THIRUVALLUVAR UNIVERSITY

E-NOTES

BCS 21 / BCA 21 – PART II
DATA STRUCTURES

(2nd Semester - B.Sc. CS / BCA)

STEERED BY

Dr. S. THAMARAI SELVI, M.E., PH.D.,
Vice Chancellor,
Tiruvalluvar University, Serkkadu, Vellore

PREPARED BY

1. **Dr. M.DURAISAMY, M.C.A., M.Phil., Ph.D., SET.,**
   Assistant Professor & Head, Department of Computer Applications,
   Thiruvalluvar University College of Arts and Science, Tirupattur.

2. **Dr. G.T. SHRIVAKSHAN, M.C.A., M.Phil., Ph.D.,**
   Assistant Professor, Department of Computer Science,
   PG Extension Centre, Thiruvalluvar University,
   Villupuram.

3. **Mr. D.CHANDRU**
   Assistant Professor, Department Of Computer Science,
   Sri Vinayaga College of Arts And Science, Ulundurpet,
   Villupuram.

ACKNOWLEDGEMENT

Creation of e-content is really an interesting and challenging endeavor. Data Structure is the quintessential base for all programming languages. This e-content aims to impart the necessary aspects of data structure in a simple manner. During the course of the content preparation, our Honorable Vice Chancellor ***Dr. S. Thamarai Selvi*** *guided us and inspired us in every stage. Her input and suggestions facilitated us to cross the hurdles which we faced during the process of this e-content creation. She encouraged us to create a course material which will suit the students of all learner levels right from the top.*

Dr. M.DURAISAMY

Dr. G.T. SHRIVAKSHAN

Mr. D.CHANDRU

SYLLABUS

Objective:

To understand the quintessential knowledge of data structure and enable the student to implement data structure concepts in C++.

Fundamental Data Structures - Definition of a Data structure - Primitive and Composite Data Types – Linear and Non Linear Data Structures,

Stacks – Stack using array – Operations on Stack –Linked Stack - Applications of Stack (Infix to Postfix Conversion).

Queue – Queue Using Array- Operations on Queue - Linked Queue-and its operations.

Linked list -Singly Linked List - Operations, Application of List (Polynomial Addition) -. Doubly Linked List and its operations.

Trees -Trees: Binary Trees –Binary Search Tree- Operations - Recursive Tree Traversals.

Graph - Graph - Definition, Types of Graphs, Graph Traversal, Dijkstra's shortest path, DFS and BFS.

References:

1. Alfred V. Aho, John E. Hopcroft and Jeffry D. Ullman, "Data Structures & Algorithms".
2. Ellis Horowitz, SartajSahni and Dinesh Mehtha, "Fundamentals of Data Structures in C++", Second Edition, Oxford University Press.
3. G A V PAI , "Data Structures and Algorithms, Concepts", Techniques and Applications, Volume 1, 1st Edition, Tata McGraw-Hill, 2008.
4. Jean-Paul Tremblay and Paul G. Sorenson , "An Introduction to Data Structures with Applications", Tata McGraw-Hill, New Delhi, 1991.
5. Kleinberg, J. and Tardos, E., "Algorithm Design", Pearson Education, (2006).
6. Mark Allen Weiss , "Data structures and algorithm analysis in C++", Florida International University, Fourth Edition, Pearson.
7. Mehta, D. P. and Sahani, S., "Handbook of Data Structures and Algorithms", Chapman and Hall/CRC (2005).
8. Narasimha Karumanchi, "Data Structures and Algorithms Made Easy", 5th Edition, Career Monk Publications.
   Pearson Education, New Delhi, 2006.
9. Robert L. Kruse , "Data Structures and Program Design in C++", 1st Edition, PHI.
10. Seymour Lipschutz , "Data Structures with C (Schaum's Outline Series)", Tata McGraw-Hill Education India.
11. Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest Clifford Stein , "Introduction to Algorithms", Second Edition, McGraw-Hill, 2002.
12. Udit Agarwal , "Data Structures Using C", S K Kataria & Sons (2013).
13. Yashavant Kanetkar, "Data Structures Through C++", BPB Publications.

DATA STRUCTURES

4.1 INTRODUCTION

A computer is a programmable data processor that accepts input and instructions to process the input (program) and generates the required output.

A computer must be instructed through a programming language, what is to be done. This sequence of instructions is known as a **program.**

A **program** that satisfies user needs as per specifications is called ***software*.**

The physical machinery that actually executes these instructions is known as ***hardware*.**

The first phase of developing a program is to define the problem statement precisely.

After defining the problem, we have to select the best suitable algorithm to solve it.

An **algorithm** is a stepwise description of an action that leads the problem from its start state to its end state. An algorithm must be very clear, definite, and efficient for the problem.

The art of programming consists of designing or choosing algorithms and expressing them in a programming language.

Program design needs different ***kinds of information to be stored in the computer and the input data to be processed.*** The information can be stored in a generalized format using variables.

Computer science includes ***the study of data, its representation***, ***and its processing*** by computers. So, it is essential to study about the terms associated with ***data and its representation.***

The success of a software project depends upon the choices ***made in the representation of the data*** and ***the choice of algorithms***, and we need better methods to describe and process the data.

Basic Terminologies:

Data

Data is nothing but a piece of information. Data input, Data processing, and Data output are the functions of computers. So, all information taken as input, processed within a computer, or provided as output to the user is nothing but **data**.

It can be a number, a string, or a set of many numbers and strings.

Atomic and Composite Data

Atomic data is the data that we choose to consider as a single, non-decomposable entity.

For example, the integer 5678 may be considered as a single integer value. We can decompose it into digits, but the decomposed digits will not have the same characteristics of the original integer.

In some languages, atomic data is known as **scalar data** because of its numeric properties.

The ***Composite data*** can be broken down into subfields that have meaning. For example, a student's record consists of Roll Number, Name, Branch, Year, and so on. Composite data is also referred to as **structured data** and can be implemented using a structure in C or a class in C++.

Data Type

Data type refers to the kind of data a variable may store. Data type is a term that specifies the type of data that a variable may hold in the programming language.

Example: Char Name [60], Int Rollno.

Built-in Data Types

In general, languages have their built-in data types. They also allow the user to define their own data types, called *user-defined data types*, using the built-in data types; for example, in the C/C++ languages, int, float, and char are built-in data types.

Using these built-in data types, we can define our own data types by means of structures, unions, and classes.

User-defined Data Types

Suppose we want to maintain a record of 100 students with the following fields in each record: roll number, name of student, and percentage of marks of the students. Then we use the C++ class as follows:

```
class Student
{
private:
int roll;
char name[20];
float percentage;
public:
void GetRecord();
void PrintRecord();
void SearchRecord();
}
```

Class, structure, and union are the user-defined data types in C++.

Data Object

A *data object* represents a container for data values i.e., a place where data values may be stored and later retrieved.

A data object is characterized by a set of attributes, one of the most important of which is its data type.

The attributes determine the number and type of values that the data object may contain and also determine the logical organization of these values.

A data object is nothing but a set of elements, say *D*. The data object 'alphabets' can be defined as *D* = {*A*, *B*, …, *Z*, *a*, *b*, …, *z*} and the data object 'integers' as *D* = {…, -3, -2, -1, 0, 1, 2, 3, …}. The data object set may be finite or infinite.

A data object is a run-time instance of data structures. The programmer explicitly creates and manipulates these data objects through declarations and statements in the program.

Data Structure

Data structures refer to data and representation of data objects within a program, that is, the implementation of structured relationships. A data structure is a collection of atomic and composite data types into a set with defined relationships.

The term ***Data Structure*** refers to the organization of data elements and the interrelationships among them.

The field of data structures is very important and central to the study of computer science and programming.

Association among Data, Data Structures and Algorithms

There is a close relationship between the structuring of data and analysis of algorithms.

A data structure and an algorithm should be thought of as one single unit, neither one making sense without the other.

Let us consider the example of searching for a person's phone number in a directory. The procedure we follow to search a person and get his phone number critically depends on how the phone number and names are arranged in the directory. Let us consider two ways of organizing the data (phone numbers and names) in the directory.

1. The data is organized randomly. Then to search a person by name, one has to linearly start from the first name till the last name in the directory. There is no other option.

2. If the data is organized by sorting the names (alphabetically sorted in ascending order), then the search is much easier. Instead of linearly searching through all records, one may search in a particular area for particular alphabets, similar to using a dictionary.

As the data is in sorted order, both the binary search and a typical directory search methods work. So, our ideas for algorithms become possible when we realize that we can organize the data as we wish.

We can say that there is a ***strong relationship between*** the Fundamental concepts structuring of data (along with inter-relationship among ***data structures***) and the operations to process the data (***algorithms***). In fact, the way we process our data depends on the way we organize it.

Algorithm

A programmer should first solve the problem in a step-by-step manner and then try to find the appropriate instruction or series of instructions that solves the problem. This step-by-step solution is called an ***algorithm.*** An algorithm is independent of the computer system and the programming language.

Each algorithm includes steps for

1. Input, 2. Processing and 3. Output.

Characteristics of Algorithms

An algorithm is a *finite ordered* set of *unambiguous* and *effective* steps which, when followed, accomplish a particular task by accepting *zero or more input quantities* and generate *at least one output*.

The following are the characteristics of algorithms:

Input - An algorithm is supplied with zero or more external quantities as input.

Output - An algorithm must produce a result, that is, an output.

Unambiguous steps - Each step in an algorithm must be clear and unambiguous. This helps The Person or computer following the steps to take a definite action.

Finiteness - An algorithm must halt. Hence, it must have finite number of steps.

Effectiveness- Every instruction must be sufficiently basic, to be executed easily.

Analysis of algorithms

Analysis involves measuring the performance of an algorithm. Performance is measured in terms of the following parameters:

1. *Space complexity* — The amount of memory needed to perform the task.

2. *Time complexity* — The amount of time taken by an algorithm to perform the intended task

The *space complexity* of an algorithm is a measure of how much storage is required by the algorithm. It is possible to design an algorithm that uses more space and less time or less space and more time.

The *time complexity* of an algorithm is a measure of how much time is required to execute an algorithm for a given number of inputs and is measured by its rate of growth relative to standard functions.

Space Complexity

Space complexity is the amount of computer memory required during program execution as a function of the input size. Space complexity measurement, which is the space requirement of an algorithm, can be performed at two different times:

1. Compile time

2. Run time

Compile Time Space Complexity

Compile time space complexity is defined as the storage requirement of a program at compile time.

This storage requirement can be computed during compile time. The storage needed by the program at compile time can be determined by summing up the storage size of each variable using declaration statements.

This includes memory requirement before execution starts.

For example, the space complexity of a non-recursive function of calculating the factorial of number *n* depends on the number *n* itself.

Run-time Space Complexity

If the program is recursive or uses dynamic variables, then there is a need to determine space complexity at run-time. It is difficult to estimate memory requirement accurately, as it is also determined by the efficiency of compiler.

Memory requirement is the summation of the program space, data space, and stack space.

Program space This is the memory occupied by the program itself.

Data space This is the memory occupied by data members such as constants and variables.

Stack space This is the stack memory needed to save the function's run-time environment while another function is called. This cannot be accurately estimated since it depends on the run-time call stack, which can depend on the program's data set. This memory space is crucially important for recursive functions.

Time Complexity

Time complexity *T* (*P*) is the time taken by a program *P*, that is, the sum of its compile and execution times. This is system-dependent.

Another way to compute it is to count the number of algorithm steps. An algorithm step is a syntactically or semantically meaningful segment of a program. We can determine the number of steps needed by a program to solve a particular problem instance in one of the following two ways:

1. Introduce a new variable, count, into the program. This is a global variable with initial value 0. Statements to increment count amount are introduced in the program at appropriate locations. This is done so that each time the statement in the original program is executed, the count is incremented by the step count of that statement. We measure the run-time of an algorithm by counting the number of steps.

2. Manually compute the number of times each statement will be executed. The number of times the statement is executed is its *frequency count*. Get the sum of frequency counts of all statements. This sum is the number of steps needed to solve a given problem.

Best Case, Worst Case, and Average Cases

The *best case* complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.

The *worst case* complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.

The *average case* complexity of an algorithm is the function defined by an average number of steps taken on any instance of size *n*.

Each of these complexities defines a numerical function — time versus size.

Computing Time Complexity of an Algorithm

The total time taken by the algorithm or program is calculated using the sum of the time taken by each of the executable statements in an algorithm or a program. The time required by each statement depends on the following:

1. The time required for executing it once

2. The number of times the statement is executed

The product of these two parameters gives the time required for that particular statement. Compute the execution time of all executable statements. The summation of all the execution times is the total time required for that algorithm or program.

