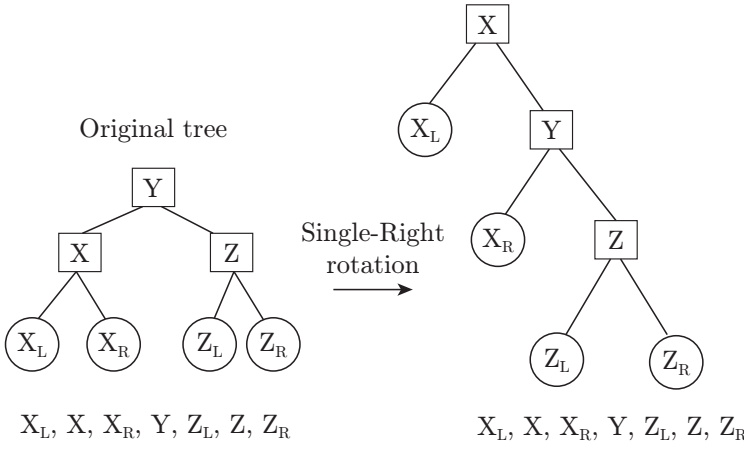



**Figure 3.24** | Single-Right rotation.

- *Double left-right rotation*: Refer Fig. 3.25.

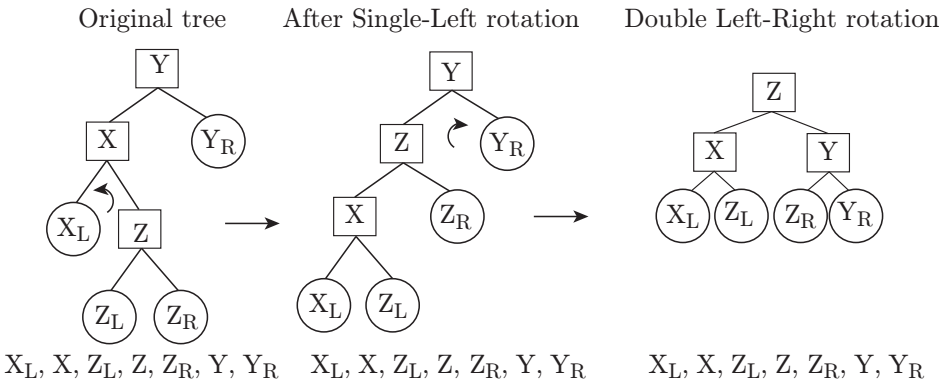



**Figure 3.25** | Double left-right rotation.

- *Double right-left rotation*: Refer Fig. 3.26.

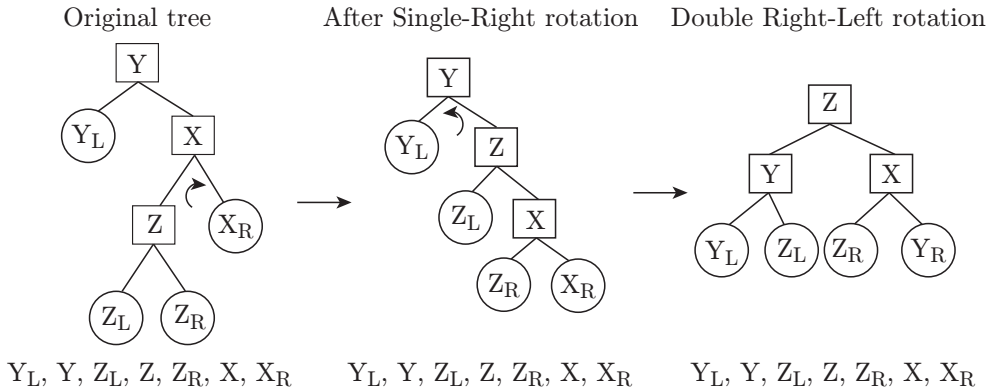



**Figure 3.26** | Double right-left rotation.

#### *3.6.6.6 Binary Heap*

A complete binary tree is said to be binary heap if all the elements below in hierarchy are either greater than or equal to the root element. This binary heap (see Fig. 3.27) is known as min-heap. In addition, if all the elements are lesser or equal to the root element, this is called max-heap.

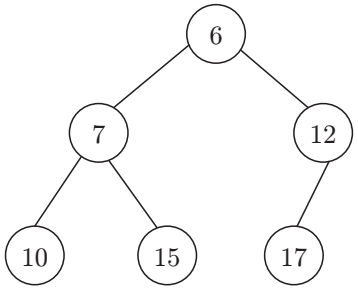



**Figure 3.27** | Binary heap.

1. The height of a binary heap is $O(\log\ n)$. The runtime of deleteMin, insert and remove is $O(\log\ n)$.
2. The root node is the highest priority element. To remove the highest/lowest priority object (root), heap data structure is preferred. Priority queue is the best example to be implemented.
3. The running time of heap sort is $O(n\log\ n)$.