In general, when we sum up the frequency count of all the statements, we get a polynomial.

In an analysis, we are interested in the order of magnitude of an algorithm, that is, we are interested in only those statements that have the greatest frequency count.

Big-O Notation

Given the speed of computers today, we are not concerned as much with the exact measurement of an algorithm's efficiency as we are with its general order of magnitude.

If the analysis of two algorithms shows that one executes 15 iterations while the other executes 25 iterations, then they are both so fast that we cannot see the difference.

On the other hand, if one iterates 15 times and the other iterates 1500 times, we should be concerned.

The number of statements executed in the function for *n* elements of data is a function of the number of elements, expressed as *f*(*n*).

We do not need to determine the complex measure of efficiency but only the factor that determines the magnitude. This factor is the big-O, as in 'on the order of', and expressed as *O*(*n*), that is, on the order of *n*.

The simplification of efficiency is known as the *big-O analysis*.

For example, if an algorithm is quadratic, we would say its efficiency is *O* (*n*2) or on the order of *n* squared.

The big-O notation can be derived from *f* (*n*) using the following steps:

1. In each term, set the coefficient of the term to 1.

2. Keep the largest term in the function and discard the others. The terms are ranked from

the lowest to the highest as follows:

$$\log_2 n \ldots n \ldots n \log_2 n \ldots n^2 \ldots n^3 \ldots n^k \ldots 2^n \ldots n!$$

For example,

1. To calculate the big-O notation for

$$f(n) = n * (n + 1)/2 = ½\, n^2 + ½\, n$$

We first remove all coefficients. This gives us $n^2 + n$

After removing the smaller factors, which gives us n^2, which, in big-O notation, is stated as $O(f(n)) = O(n^2)$.

2. To consider another example, let us look at the polynomial expression

$$f(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$$

We first eliminate all the coefficients as follows:

$$(n) = n^k + n^{k-1} + \ldots + n^2 + n + 1$$

The largest term in the expression is the first one, so we can say that the order of this polynomial expression is

$$O(f(n)) = O(n^k)$$

Consider the situation in which you can solve a problem in three ways: one is the linear method, another is the linear logarithmic method, and the third is the quadratic method. We should be able to analyze and select one among the many possible algorithms.

Time Complexity- Big O Notation	**Example Algorithm**
O(1) Constant	Finding Odd or Even number
O(log n) Logarithmic	Finding element on sorted array with binary search
O(n) Linear	Find minimum element in unsorted array
O(n log n) Linear Logarithmic	Sorting elements in array with merge sort
$O(n^2)$ Quadratic	Sorting array with bubble sort
$O(n^3)$ Cubic	Three variables equation solver
$O(2^n)$ Exponential	Find all subsets
O(n!) Factorial	Find all permutations of a given set/string

Data Structure - Definition

Data structures refer to data and representation of data objects within a program that is the implementation of structured relationships. A data structure is a collection of atomic and composite data types into a set with defined relationships.

A data structure is
1. a combination of elements, each of which is either as a data type or another data structure and
2. a set of associations or relationships (structures) involving the combined elements.

Most of the programming languages support several data structures. In addition, The modern programming languages allow programmers to create new data structures for an application.

We can also define *data structures* as follows:

A data structure is a set of domains D, a designated domain d Ɛ D, a set of functions F, and a set of axioms A. The triple structure (D, F, A) denotes the data structure with the following elements:

***Domain* (D)** This is the range of values that the data may have.

***Functions* (F)** This is the set of operations for the data. We must specify a set of operations for a data structure to operate on.

***Axioms* (A)** This is a set of rules with which the different operations belonging to F can actually be implemented.

4.1.1. Abstract Data Type (ADT)

An abstract data type (ADT) is a set of objects together with a set of operations. Abstract data types are mathematical abstractions; Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, reals, and Booleans are data types.

In other words, an ADT is a data declaration packaged together with the operations that are meaningful for the data type. We encapsulate the data and the operations on this data and hide them from the user.

Let us consider ADT for the **Integer.**

Abstract data type Integer Operations

zero()⟶ int

ifzero(int) ⟶ boolean

increment(int) ⟶ int

add(int, int) ⟶ int

equal(int, int) ⟶ Boolean

Rules/axioms for operations

for all x, y OE integer let

ifzero(zero())⟶ true;

ifzero(increment(zero())⟶ false

add(zero(), x) ⟶ x

add(increment(x), y) ⟶ increment(add(x, y))

equal(increment(x), increment(y) ⟶ equal(x, y)

end Integer

This is an example of the `Integer` data structure; five basic functions are defined on a set of integer data object. These functions are as follows:

1. zero()→ int — It is a function which takes no input but generates the integer zero as result. That is, its output is 0.
2. ifzero(int) → Boolean — This function takes one integer input and checks whether that number is 0 or not. It generates output of type True/False, that is, of the Boolean type.
3. increment(int) → int — This function reads one integer and produces its incremented value, that is, (integer + 1), which is again an integer.

   For example, increment(3) → 4
4. add(int, int) → int — This function reads two integers and adds them producing another integer.
5. equal(int, int) → Boolean — This function takes two integer values and checks whether they are equal or not. Again, it gives output of the True/False type. So its output is of Boolean type.

The set of axioms which describes the rules of operations is as follows:

1. ifzero(zero) → true — This axiom says that the zero() function which produces an integer zero, is checked by the ifzero()function, and ultimately the result is true.
2. ifzero(increment(zero()))→ false — The value of increment(zero) is 1 and hence ifzero(1) is false.
3. add(zero(), x) → x —This means that 0 + x = x.
4. add(increment (x), y) → increment (add(x, y))—Assuming x = 3 and y = 5, this means that add(increment (3), 5) = increment(add(3, 5)) = add(4, 5) = increment(8) = 9.
5. equal(increment(x), increment(y)) → equal(x, y)— This axiom specifies that if x and y are equal, then x + 1 and y + 1 are also equal.

4.1.2. Types of Data Structures

The various types of data structures are as follows:

1. Primitive and non-Primitive
2. Linear and non-Linear

***Primitive* data structures** define a set of primitive elements that do not involve any other elements as its subparts — for example, data structures defined for integers and characters. These are generally primary or built-in data types in programming languages.

***Non-primitive* data structures** are those that define a set of derived elements such as arrays.

A data structure is said to be *linear* if its elements form a sequence or a linear list. In a linear data structure, every data element has a unique successor and predecessor. There are two basic ways of representing linear structures in memory. One way is to have the relationship between the elements by means of pointers (links), called linked lists. The other way is using sequential organization, that is, arrays.

***Non-linear* data structures** are used to represent the data containing hierarchical or network relationship among the elements. Trees and graphs are examples of non-linear data structures. In non-linear data structures, every data element may have more than one predecessor as well as successor. Elements do not form any particular linear sequence. Figure 4.1 depicts both linear and non-linear data structures.

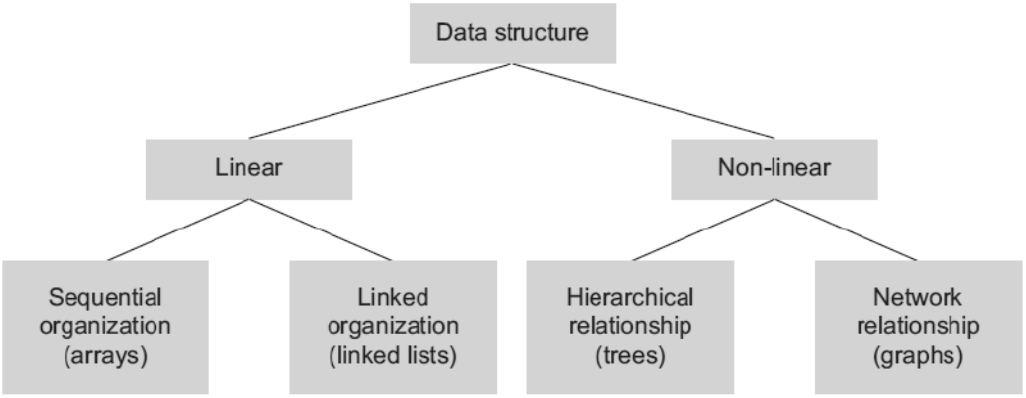


Figure 4.1. Linear and Non-Linear Data Structures

4.1.3. Basic Operations of Data Structures

All data in the data structures are processed by some specific operations. The specific data structure that has been chosen mostly depends on the number of time of the occurrence of the operation which needs to be carried out on the data structure. Names of such operations are listed below:

- Traversing - Visiting all elements at once in the data structure.
- Searching - Searching an element in the data structure.
- Insertion - Insert an element in the data structure.
- Deletion - Remove an element from the data structure.
- Sorting - Arranging all elements in the data structure.
- Merging - Merging elements from two parts of data structure.

4.2. PRIMITIVE AND COMPOSITE DATA TYPES

Data Type

A data type (in computing terms) is a set of data values having predefined characteristics. Example data types would be integer, floats, string and characters. Generally in all programming languages we have a limited number of such data types.

The range of values that can be stored in each of these data types is defined by the language and the computer hardware that you are using the high level language on.

These basic data types are also known as the **primitive data types**. The Data types are mainly categorized into three major types. These are:

1. Built in data type or Primitive Data Type
2. Derived Data type or Composite Data Type

Built in data type or Primitive Data Type

These types of data types are pre-defined and have a fixed set of rules for declaration. In other words, these data types when belong to a particular programming language has built in support and hence they are also called as built-in data types.

Examples

Such data types are:

- Integer type
- Boolean type
- Character type
- Floating type

Derived Data type or Composite Data Type

These data types can be implemented independently within a language. These data types are basically built by combining both primary and / or built-in data types and then associating operations on them.

Examples

Such data types are:

- Array
- Stack
- Queue
- List

You might be familiar with these basic data types if you have read either C or C++. For dealing with the various concepts of data structures, you can use any programming language. But it is recommended to use either C or C++ for better implementation purpose.

4.2.1. Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. The Table 4.1 lists down seven basic C++ data types are:

Type	Keyword
Boolean	**Bool**
Character	**Char**
Integer	**Int**
Floating point	**Float**
Double floating point	**Double**
Valueless	**Void**

Table 4.1. Built in Data Types

Several of the basic types can be modified using one or more of these type modifiers −

- Signed
- Unsigned
- Short
- Long

The Table 4.2 shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
Char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
Int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
Float	4bytes	+/- 3.4e +/- 38 (~7 digits)
Double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

Table 4.2. Built in Data Types and Its Memory Allocations

The size of variables might be different from those shown in the table 4.2, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

```
#include <iostream>
using namespace std;

int main() {
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
     return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

4.2.2. Composite Data Type

The data structures that are not atomic are called non primitive or composite. Examples are records, arrays and strings. These are more sophisticated. The non-primitive data structures emphasize on structuring a group of homogenous or heterogeneous data items.

Arrays: **-** An array is defined as a set of finite number of homogenous elements or data Items. It means an array can contain one type of data only, either all integers, all floating point numbers or all characters.

int a[10];

The individual element of an array can be accessed by specifying name of the array, followed by index or subscript inside square bracket.

Operation performed on array

- Creation of an array
- Traversing of an array
- Insertion of new elements
- Deletion of required elements
- Modification of an element
- Merging of arrays

Lists: **-** A list can be defined as a collection of variable number of data items. List is the most commonly used non-primitive data structures. An element of list must contain at least two fields, one for storing data or information and other for storing address of next element.

Stack: **- A** stack is also an ordered collection of elements like array but it has a special feature that deletion and insertion of elements can be done only from one end, called the top of the stack.

- It is a non-primitive data structure.
- It is also called as last in first out type of data structure (LIFO).
- Ex. Train, stack of trays etc.
- At the time of insertion or removal of an element base of stack remains same.
- Insertion of element is called Push
- Deletion of element is called Pop

Queue: **-** Queues are first in first out type of data structures. In a queue new elements are added to the queue from one end called rear end and the elements are always removed from other end called the front end.

- New elements are added to the queue from one end called REAR end
- Elements are always removed from other end called front end
- Ex. : queue of railway reservation.
- FIFO

4.3. STACK

In our everyday life, we come across many examples of stacks, for example, a stack of cards, a stack of plates, or a stack of chairs. The data structure stack is very similar to these practical examples.

Figure 4.2. Stack Examples

Definition: A *stack* is defined as a restricted list where all insertions and deletions are made only at one end, the *top.*

Elements are taken out in the reverse order of the insertion sequence. So a stack is often called *last in first out* (*LIFO*) data structure.

Given a stack S = (a_1, a_2, ..., a_n). We say that as a_1 is the bottommost element, a_n is on top of the stack, and the element a_{i+1} is said to be on the top of a_i, $1 < i \leq n$.

S = (*A*, *B*, *C*), where *A* is the bottommost element and *C* is the topmost element.

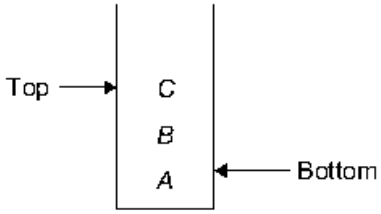


Figure 4.3. Elements of stacks

4.3.1. Array Representation of Stack

A stack can be implemented using both a static data structure (array) and a dynamic data structure (linked list). The simplest way to represent a stack is by using a one-dimensional array. A stack implemented using an array is also called a *contiguous stack*.

An array is used to store an ordered list of elements. A stack is an ordered collection of elements. Hence, it would be very simple to manage a stack when represented using an array.

The only difficulty with an array is its static memory allocation. Once declared, the size cannot be modified during run-time. This is because we declare an array to be of arbitrarily maximum size before compilation.

Let Stack[n] be a one-dimensional array. When the stack is implemented using arrays, one of the two sides of the array can be considered as the top (upper) side and the other as the bottom (lower) side.

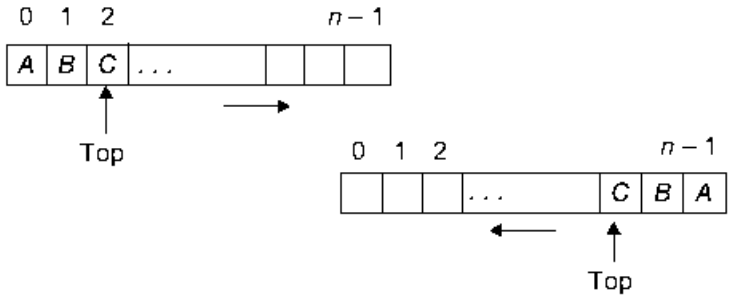


Figure 4.4. Stack using array

The elements are stored in the stack from the first location onwards. The first element is stored at the 0th location of the array Stack, which means at Stack[0], the second element at Stack[1], the i^{th} element at Stack[i - 1], and the nth element at Stack[n - 1].

Associated with the array will be an integer variable, *top*, which points to the top element in the stack. The initial value of *top* is -1 when the stack is empty. It can hold the elements from index 0, and can grow to a maximum of *n* - 1 as this is a static stack using arrays.

4.3.2. Stack Operations

Each stack abstract data type (**ADT**) has a data member, commonly named as *top*, which points to the topmost element in the stack.

There are two basic operations **push** and **pop** that can be performed on a stack; insertion of an element in the stack is called ***push*** and deletion of an element from the stack is called ***pop*.**

In stacks, we cannot access data elements from any intermediate positions other than the top position.

1. ***Push***—inserts an element on the top of the stack
2. ***Pop***—deletes an element from the top of the stack
3. ***Get Top***—reads (only reading, not deleting) an element from the top of the stack.
4. ***Stack initialization***—sets up the stack in an empty condition
5. ***IsEmpty***—checks whether the stack is empty
6. ***IsFull***—checks whether the stack is full

The simplest way to implement an ADT stack is using arrays. We initialize the variable top to -1 using a constructor to denote an empty stack. The bottom element is represented using the 0^{th} position, that is, the first element of the array. The next element is stored at the 1^{st} position and so on. The variable top indicates the current element at the top of the stack.

(1)PUSH Operation

The push operation inserts an element on the top of the stack. The recently added element is always at the top of the stack. Before every push, we must ensure whether there is a space for a new element.

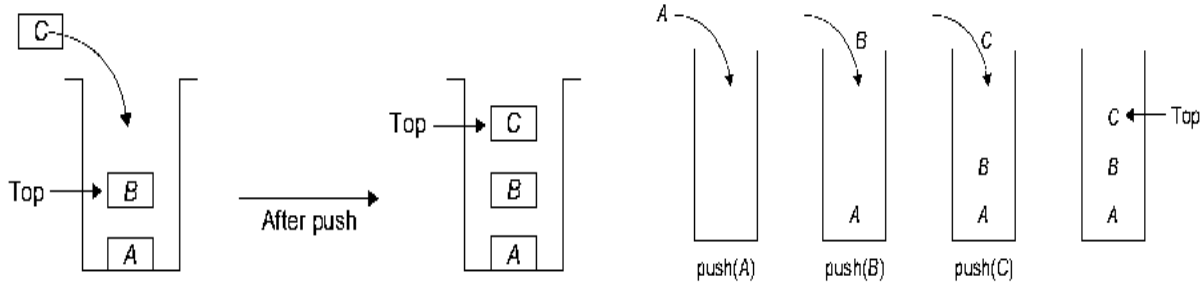


Figure 4.5. PUSH Operation

The push operation inserts an element onto the stack of maximum size MaxCapacity. Element insertion is possible only if the stack is not full. If the stack is not full, the top is incremented by 1 and the element is added on the top of the stack.

In brief,

```
if(top == MaxCapacity − 1)
cout << "Stack overflow (full)";
else
{
top ++; //increment top by one
stack[top] = Element; //add the element in new top position
}
```

(2)POP Operation

The pop operation deletes an element from the top of the stack and returns the same to the user. It modifies the stack so that the next element becomes the top element.

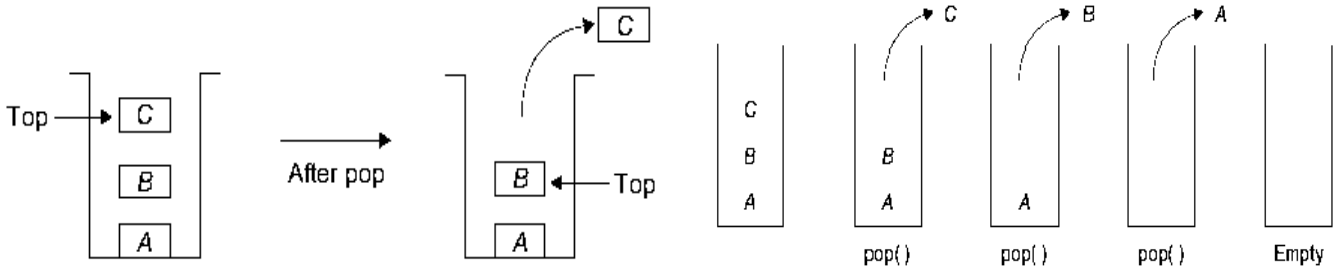


Figure 4.6. POP Operation

The pop operation deletes the element at the top of the stack and returns the same. This is done only if the stack is not empty.

If the stack is not empty, then the element at the top of the stack is returned and the top is decreased by one.

This is executed as

```
if(top == −1)
cout << "Stack underflow\n";
else
return(Stack[top−−]);
```

When there is no element available on the stack, the stack is said to be empty. If pop is performed when the stack is empty, then the stack is said to be in an underflow state.

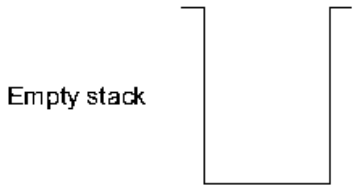


Figure 4.7. Empty Stack

(3)GETTOP Operation

The getTop operation gives information about the topmost element and returns the element on the top of the stack.

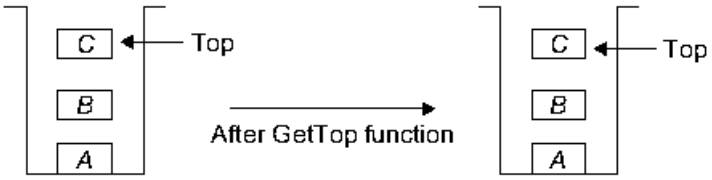


Figure 4.8. Retrieving Top Element from Stack

The getTop operation checks for the stack empty state. If the stack is empty, it reports the 'stack underflow' error message; else it returns a copy of the element that is at the top of the stack

GetTop can be described using the following statement:

```
if(top == −1)
cout << "Stack underflow (empty)" << endl;
else
return(Stack[top]);
```

This is the key difference between the pop and getTop operations. The getTop operation does not modify the variable *top.* It signals the stack underflow error if the stack is empty.

(4)Stack initialization

The following statements create an empty stack of size 100, which will hold integer values, and the variable *top* is initialized to -1.

```
int Stack[100];
int top = −1;
```

(5)IsEmpty

IsEmpty is an operation that takes the stack as an argument, checks whether it is empty or not, and returns the Boolean value *true* or *false*, respectively. The stack empty state can be checked by comparing the value of *top* with the value -1, because *top* = -1 represents an empty stack.

```
if(top == −1)
return 1;
else
return 0;
```

As both insert and delete operations are allowed only at one end of the stack, it retrieves data in the reverse order in which the data is stored. In Figure 4.9, let S = {A, B, C}.

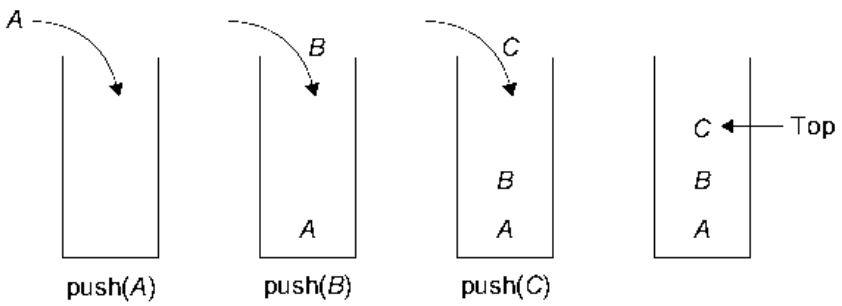


Figure 4.9. Stack and PUSH Operations

Suppose that the order of the operations is push(A), push(B), and then push(C). When we remove these elements out of the stack, they will be removed in the order C, B, and then A. This is shown in Figure. 4.10.

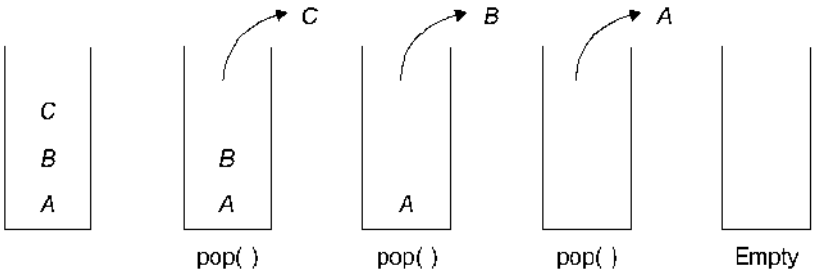


Figure 4.10. Stack and POP Operations

4.4. LINKED STACK

A stack implemented using a linked list is also called **linked stack**. Each element of the stack will be represented as a **node** of the list. The addition and deletion of a node will be only at one end. The first node is considered to be **at the top of the stack**, and it will be pointed to by a pointer called top. The last node is the bottom of the stack, and its link field is set **to Null**. An empty stack will have Top = Null.

A linked stack with elements (*X*, *Y*, Z) in order (*X* on top) may be represented as:

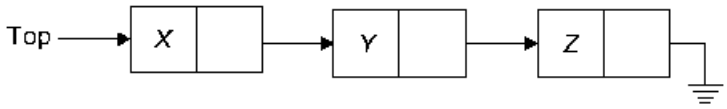
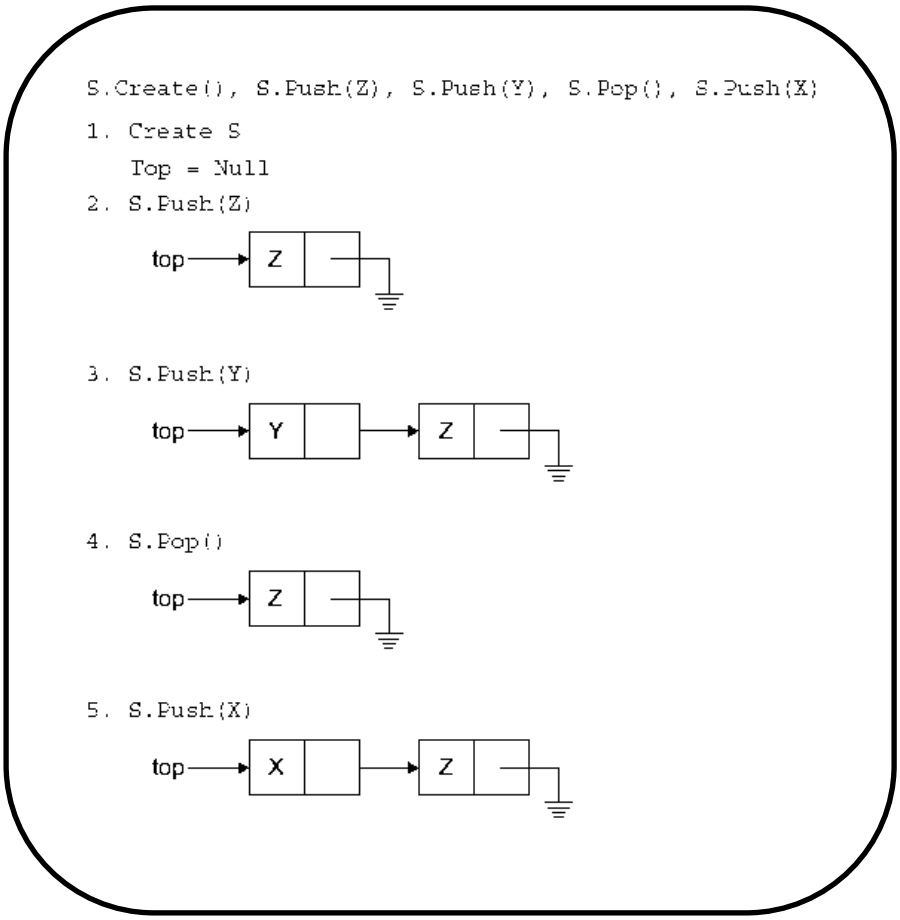


Figure 4.11. Linked Stacks

4.4.1. Linked Stack ADT



4.4.2. Operations on Linked Stack in C++

We have designed the functions for operations on stack, where the stack is implemented using linked organization. The Top is initialized to Null to indicate empty stack. The *Push( )* function dynamically creates a new node. After creating a new node, the pointer variable Top should point to the newly added node in the stack.