## IMPORTANT FORMULAS

1. Insertion/Deletion in array: $O(n)$
2. Split/Merge in array: $O(n)$
3. Time complexity of Push(): $O(1)$
4. Time complexity of Pop(): $O(1)$
5. Time complexity of Enqueue(): $O(1)$
6. Time complexity of Dequeue(): $O(1)$
7. Number of elements in queue =

$$\begin{cases} \text{rear} - \text{front} + 1, & \text{if rear} = \text{front} \\ \text{rear} - \text{front} + n, & \text{otherwise} \end{cases}$$

8. Number of nodes in full binary tree: $2^{h+1} - 1$ [$h$: levels]
9. Number of leaf nodes in full binary tree: $2^h$
10. Number of waste pointers in complete binary tree of $n$ nodes: $n + 1$
11. Complexity of inorder, preorder and postorder tree traversal: $O(n)$
12. Minimum number of moves for tower of Hanoi: $2^n - 1$
13. Number of unique binary trees: $\dfrac{^{2n}C_n}{n+1}$
14. **One-dimensional arrays**

    In one dimension, an array '$A$' is declared as: $A$[lb … … ub]

    where lb is the lower bound of array and ub is the upper bound of array.

    Suppose we want to calculate the $i$th element address, then

    $$\text{Address (arr}[i]) = \text{BA} + (i - \text{lb}) * c$$

    where BA is the base address of array and lb is the lower bound of array and $c$ is the size of each element

15. **Two-dimensional arrays**

    In two dimensions, an array '$A$' is declared as:

    $$A[\text{lb}_1 \ldots \ldots \text{ub}_1][\text{lb}_2 \ldots \ldots \text{ub}_2]$$

    where $\text{lb}_1$ is the lower bound for row, $\text{lb}_2$ is the lower bound for column, $\text{ub}_1$ is the upper bound for row and $\text{ub}_2$ is the upper bound for column.

    **Row major order**

    $$\text{Address (arr}[i][j]) = \text{BA} + [i - \text{lb}_1) * \text{Nc} + (j - \text{lb}_2)] * c$$

    **Column major order**

    $$\text{Address (arr}[i][j]) = \text{BA} + [j - \text{lb}_2) * \text{Nr} + (i - \text{lb}_1)] * c$$

    where BA is the base address, Nr is the number of rows = $(\text{lb}_2 - \text{lb}_1 + 1)$, and Nc is the number of columns = $(\text{ub}_2 - \text{ub}_1 + 1)$.

## SOLVED EXAMPLES

**1.** Variables of function call are allocated in

(a) registers and stack. (b) cache and heap.
(c) stack and heap. (d) registers and heap.

*Solution:* The variables of function call can be stored in stack and heap.

Ans. (c)

**2.** Predict the output of:

```
void main()
{
     float x=1.1;
     double y= 1.1;
if(x==y)
printf("I see You");
else
printf("I hate You");
     }
```

(a) I see You
(b) I hate You
(c) Cannot compare double and float
(d) Run time error

*Solution:* The stored value may or may not be exact, the precision of the value depends upon the number of bytes. Double stores value 1.1 with more precision than float because double takes 8 bytes, whereas float takes 4 bytes.

Ans. (b)

**3.** Global variable conflicts due to multiple file occurrence is resolved during

(a) load time.
(b) execution time.
(c) link time.
(d) parsing phase of compiler.

*Solution:* A global variable provides access from multiple files. Due to merging of one file code to another code, conflict occurs in the variable names and this cause an error at link time.

Ans. (c)

**4.**
```
#define f(a,b) a+b
#define g(a,b)a*b
main()
{
     int m;
     m=2*f(3,g(4,5));
     printf("\n m is %d", m);
}
```

What is the value of *m*?

(a) 70 (b) 50
(c) 46 (d) 69

*Solution:* `m=2*f(3,g(4,5));`
The function g(4,5) will return 20 by performing multiplication operations. Then function f (3,20) will return 23 by performing addition operation. In the last output of function f (3,20) is multiplied with 2. So, the final answer will be 46.

Ans. (c)

**5.** A program to find the factorial of a given number is written (a) using recursion and (b) without using recursion. Which of these will result in stack overflow for a given number?

(a) Only first program with recursion
(b) Both may result for the same number
(c) Only second program
(d) None of these

*Solution:* When the factorial program is written using recursion only then will it use stack, so stack overflow will occur in recursive program only.

Ans. (a)

**6.** If two strings are identical then strcmp() returns:

(a) True (b) 1
(c) −1 (d) 0

*Solution:* strcmp() is a pre-defined function of C library which returns value by calculating value=(second string length – first string length). Hence, if both the strings are identical then it will return 0.