```
class Stack_Node
{
public:
int data;
Stack_Node *link;
};

class Stack
{
private:
Stack_Node *Top;
int Size;
int IsEmpty();
public:
Stack()
{
Top = Null;
Size = 0;
}
int GetTop();
int Pop();
void Push(int Element);
int CurrSize();
};
int Stack :: IsEmpty()
{
if(Top == Null)
return 1;
else
return 0;
}
```

```
int Stack :: GetTop()
{
if(!IsEmpty())
return(Top->data);
}

void Stack :: Push(int value)
{
Stack_Node* NewNode;
NewNode = new Stack_Node;
NewNode->data = value;
NewNode->link = Null;
NewNode->link = Top;
Top = NewNode;
}
int Stack :: Pop()
{
Stack_Node* tmp = Top;
int data = Top->data;
if(!IsEmpty()){
Top = Top->link;
delete tmp;
return(data);
}
void main()
{
Stack S;
S.Push(5);
S.Push(6);
cout << S.GetTop()<<endl;
cout << S.Pop()<<endl;
S.Push(7);
cout << S.Pop()<<endl;
cout << S.Pop()<<endl;
}
```

Output

```
6
6
7
5
```

4.5. APPLICATIONS OF STACK

The stack data structure is used in a wide range of applications. A few of them are the following:

1. Converting infix expression to postfix and prefix expressions
2. Evaluating the postfix expression
3. Checking well-formed (nested) parenthesis
4. Reversing a string
5. Processing function calls
6. Parsing (analyse the structure) of computer programs
7. Simulating recursion
8. In computations such as decimal to binary conversion
9. In backtracking algorithms (often used in optimizations and in games)

4.5.1. Arithmetic Expression

When high level programming languages came into existence, one of the major difficulties faced by computer scientists was to generate machine language instructions that could properly evaluate any arithmetic expression.

The most frequent application of stacks is in the evaluation of arithmetic expressions. An arithmetic expression is made of *operands*, *operators*, and *delimiters*.

Example: X = (A/B + C XD - F X G/Q)

Let us see the difficulties in understanding the meaning of expressions. The first problem in understanding the meaning of an expression is to decide the order in which the operations are to be carried out.

This demands that every language must uniquely define such an order. While evaluating an expression, the following operation precedence is usually used, the following operators are written in descending order of their precedence:

Arithmetic, boolean, and relational operators	Priority
^, Unary +, Unary – , ~	1
×, /	2
+, –	3
<, ≤, =, ≠, ≥, >	4
AND	5
OR	6

Table 4.3 Order of Operator Precedence

When there are two adjacent operators with the same priority, again the question arises as to which one to evaluate first. For example, the expression, *A* + *B* - *C* can be understood in two ways—(*A* + *B*) - *C* or *A* + (*B* - *C*).

This needs a decision on whether to evaluate the expression from right to left or left to right. Expressions such as *A* + *B* - *C* and *A* \ *B*/*C* are to be evaluated from left to right. However, the expression A ^ B ^ C is to be evaluated from right to left as A ^ (B ^ C). For example, to compute 2 ^ 3 ^ 2, we need to represent and evaluate it as 2 ^ (3 ^ 2). When evaluated from left to right, the expression may be evaluated as ((2 ^ 3) ^ 2), which is wrong! Hence, the operators need to decide on a rule for proceeding from left to right for all expressions except the operator exponential.

This order of evaluation, from left to right or right to left, is called *associativity*. Exponentiation is right associative and all other operators are left associative. When we write a parenthesized expression, these rules can be overridden. In the parenthesized expressions, the innermost parenthesized expression is evaluated first.

Still the question remains as to how a compiler can accept such an expression and produce the correct code. The solution is to rework on the expression to a form called the ***postfix notation***.

4.5.2. Polish Notation and Expression Conversion

The Polish Mathematician Han Lukasiewicz suggested a notation called *Polish notation***,** which gives two alternatives to represent an arithmetic expression, namely the *postfix* and *prefix* notations.

The fundamental property of Polish notation is that the order in which the operations are to be performed is determined by the positions of the operators and operands in the expression.

Hence, the advantage is that parentheses is not required while writing expressions in Polish notation.

The conventional way of writing the expression is called ***infix***, because the binary operators occur between the operands, and unary operators precede their operand. For example, the expression $((A + B) \setminus C)/D$ is an infix expression.

In **postfix notation**, the operator is written after its operands, whereas in **prefix notation**, the operator precedes its operands.

Infix	Prefix	Postfix
(operand)(operator)(operand)	(operator)(operand)(operand)	(operand)(operand)(operator)
$(A + B) \times C$	$\times + ABC$	$AB - C\times$

Table 4.4. Types of Expressions

4.5.3. Need for Prefix and Postfix Expressions

We just studied that evaluation of an infix expression using a computer needs proper code generation by the compiler without any ambiguity and is difficult because of various aspects such as the operator's priority and associativity. This problem can be overcome by writing or converting the infix expression to an alternate notation such as the prefix or the postfix. The postfix and prefix expressions possess many advantages as follows:

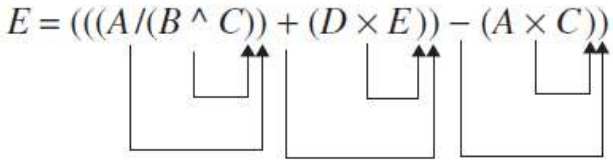
1. No need for parenthesis as in an infix expression
2. The priority of operators is no longer relevant.
3. The order of evaluation depends on the position of the operator but not on priority and associativity.
4. The expression evaluation process is much simpler than attempting a direct evaluation from the infix notation.

4.5.4. Infix to Postfix Expression Conversion

The evaluation of a postfix expression is simple, but now we need to convert an infix expression to its postfix form. Let us consider an example $E = A/B \wedge C + D \setminus E - A \setminus C$.

Let us fully parenthesize the same as $E = (((A/(B \wedge C)) + (D \setminus E)) - (A \setminus C))$

Let us move all operators to the corresponding right parenthesis and replace the same.



Now let us eliminate all parentheses. We get the postfix equivalent of the infix expression.

$$E(\text{postfix}) = A\ B\ C\ \wedge /\ D\ E\ \setminus +\ A\ C\ \setminus -$$

This procedure is a suitable method to manually convert the expression only. Let us try to work out the algorithm to convert an infix to a postfix

```
1. Scan expression E from left to right, character by character, till
   character is '#'
      ch = get_next_token(E)
2. while(ch != '#'}
      if(ch = ')') then ch = pop()
         while(ch !='(')
            Display ch
            ch = pop()
         end while
      if(ch = operand) display the same
      if(ch = operator) then
         if(ICP > ISP) then push(ch)
         else
            while(ICP <= ISP)
               pop the operator and display it
            end while
         ch = get_next_token(E)
   end while
3. if(ch = #) then while(!emptystack()) pop and display
4. stop
```

Let us convert the following infix expression to its postfix form:

A ^ B \ C - C + D/A/(E + F)

Character scanned	Stack contents	Postfix expression
A	Empty	*A*
^	^	*A*
B	^	*AB*
×	×	*AB^*
C	×	*AB ^ C*
–	–	AB ^ *C* ×
C	–	*AB ^ C* × C
+	+	*AB ^ C* × *C*–
D	+	*AB ^ C* × *C* – *D*
/	+/	*AB ^ C* × *C* – *D*
A	+/	*AB ^ C* × *C* – DA
/	+/	*AB ^ C* × *C* – *DA*/
(	+/(	*AB ^ C* × *C* – *DA*/
E	+/(	*AB ^ C* × *C* – *DA*/*E*
+	+/(+	*AB ^ C* × *C* – *DA*/*E*
F	+/(+	*AB ^ C* × *C* – *DA*/*EF*
)	+/	*AB ^ C* × *C* – *DA*/*EF*+
	Empty	*AB ^ C* × *C* – *DA*/*EF*+/+

Table 4.5. Infix to Postfix Conversion of the Expression *A ^ B \ C - C + D/A/(E + F)*

Infix Expression:

A ^ B \ C - C + D / A / (E + F)

Postfix Expression:

A B ^ C X C – D A / E F + / +

4.6. QUEUE

A queue is a common example of a linear list or an ordered list where data can be inserted at and deleted from different ends.

The end at which data is inserted is called the *rear* and that from which it is deleted is called the *front*. These limits guarantee that the data is processed in the sequence in which they are entered. In short, a queue is a *first in first out* (FIFO) list.

Consider an ordered list $L = \{a1, a2, a3, a4, \ldots, an\}$. If we assume that L represents a queue, then a_1 is the front-end element and a_n is the rear-end element. In addition, a_i is behind a_{i-1}.

Let us consider a queue Q of customers standing at a ticket counter. Q = {Sam, Arun, Shanthi, Vishal, Shiva, Abi, Deva, Anu}. In the queue Q, Sam is at the front end and Anu is at the rear end.

4.6.1. Queue as ADT

The basic operations performed on the queue include adding and deleting an element, traversing the queue, checking whether the queue is full or empty, and finding who is at the front and who is at the rear ends.

A minimal set of operations on a queue is as follows:

1. create()—creates an empty queue, *Q*
2. add(i,Q)—adds the element *i* to the rear end of the queue, *Q* and returns the new queue
3. delete(Q)—takes out an element from the front end of the queue and returns the resulting queue
4. getFront(Q)—returns the element that is at the front position of the queue
5. Is_Empty(Q)—returns true if the queue is empty; otherwise returns false

Since a queue is a linear data structure, it can be implemented using either arrays or linked lists. For the array, we use static memory allocation and for the linked list, we use dynamic memory allocation.

4.6.2. Queue Using Array

Let us see the implementation of the various operations on the queue using arrays.

1.***Create:*** This operation should create an empty queue. Here **max** is the maximum initial size that is defined.

```
#define max 50
int Queue[max];
int Front = Rear = −1;
```

2. ***Add:*** This operation adds an element in the queue if it is not full. As Rear points to the last element of the queue, the new element is added at the (rear + 1)th location.

```
void Add(int Element)
{
if(Is_Full())
cout << "Error, Queue is full";
else
Queue[++Rear] = Element;
}
```

3. ***Delete:*** This operation deletes an element from the front of the queue and sets Front to point to the next element. Front can be initialized to one position less than the actual front. We should first increment the value of Front and then remove the element.

```
int Delete()
{
if(Is_Empty())
cout << "Sorry, queue is Empty";
else
return(Queue[++Front]);
}
```

4. ***getFront:*** The operation getFront returns the element at the front, but unlike delete, this does not update the value of Front.

```
int getFront()
{
if(Is_Empty())
cout << "Sorry, queue is Empty";
```

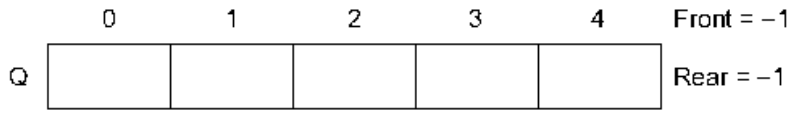
5. ***Is_Empty:*** This operation checks whether the queue is empty or not. This is confirmed by comparing the values of Front and Rear. If Front = Rear, then Is_Empty returns true, else returns false.

```
bool Is_Empty()
{
if(Front == Rear)
return 1;
else
return 0;
}
```

6. ***Is_Full:*** We need to check the state of the queue for being full. It is recommended that before insertion, the queue must be checked for the Queue_Full state. When Rear points to the last location of the array, it indicates that the queue is full, that is, there is no space to accommodate any more elements.

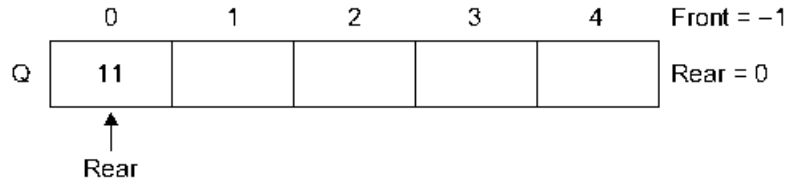
```
bool Is_Full()
{
if(Rear == max − 1)
return 1;
else
return 0;
}
```

Let *Q* be an empty queue with Front = Rear = -1. Let max = 5.

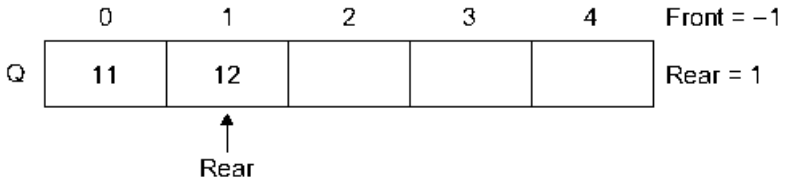


Consider the following statements:

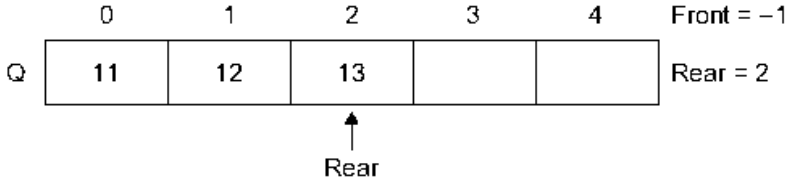
1. Q.Add(11)



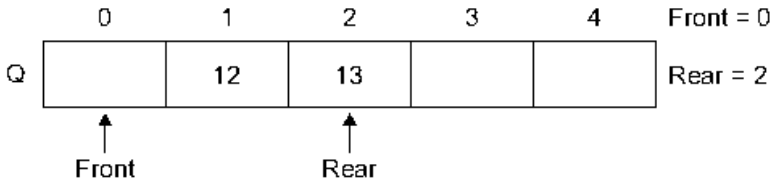
2. Q.Add(12)



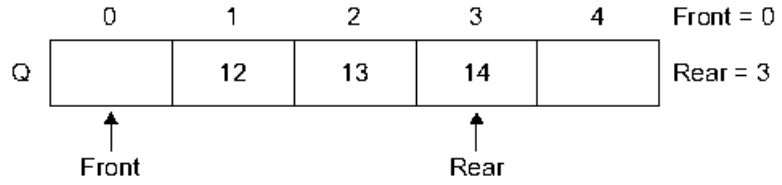
3. Q.Add(13)



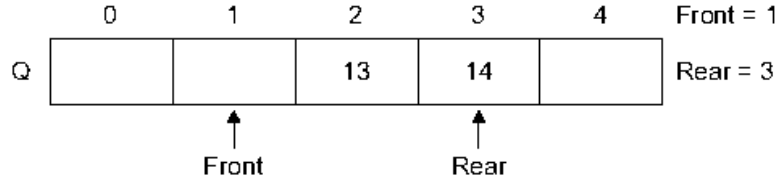
4. A = Q.Delete()
   Here, A = Q[++Front] = Q[0] = 11



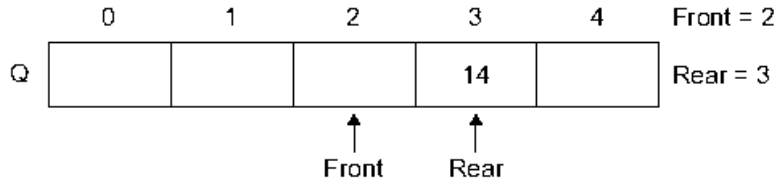
5. Q.Add(14)



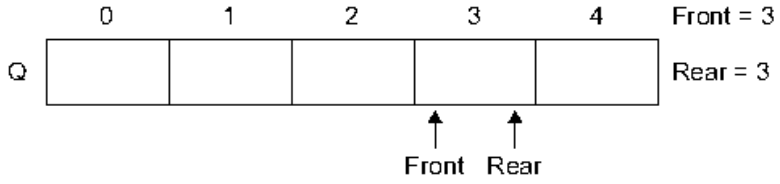
6. A = Q.Delete()
   A = Q[++ Front] = Q [1] = 12



7. A = Q.Delete()
   A = 13

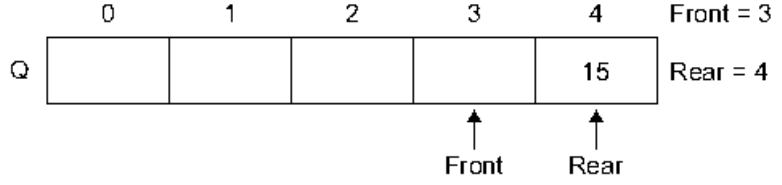


8. A = Q.Delete()



9. A = Q.Delete()
   Here we get the Queue_empty error condition as Front = Rear = 3
   Let us execute a few more statements.

10. Q.Add(15)



11. Q.Add(16)
    This statement will generate the message Queue full, because Rear = 4.

C++ Program to implement Queue ADT:

```
//Queue ADT
class queue
{
private:
int Rear, Front;
int Queue[50];
int max;
int Size;
public:
queue()
{
Size = 0; max = 50;
Rear = Front = −1 ;
}
int Is_Empty();
int Is_Full();
void Add(int Element);
int Delete();
int getFront();
};

int queue :: Is_Empty()
{
if(Front == Rear)
return 1;
else
return 0;
}
int queue :: Is_Full()
{
if(Rear == max − 1)
return 1;
else
return 0;
}
void queue :: Add(int Element)
{
if(!Is_Full())
Queue[++Rear] = Element;
Size++;
}
int queue :: Delete()
{
if(!Is_Empty())
{
Size−−;
return(Queue[++Front]);
}
}
int queue :: getFront()
{
if(!Is_Empty())
return(Queue[Front + 1]);
}

void main(void)
{
queue Q;
Q.Add(11);
Q.Add(12);
Q.Add(13);
cout << Q.Delete() << endl;
Q.Add(14);
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
Q.Add(15);
Q.Add(16);
cout << Q.Delete() << endl;
}
```

4.7. LINKED QUEUE

The direction link for nodes is to facilitate easy insertion and deletion of nodes. One can easily add a node at the rear and delete a node from the front.

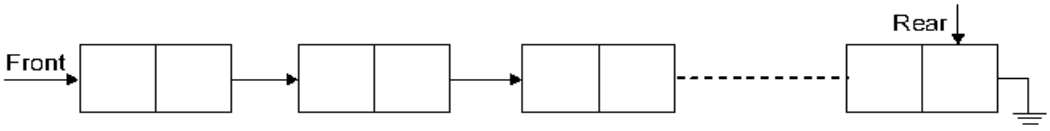


Figure 4.12. Linked Queue

Let us consider the following node structure for studying the linked queue and operations:

```
class QNode
{
public:
int data;
QNode *link;
};
class Queue
{
QNode *Front, *Rear;
int IsEmpty();
public:
Queue()
{
Front = Rear = Null;
}
void Add( int Element);
int Delete();
int FrontElement();
~Queue();
};
```

The following code will check the Isempty status of the Queue.

```
int Queue :: IsEmpty()
{
if(Front == Null)
return 1;
else
return 0;
}
int Queue :: GetFront()
{
if(!IsEmpty())
return(Front->data);
}
```

The following program code shows the addition of an element to a linked queue.

```
void Queue :: Add(int x)
{
QNode *NewNode;
NewNode = new QNode;
NewNode->data = x;
NewNode->link = Null;
// if the new is a node getting added in empty queue
//then front should be set so as to point to new
if(Rear == Null)
{
Front = NewNode;
Rear = NewNode;
}
else
{
Rear->link = NewNode;
Rear = NewNode;
}
```

Delete() function first verifies if there is any data element in the queue. If there is an element, Delete() gets and returns the data at the front of the queue to the caller function. Then, the front is set to point to the new queue front node, which is next to the node being deleted.

If the last node is being deleted, then the deleted node's next pointer is guaranteed to be Null. Note that if the current deletion of a node results in queue empty state, then along with the front, the rear should also be set to Null.

```
int Queue :: Delete()
{
int temp;
QNode *current = Null;
if(!IsEmpty())
{
temp = Front->data;
current = Front;
Front = Front->link;
delete current;
if(Front == Null)
Rear = Null;
return(temp);
}
}
```

FrontElement() returns the data element at the front of the queue. Here, the front is not updated. FrontElement() just reads what is at front.

```
int Queue :: FrontElement()
{
if(!IsEmpty())
return(Front->data);
}
```

```
void main()
{
Queue Q;
Q.Add(11);
Q.Add(12);
Q.Add(13);
cout << Q.Delete() << endl;
Q.Add(14);
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
Q.Add(15);
Q.Add(16);
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
}
```

Output

```
11
12
13
14
15
16
```

4.8. SINGLY LINKED LIST

Linked list is the most commonly used data structure used to store similar type of data in memory. Each element in the list is referred as a node.

A list with one link field using which every element is associated to its successor is known as a ***singly linked list*** **(SLL).**

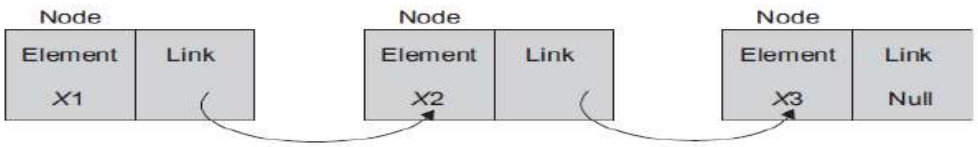


Figure 4.13. Singly Linked List

Each node of the linked list has at least the following two elements:

1. The data member(s) being stored in the list.
2. A pointer or link to the next element in the list.

4.8.1. Linked List Terminology

Header node: A header node is a special node that is attached at the beginning of the linked list. This header node may contain special information (metadata) about the linked list.

Data node: The list contains data nodes that store the data members and link(s) to its predecessor (and/or successor).

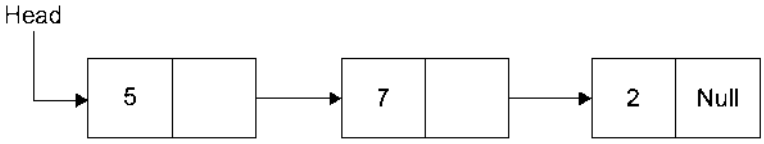


Figure 4.14. Singly Linked List with Head and Null

Tail pointer Similar to the head pointer that points to the first node of a linked list, we may have a pointer pointing to the last node of a linked list called the *tail pointer*.

4.8.2. Primitive Operations of Singly Linked List

The following are basic operations associated with the linked list as a data structure:

1. Creating an empty list
2. Inserting a node
3. Deleting a node
4. Traversing the list

To represent this linked list, we consider it as an object of class LList whose definition is as follows:

```
class LList
{
private:
Node *Head;
Node *Tail; // optional data members
int Size;
public:
LList()
{
        Head = Tail = Null;
        Size = 0;
}
void Create();
void Traverse();
void Insert( int data, position);
void Append(int data);
void Delete(int position);
void Reverse();
};
```

4.8.3. Insertion of a node

Insertion of a Node at a Middle Position

Assume that a node is to be inserted at some position other than the first position. Let Prev refer to the node after which NewNode node is to be inserted.

We need the following two steps:

```
NewNode->link = Prev->link;
Prev->link = NewNode;
```

The node NewNode is to be inserted between Prev and the successor of Prev. The link manipulation required to accomplish this is shown in Figure 4.15 with dotted lines.

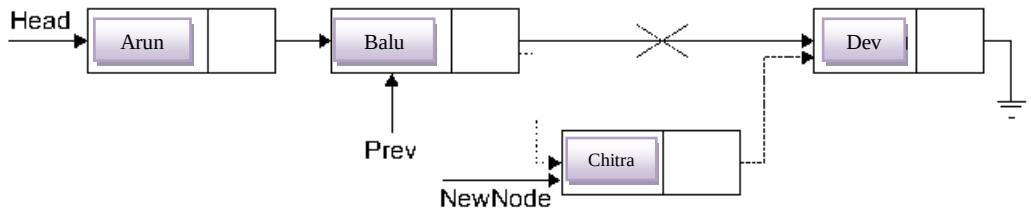


Figure 4.15. Inserting a Node at Middle Position

Insertion of a Node at the First Position

Let us consider a situation when the node is to be inserted at the first position. As per the steps discussed for insertion of a node at the middle, we need Prev, which is a pointer to the node after which NewNode is to be added.

To insert a node at the first position, there exists no Prev node. The following two steps will insert NewNode at the beginning of the linked list.