Ans. (d)

**7.** In tree construction, which one will be suitable and efficient data structure?

(a) Queue (b) Linked list
(c) Heap (d) String

*Solution:* Linked list is the best suitable and efficient data structure for constructing a tree because an item can be inserted and deleted from linked list with less cost.

Ans. (b)

**8.** Which of the following is not true about spanning tree?

(a) It is a tree derived from a graph.
(b) All the nodes of a network appear on the tree only once.

(c) Spanning tree cannot have at most two edges repeated.
(d) Spanning tree cannot be minimum and maximum.

*Solution:* A spanning tree is a subgraph of a given graph G, which covers all the vertices of G and should be a tree. Spanning tree could be minimum or maximum.

Ans. (d)

**9.** Which of the following has compilation error in C?

(a) int $n = 17$;
(b) char ch = 99;
(c) float $f =$ (float) 99.32;
(d) #include<stdio.h>

*Solution:* The syntax for inclusion of header file is `#include <stdio.h>`. There should be space between #include and <stdio.h>.

Ans. (d)

**10.** _____ is often used to prove the correctness of a recursive function.

(a) Associativity
(b) Commutativity
(c) Mathematical induction
(d) Factorial

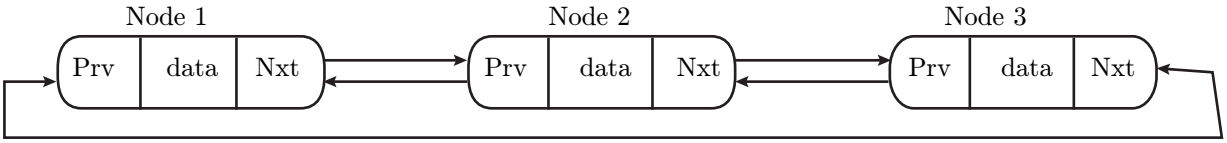
*Solution:* Mathematical induction is the method through which correctness of recursive function can be proved.

Ans. (c)

**11.** In a doubly linked list, the number of pointers affected for an insertion operation will be:

(a) 4
(b) 0
(c) 1
(d) Depends upon the nodes of the doubly linked list.

*Solution:* Let a node X be inserted after node 1, then both pointer of node X(Prv and Nxt) will be affected and node 1's Nxt pointer will point to node X and node 2's Prv pointer will also point to node X. So, total four pointers will be affected.

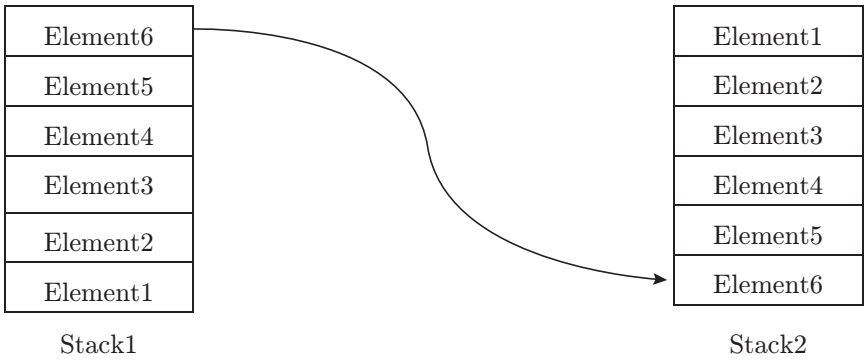



Ans. (a)

**12.** The queue data structure is to realised by using stack. The number of stacks needed would be:

(a) It cannot be implemented.
(b) 2 stacks.
(c) 4 stacks.
(d) 1 stacks.

*Solution:* Let there are 6 elements (1, 2, 3, 4, 5 and 6). First we will push them in stack 1 and then pop from stack 1 and push in stack2. The same behaviour will be observed when we will pop from stack2




So, only 2 stacks are required. So, option (b) is correct.

Ans. (b)

**13.** Postfix expression for the infix expression

$$(A*B - (C*D))/(F/D + E)$$

(a) $AB*CD* - FD/E+/$ (b) $AB*CD - *FDE//+$
(c) $AB*CD - FD*E//+$ (d) $AB*CD - *FDE/+/$

*Solution:*

| Infix Expression | Postfix Expression | Stack |
|---|---|---|
| $(A*B - (C*D))/(F/D + E)$ | | |
| $*B - (C*D))/(F/D + E)$ | A | ( |
| $B - (C*D))/(F/D + E)$ | A | (* |
| $- (C*D))/(F/D + E)$ | AB | (* |
| $(C*D))/(F/D + E)$ | AB* | (– |
| $*D))/(F/D + E)$ | AB*C | (–( |
| $D))/(F/D + E)$ | AB*C | (–(* |
| $/(F/D + E)$ | AB*CD | (–(*)) |
| $/(F/D + E)$ | AB*CD*– | |
| $(F/D + E)$ | AB*CD*– | / |
| $/D + E)$ | AB*CD*–F | / |
| $D + E)$ | AB*CD*–F | /(/ |
| $+ E)$ | AB*CD*–FD | /(/ |
| $E)$ | AB*CD*–FD/ | /(+ |
| | AB*CD*–FD/E | /(+) |
| | **AB*CD*–FD/E+/** | |