```
NewNode->link = Head;
Head = NewNode;
```

The link manipulations needed to add a node at the first location is shown in following figure 4.16 using dotted lines.

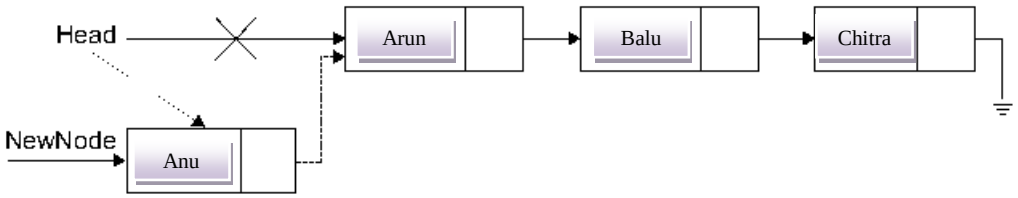


Figure 4.16. Inserting a Node as First Node

Insertion of a Node at the End

The steps for inserting a node in the middle of a list also work for inserting a node at the end of the list. As the node is to be inserted after the last node, Prev is a pointer to the last node.

Let the node to be inserted be NewNode as shown in Figure 4.17. The following two steps will insert NewNode at the end of the linked list.

```
NewNode->link = Prev->link
Prev->link = NewNode;
```

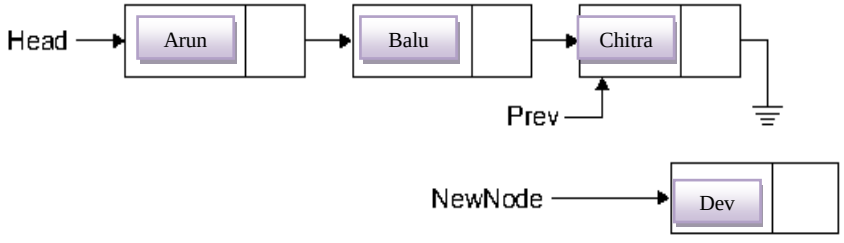


Figure 4.17. Inserting a Node at End

4.8.4. Deletion of a Node

Let us assume that the node to be deleted contains data *x*. We need the following steps to delete the same. Let *x* = 13 and let it be pointed to by the pointer Curr. To delete this node, the required link manipulations are shown in following figure with dotted lines.

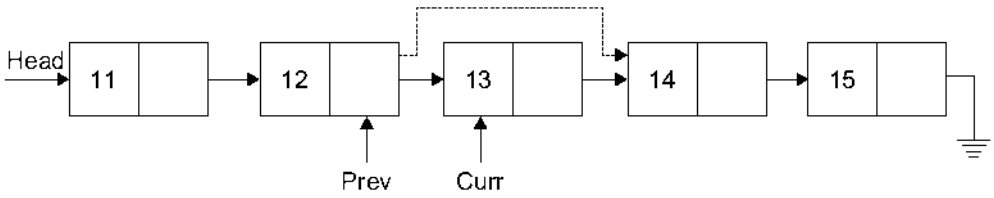


Figure 4.18. Deleting a Node from a List

To delete the node Curr, we need to modify the link between Curr and its previous node, and the link between Curr and its successor.

We need to modify them as shown in Figure 4.16. The Prev is pointing to Curr as its current successor. As the Curr is to be deleted, the Prev's link should be modified such that it points to the successor of Curr. This makes the successor of Curr the successor of Prev. This deletes the node Curr from the linked list.

Note that we need the address of the node to be deleted as well as its predecessor to modify the links such that the node is deleted. This can be achieved by the following steps:

1. Let both Curr and Prev be set to Head.
2. Traverse the list and search the node to be deleted.
3. Let Curr point to the node to be deleted and Prev be its previous node.
4. Modify the link field of Prev so that it skips Curr and points to its next.

   Prev->link = Curr->link
5. Free the memory allocated for the node Curr.
6. Stop

The node to be deleted can be at any position. It could be the first, middle, or last node.

Deleting the First Node

Deleting the first node is also referred to as deleting a header node. If the node at the first position is to be deleted, then we need to modify the pointer pointing to the first node (also called as the head pointer), say Head.

Deletion of the first node needs the link manipulations shown in Fig. 4.19 with dotted lines.

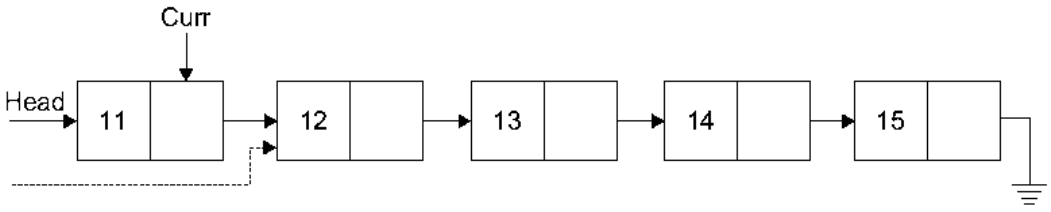


Figure 4.19. Deleting First Node from a List

We should also release the first node using the delete operator. Hence, this can be accomplished in two steps as

1. Set another pointer to the first node before modifying Head, which is the pointer pointing to the first node. Set Head to point to the second node. This can be accomplished by the statements,

```
Curr = Head;
Head = Head->link;
```

2. Now, release the memory allocated for the fi rst node.

```
delete Curr;
```

These two statements will delete the fi rst node, and Head will point to the second node so that the second node becomes the fi rst node. Later, the memory allocated for the first node is freed.

Deleting a Middle Node

Let curr point to the node to be deleted, and prev be the predecessor of curr. Then, the following statements will delete the node curr.

```
prev->link = curr->link;
delete curr;
```

These two statements will also delete the last node of the list.

4.8.5. Linked List Traversal

List traversal is the basic operation where all elements in the list are processed sequentially, one by one. To traverse the linked list, we have to start from the first node.

We can access the first node through a pointer variable Head. Once we access the first node, through its link field, we can access the second node; through the second node's link field, we can access the third, and so on, as every node points to its successor till the last node.

List Traversal Algorithm

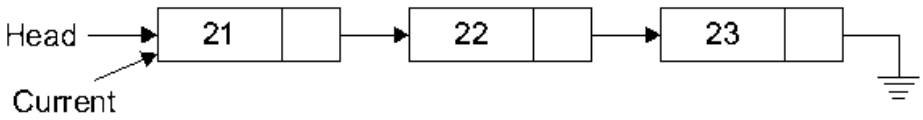
1. Get the address of the first node, call it current; current = Head.
2. if current is Null, goto step 6.
3. Process the data field of the current node (node pointed by current). Here, the process may include printing data, updating, and so on.
4. Move to the next node–current = current->link (Now current should point to the next node. The address of next node is in the link field of current. Hence, set current to the link field of current}
5. goto step 2
6. stop

Let us see output for list *L* pictorially.

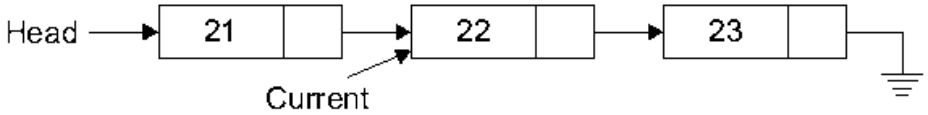
L = {21, 22, 23}

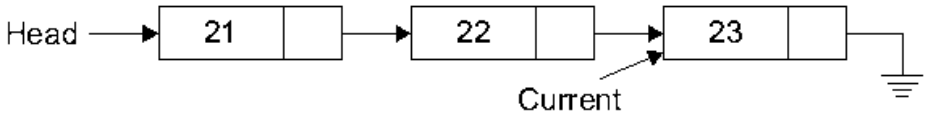
1. Current = Head

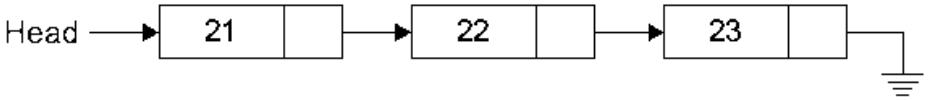
2. After execution of statement 1



3. As current!= Null, statements 3, 4, and 5 are executed.







Now Current = Null is true, while loop condition is false; hence stop.

Output

21

21 22

21 22 23

4.9. DOUBLY LINKED LIST

In a doubly linked list (DLL), each node has two link fields to store information about the one to the next and also about the one ahead of the node. In DLL, from every node, the list can be traversed in both the directions.

We can use DLLs where each node contains two links, one to its predecessor and other to its successor.

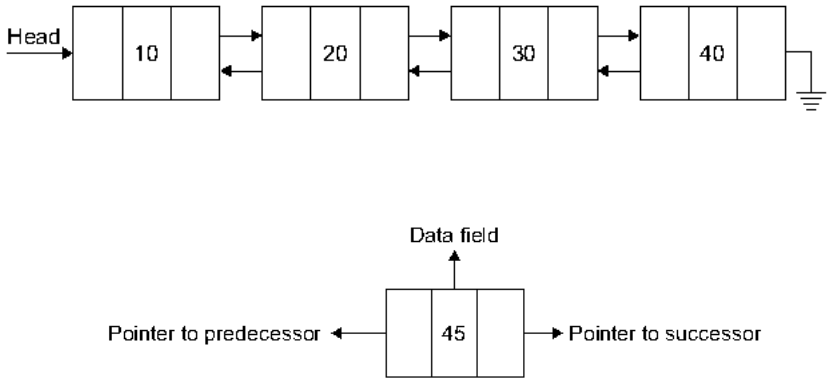


Figure 4.20. Node structure of doubly linked list

The following program code shows the class of a doubly linked list node.

```
class DLL_Node
{
Public:
int Data;
DLL_Node *Prev, *Next;
DLL_Node()
{
Prev = Next = Null;
}
};
```

A DLL may either be linear or circular and it may or may not contain a header node. DLLs are also called *two-way lists*.

4.9.1. Deletion of a node from a doubly Linked List

Deleting from a DLL needs the deleted node's predecessor, if any, to be pointed to the deleted node's successor. In addition, the successor, if any, should be set to point to the predecessor node as shown in following figure.

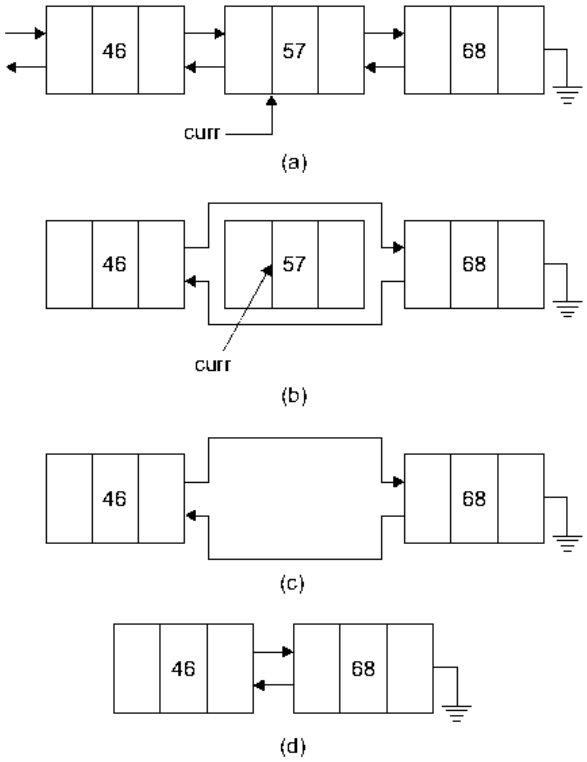


Figure 4.21. Deletion node in doubly linked list (a) Links modified on deletion of node (b) Memory of the deleted node freed (c) Realignment of nodes (d) After node deletion.

The core steps involved in this process are the following:

```
(curr->Prev)->Next = curr->Next;
(curr->Next)->Prev = curr->Prev;
delete curr;
```

C++ Program Code – Deletion of Nodes in DDL

```
void DList :: DeleteNode(int val)
{
DLL_Node *curr, *temp;
curr = Head;
while(curr!=Null)
{
if(curr->Data == val)
break;
// curr is pointing to the node to be deleted
curr = curr->Next;
}
if(curr != Null)
{
if(curr == Head) // delete first node
{
Head = Head->Next;
Head->Prev = Null;
delete curr;
}
else
{
if(temp == Tail) // delete last node
{
Tail = temp->Prev;
(temp->Prev)->Next = Null;
delete temp;
}
else
{
(curr->Prev)->Next = curr->Next;
(curr->Next)->Prev = curr->Prev;
delete curr;
}
}
if(Head == Null)
{
Tail = Null;
}
}
else
cout << "Node to be deleted is not found \n";
}
```

4.9.2. Insertion of a node in a doubly Linked List

Let us discuss inserting a node in DLL. To insert a node, say Current, we have to modify four links as each node points to its predecessor as well as successor. Let us assume that the node Current is to be inserted in between the two nodes say node1 and node2.

We have to modify the following links:
node1->Next, node2->Prev, Current->Prev, and Current->Next

When the Current node is inserted in between node1 and node2, node1's successor node changes. Hence, we need to modify node1->Next. For the node node2, its predecessor changes. Therefore, we need to modify node2->Prev. This is shown in following figure 4.22.

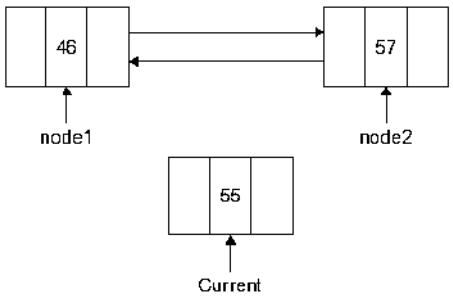


Figure 4.22. Inserting a First Node Current

Current is a new node to be inserted. We need to set both its predecessor and successor by setting the links as Current->Prev and Current->Next. After the insertion of Current, the resultant modified links should be shown as in following figure 4.23.

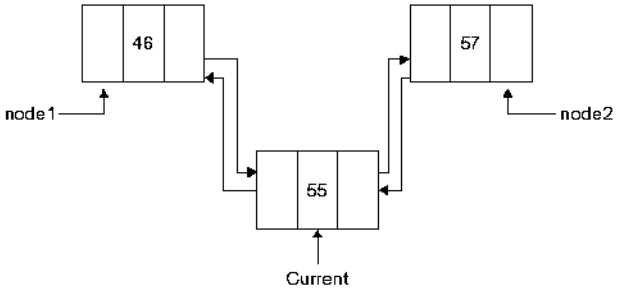


Figure 4.23. Link modification for insertion of a node in a DLL

Hence, to modify the links, the statements would be

1. To modify node1->Next we use the operation

   node1->Next = Current;

2. To modify node2->Prev we use the operation

   node2->Prev = Current;

3. To set curr->Next, we use the operation

   Current->Next = node2;

4. To set curr->Prev, we use the operation

   Current->Prev = node1;

In brief, the statements to insert a node in between node1 and node2 are as follows:

```
node1->Next = Current;
node2->Prev = Current;
Current->Next = node2;
Current->Prev = node1;
```

These statements are with respect to Fig. 4.21, where we considered that the node is to be inserted in between node1 and node2.

Though the new node is to be inserted between node1 and node2, we need to know only about node1. The node2 is the successor of node1, which can be accessed through node1->Next. Practically, the node can be inserted in DLL given only one node after which (or before which) the node is to be inserted.

Let us consider the insertion of a node given one node before or after which the node is to be inserted, say before node2. Then, the four statements could be

```
(node2->Prev)->Next = Current;
Current->Prev = node2->Prev;
Current->Next = node2;
node2->Prev = Current;
```

In brief, a node can be inserted anywhere in the DLL given a node after/before which it is to be inserted. The function can be written by passing to it either a node after/before which to insert or the position where to insert. One of the parameters would be the node to be inserted. Let us see how to insert a node at the fi rst position. We are given a pointer to the DLL say Head.

This is represented by the following statements:

```
Current->Next = Head;
Head->Prev = Current;
Head = Current;
Current->Prev = Null;
```

We have to modify the links as shown in Figure. 4.24.

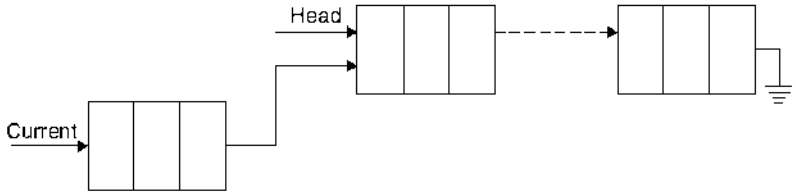


Figure 4.24. Inserting a node before first node

4.10. APPLICATION OF SINGLY LINKED LIST (Polynomial Addition)

The manipulation of symbolic polynomials is a good application of list processing. Let the polynomial we want to represent using a linked list be $A(x)$. It is expanded as,

$$A(x) = k_1x^m + \ldots + k_{n-1}x^2 + k_nx^1$$

where k_i is a non-zero coefficient with exponent m such that $m > m - 1 > \ldots > 2 > 1 \geq 0$.

A node of the linked list will represent each term. A node will have 3 fields, which represent the coefficient and exponent of a term and a pointer to the next term (Figure. 4.25).

Figure 4.25. Polynomial Node

For instance, the polynomial, say $A = 6x^7 + 3x^5 + 4x^3 + 12$ would be stored as in Figure. 4.26.

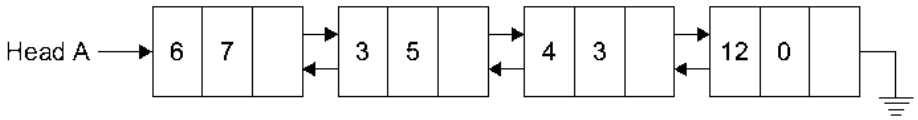


Figure 4.26. Polynomial A= $6x^7 + 3x^5 + 4x^3 + 12$

The polynomial $B = 8x^5 + 9x^4 - 2x^2 - 10$ would be stored as in Figure. 4.27.

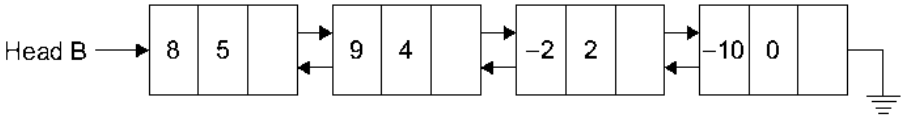


Figure 4.27. Polynomial $B = 8x^5 + 9x^4 - 2x^2 - 10$

The function for the creation of a polynomial can be written as follows:

The difference is the data field we used earlier had single integer data fields, whereas here, we have two data fields and one linked field. The two data fields are the exponent and the coefficient of each term of the polynomial.

The following Program Code shows the creation of a polynomial.

```
class PolyNode
{
public:
int coef;
int exp;
PolyNode *link;
};
```

```
class Poly
{
private:
PolyNode *Head, *Tail;
public:
Poly() {Head = Tail = Null;} //
constructor
void Create();
PolyNode *GetNode();
void Display();
};
```

```
void Poly :: Create()
{
char ans;
PolyNode *NewNode;
while(1)
{
cout << "Any term to be
added? (Y/N)\n";
cin >> ans;
if(ans == 'N'|| ans == 'n')
break;
NewNode = GetNode();
if(Head == Null)
{
Head = NewNode;
Tail = NewNode;
}
}
```

```
PolyNode* Poly :: GetNode()
{
PolyNode *NewNode;
NewNode = new PolyNode;
if(NewNode == Null)
{
cout << "Error in memory
allocation \ n";
// exit(0);
}
cout << "Enter coeffi cient and
exponent";
cin >> NewNode->coef;
cin >> NewNode->exp;
NewNode->link = Null;
return(NewNode);
        }
```

4.10.1. Polynomial Addition

The polynomial *A* and *B* are to be added to yield the polynomial *C*. The assumption here is the two polynomials are stored in linked list with descending order of exponents.

The two polynomials *A* and *B* are stored in two linked lists with pointers *ptr1* and *ptr2* pointing to the first node of each polynomial, respectively.

To add these two polynomials, let us use these two pointers ptr1 and ptr2 to move along the terms of *A* and *B*.

If the exponents of the two terms are equal, then their coefficients are added and a new term is created for the resultant polynomial *C*. The pointer ptr1 and ptr2 are advanced to the next term.

The following are the steps to add two polynomials *A* and *B* to yield the polynomial *C*.