Ans. (a)

**14.** There are six nodes. The different types of trees that can be realised are _____.

(a) 10 (b) 64
(c) 58 (d) 65

*Solution:* 58 trees can be realized.

Ans. (c)

**15.** In which of the following case(s) is it possible to obtain different results for call-by-reference and call-by-name parameter passing

(a) Passing an expression as a parameter
(b) Passing an array as a parameter
(c) Passing a pointer as a parameter
(d) Passing an array elements as a parameter

*Solution:* Passing an element of an array by call by name behaves similar to call by value. Therefore, if array elements are passed as parameter then they can produce different results for call-by-reference and call-by-name parameter passing.

Ans. (d)

**16.** A variant record in Pascal is defined by

```
typevarirec = record
     number: integer;
     case (var1, var2) of
          var1; (x, y : integer)
           var2: (p,q: real)
     end
end
```

Suppose an array of 100 records was declared on a machine which uses 6 bytes for an integer and 10 bytes for a real. How much space would the compiler have to reserve for the array

(a) 3800 (b) 3200
(c) 2800 (d) 4000

*Solution:* There are 100 records
So 600 B for 'record no': [100 × 6]
1200 B for var($x$, $y$): [so, (6 + 6)*100]
2000 B for var($p$, $q$) : [so, (10 + 10)*100]

Ans. (a)

**17.** What is the result of the following program?

```
Program side-effect (input, output)
var x, result: integer;
Function f (var x: integer): integer;
begin
     x: x+1;
     f:=x;
end
begin
     x:=5;
     result:=f(x)*f(x);
     writeln (result)
end
```

(a) 5 (b) 25
(c) 36 (d) 42

*Solution:* Function '$f$' uses '$x$' as local variable. So value of '$x$' does not change globally and both the time $x = 5$ is passed to the function '$f$'.

Ans. (c)

**18.** Consider the following C declaration

```
struct {
short s [5]
union{
float y;
long z;
}u;
}t;
```

Assume that objects of the type short, float and long occupy 2 bytes, 4 bytes and 8 bytes, respectively. The memory requirement for variable $t$, ignoring alignment considerations, is

(a) 22 bytes (b) 18 bytes
(c) 14 bytes (d) 10 bytes

*Solution:* The structure is created with total size of 'short and union' short have size of 2 byte, as it is an array of five elements so it will take 10 bytes. Union will consider the maximum of {float $y$ (4 bytes) and long $z$(8 bytes)} 12 bytes. So, total size will be 22 bytes (10 + 12).

Ans. (a)

**19.** In a compact single-dimensional array representation for lower triangular matrices (i.e. all the elements above the diagonal are zero) of size $n \times n$, non-zero elements (i.e. elements of the lower triangle) of each row are stored one after another, starting from the first row, the index of the ($i$,$j$)th element of the lower triangular matrix in this new representation is

(a) $i + j$ (b) $i + j - 1$

(c) $(j-1) + \frac{i-1}{2}$ (d) $i + \frac{j(j-1)}{2}$

*Solution:* To find location in lower triangular matrix formula is given as

$$LOC(i,j) = (j-1) + \frac{i(i-1)}{2}$$

Ans. (c)

**20.** The following sequence of operation is performed on a stack:

PUSH(10), PUSH(20), POP, PUSH(10), PUSH(20), POP, POP, POP, PUSH(20), POP

The sequence of values popped out is

(a) 20, 10, 20, 10, 20 (b) 20, 20, 10, 10, 20
(c) 10, 20, 20, 10, 20 (d) 20, 20, 10, 20, 10

*Solution:*

| Operation | Stack | Popped Elements List |
|---|---|---|
| Push | 10 | |
| Push | 10, 20 | |
| Pop | 10 | 20 |
| Push | 10, 10 | 20 |
| Push | 10, 10, 20 | 20 |
| Pop | 10, 10 | 20, 20 |
| Pop | 10 | 20, 20, 10 |
| Pop | | 20, 20, 10, 10 |
| Push | 20 | 20, 20, 10, 10 |
| Pop | | 20, 20, 10, 10, 20 |

So, option (b) is correct.

Ans. (b)