```
poly Poly :: PolyAdd(Poly P2)
{
PolyNode *Aptr = Head;
PolyNode *Bptr = P2.Head;
Poly C;
PolyNode *NewTerm;
while(Aptr != Null && Bptr != Null)
{
NewTerm = new PolyNode;
NewTerm->link = Null;
if(Aptr->exp == Bptr->exp)
{
NewTerm->coef = Aptr->coef + Bptr->coef;
NewTerm->exp = Aptr->exp;
C.Append(NewTerm);
Aptr = Aptr->link;
Bptr = Bptr->link;
}
else if(Aptr->exp > Bptr->exp)
{
NewTerm->coef = Aptr->coef;
NewTerm->exp = Aptr->exp;
C.Append(NewTerm);
Aptr = Aptr -> link;
}
else
{
NewTerm->coef = Bptr->coef;
NewTerm->exp = Bptr->exp;
C.Append(NewTerm);
Bptr = Bptr -> link;
}
} // end of while
while(Aptr != Null)
{
NewTerm = new PolyNode;
NewTerm->link = Null;
NewTerm->coef = Aptr->coef;
NewTerm->exp = Aptr->exp;
C.Append(NewTerm);
Aptr = Aptr->link;
}
while(Bptr != Null)
{
NewTerm = new PolyNode;
NewTerm->link = Null;
NewTerm->coef = Bptr->coef;
NewTerm->exp = Bptr->exp;
C.Append(NewTerm);
Bptr = Bptr->link;
}
return C;
}
```

If the exponent of the current term in *A* is less than the exponent of the current term of *B*, then a duplicate of the term in *B* is created and attached to *C*. The pointer ptr2 is advanced to the next term.

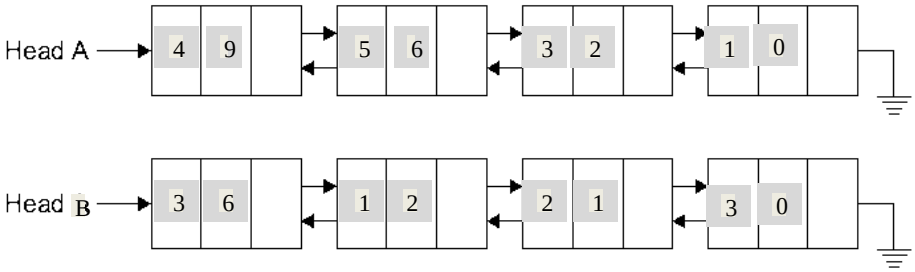
Similar action is taken on *A* if the exponent of the current term of *A* is greater than the exponent of the current term of *B*.

Each time a new node is generated, its exponent and coefficient fields are set accordingly, and the resultant term is attached to the end of the resultant term *C*. For polynomial *C*, we have ptr3 to move along the resultant polynomial *C*.

Let two polynomials *A* and *B* be

$A = 4x^9 + 5x^6 + 3x^2 + 1$

$B = 3x^6 + x^2 + 2x + 3$



Output:

C=$4x^9 + 8x^6 + 4x^2 + 2x + 4$

TWO MARKS QUESTIONS AND ANSWERS:

1. **Define Data Structures.**

   Data Structures is defined as the way of organizing all data items that consider not only the elements stored but also stores the relationship between the elements.

2. **What are Primary Data Structures?**

   Primary data structures are the basic data structures that directly operate upon the machine instructions. All the basic constants (integers, floating-point numbers, character constants, string constants) and pointers are considered as primary data structures.

3. **Differentiate static and dynamic data structures.**

   A data structure formed when the number of data items are known in advance is referred as static data structure or fixed size data structure.

   A data structure formed when the number of data items are not known in advance is known as dynamic data structure or variable size data structure.

4. **Compare Linear and Non- Linear Data Structures.**

   Linear data structures are data structures having a linear relationship between its adjacent elements. Ex. Array and Stack.

   Non-linear data structures are data structures that don't have a linear relationship between its adjacent elements but have a hierarchical relationship between the elements. Ex. Trees and Graphs.

5. **Define an Abstract Data Type (ADT).**

   An abstract data type is a set of operations. ADTs are mathematical abstractions; nowhere in an ADT's definition is there any mention of how the set of operations is implemented. Objects such as lists, sets and graphs, along with their operations can be viewed as abstract data types.

6. Define Stack.

Stack is an ordered collection of elements in which insertions and deletions are restricted to one end. The end from which elements are added and/or removed is referred to as top of the stack. Stacks are also referred as piles; push-down lists and last-in-first- out (LIFO) lists.

7. List out the basic operations that can be performed on a stack.

The basic operations that can be performed on a stack are

- Push operation
- Pop operation
- Peek operation
- Empty check
- Fully occupied check

8. List the applications of stacks.

- Towers of Hanoi
- Reversing a string
- Balanced parenthesis
- Recursion using stack
- Evaluation of arithmetic expressions

9. State the different ways of representing expressions.

The different ways of representing expressions are

- Infix Notation
- Prefix Notation
- Postfix Notation

10. List the advantages of using postfix notations.

- Need not worry about the rules of precedence
- Need not worry about the rules for right to left associativity
- Need not need parenthesis to override the above rules

11. What ate the rules to be followed during infix to postfix conversions?

- Fully parenthesize the expression starting from left to right. During parenthesizing, the operators having higher precedence are first parenthesized
- Move the operators one by one to their right, such that each operator replaces their corresponding right parenthesis
- The part of the expression, which has been converted into postfix is to be treated as single operand.

12. Define a queue.

Queue is an ordered collection of elements in which insertions are restricted to one end called the rear end and deletions are restricted to other end called the front end. Queues are also referred as First-In-First-Out (FIFO) Lists.

13. Define a priority queue.

Priority queue is a collection of elements, each containing a key referred as the priority for that element. Elements can be inserted in any order (i.e., of alternating priority), but are arranged in order of their priority value in the queue. The elements are deleted from the queue in the order of their priority (i.e., the elements with the highest priority is deleted first). The elements with the same priority are given equal importance and processed accordingly.

14. What are the types of queues?

- Linear Queues – The queue has two ends, the front end and the rear end. The rear end is where we insert elements and front end is where we delete elements. We can traverse in a linear queue in only one direction ie) from front to rear.
- Circular Queues – Another form of linear queue in which the last position is connected to the first position of the list. The circular queue is similar to linear queue has two ends, the front end and the rear end. The rear end is where we insert elements and front end is where we delete elements. We can traverse in a circular queue in only one direction ie) from front to rear.
- Double-Ended-Queue – Another form of queue in which insertions and deletions are made at both the front and rear ends of the queue.

15. Define a DEQUE.

DEQUE (Double-Ended Queue) is another form of a queue in which insertions and deletions are made at both the front and rear ends of the queue. There are two variations of a DEQUE, namely, input restricted DEQUE and output restricted DEQUE. The input restricted DEQUE allows insertion at one end (it can be either front or rear) only. The output restricted DEQUE allows deletion at one end (it can be either front or rear) only.

16. List the applications of queues

- Jobs submitted to printer
- Real life line
- Calls to large companies
- Access to limited resources in Universities
- Accessing files from file server

17. Define Linked Lists.

Linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the Pointer. The last cell's pointer points to NULL.

18. List the basic operations carried out in a linked list

The basic operations carried out in a linked list include:

- Creation of a list
- Insertion of a node
- Deletion of a node
- Modification of a node
- Traversal of the list

19. What are the advantages of using a linked list?

- It is not necessary to specify the number of elements in a linked list during its declaration
- Linked list can grow and shrink in size depending upon the insertion and deletion that occurs in the list
- Insertions and deletions at any place in a list can be handled easily and efficiently
- A linked list does not waste any memory space

20. List out the disadvantages of using a linked list.

- Searching a particular element in a list is difficult and time consuming
- A linked list will use more storage space than an array to store the same number of elements.

21. List out the applications of a linked list.

Some of the important applications of linked lists are manipulation of polynomials, sparse matrices, stacks and queues.

22. State the difference between arrays and linked lists.

Arrays	Linked Lists
Size of an array is fixed	Size of a list is variable
It is necessary to specify the number of elements during declaration	It is not necessary to specify the number of elements during declaration
Insertions and deletions are carried out easily	Insertions and deletions are somewhat difficult
It occupies less memory than a linked list for the same number of elements	It occupies more memory

23. State the difference between stacks and linked lists.

The difference between stacks and linked lists is that insertions and deletions may occur anywhere in a linked list, but only at the top of the stack

24. Mention the advantages of representing stacks using linked lists than arrays.

• It is not necessary to specify the number of elements to be stored in a stack during its declaration, since memory is allocated dynamically at run time when an element is added to the stack

• Insertions and deletions can be handled easily and efficiently

• Linked list representation of stacks can grow and shrink in size without wasting memory space, depending upon the insertion and deletion that occurs in the list

• Multiple stacks can be represented efficiently using a chain for each stack

25. List out the differences between queues and linked lists.

The difference between queues and linked lists is that insertions and deletions may occur anywhere in the linked list, but in queues insertions can be made only in the rear end and deletions can be made only in the front end.

SOLVED PROBLEMS:

1. **Assume that there are 10 books in a library, which form a specific sequence. This ordered set of 10 books is to be kept in a shelf. There are two ways to arrange the books. One of the arrangements is to keep all the 10 books in 10 continuous empty slots (similar to an array). The second possible arrangement is to place the books at available locations in a distributed manner (similar to a linked list) by keeping track of the various locations of the books. Consider an array and linked list and show their representations.**

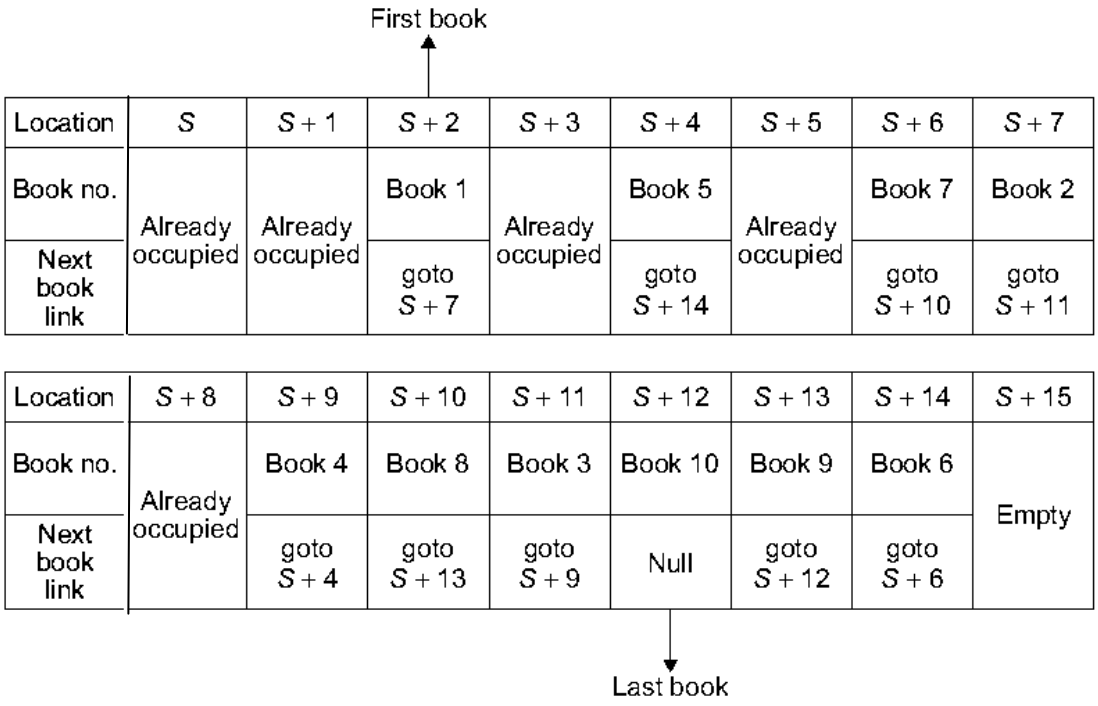
Solution:

Let Books = {book1, book2, book3, …, book10}

Array Representation:

Shelf position	*S*	*S* + 1	*S* + 2	*S* + 3	*S* + 4	*S* + 5	*S* + 6	*S* + 7	*S* + 8	*S* + 9
Book Number	Book1	Book2	Book3	Book4	Book5	Book6	Book7	Book8	Book9	Book10

Linked List Representation



2. **Consider five jobs with profits (*p*1, *p*2, *p*3, *p*4, *p*5) = (20, 15, 10, 5, 1) and maximum delay allowed (*d*1, *d*2, *d*3, *d*4, *d*5) = (2, 2, 1, 3, 3). Find an optimized high profit solution.**

Solution:

Here, the maximum number of jobs that can be completed is

Min(*n*, maxdelay(*di*)) = Min(5, 3) = 3

Hence, there is a possibility of doing 3 jobs, and there are 3 units of time, as shown in following table.

Job Scheduling

Time Slot	Profit	Job
0-1	20	1
1-2	15	2
2-3	0	3 Cannot be accommodated
2-3	5	4
Total Profit	40	

In the first unit of time, job 1 is done and a profit of 20 is gained; in the second unit, job 2 is done and a profit of 15 is obtained. However, in the third unit of time, job 3 is not available, so job 4 is done with a gain of 5. Further, the deadline of job 5 has also passed; hence three jobs 1, 2, and 4 are completed with a total profit of 40.

3. **What would be the Prefix notation for the given equation A+ (B*C).**

Solution:

Reverse the equation or scan the equation from right to left. Apply the infix-postfix algorithm. The equation inside the bracket evaluates to CB* and outside the bracket evaluates to A+ therefore getting CB*A+. Reversing this and we get +A*BC.

4. **What is the result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, *?**

Solution:

The postfix expression is evaluated using stack. We will get the infix expression as (5*(4+6))*(4+9/3). On solving the Infix Expression,

We get (5*(10))*(4+3) = 50*7 = **350**.

5. **Find the postfix form of the expression (A+ B)*(C*D- E)*F / G.**

Solution:

(((A+ B)*(C*D- E)*F) / G) is converted to postfix expression as

(AB+(*(C*D- E)*F)/ G)

(AB+CD*E-*F) / G

(AB+CD*E-*F * G/).

Thus Postfix expression is **A B + C D * E - * F * G /**

6. **Consider the following operation performed on a stack of size 5.**

Push(1);

Pop();

Push(2);

Push(3);

Pop();

Push(4);

Pop();

Pop();

Push(5);

After the completion of all operation, find the number of elements present in stack.

Solutions:

Number of elements present in stack is equal to the difference between number of push operations and number of pop operations. **Number of elements is 5-4=1.**

7. **If the elements "A", "B", "C" and "D" are placed in a queue and are deleted one at a time, in what order will they be removed?**

Solution:

Queue follows FIFO approach. i.e. First in First Out Approach. So, the orders of removal elements are A B C D.

8. What is the output of following function for *start* pointing to first node of following linked list?

1->2->3->4->5->6

```
void fun(struct node* start)
{
   if(start == NULL)
   return;
   printf("%d  ", start->data);
   if(start->next != NULL )
   fun(start->next->next);
   printf("%d  ", start->data);
}
```

Solution:

fun() prints alternate nodes of the given Linked List, first from head to end, and then from end to head. **Output is 1 3 5 5 3 1.**

9. Consider the following doubly linked list: head-1-2-3-4-5-tail, What will be the list after performing the given sequence of operations?

```
Node temp = new Node(6,head,head.getNext());
Node temp1 = new Node(0,tail.getPrev(),tail);
head.setNext(temp);
temp.getNext().setPrev(temp);
tail.setPrev(temp1);
temp1.getPrev().setNext(temp1);
```

Solution:

The given sequence of operations performs addition of nodes at the head and tail of the list.

Output : head-6-1-2-3-4-5-0-tail

10. Evaluate the infix expression a/b+c*d where a=4, b=2, c=2, d=1.

Solution:

* and / have higher priority. Hence, they are evaluated first. Then, + is evaluated.

Hence, **(4/2) + (2*1) =2+2=4.**

TREES

5.1. INTRODUCTION

A **TREE data structure** arranges the data elements in a hierarchical tree format. The data elements of the tree are referred as nodes. A node is called **parent node** if it has child nodes descending from it.

A tree *T* is defined recursively as follows:

1. A set of zero items is a tree, called the *empty tree* (or null tree).
2. If *T*1, *T*2, ..., *Tn* are *n* trees for *n* > 0 and *R* is a *node*, then the set *T* containing *R* and the trees.

Where *T*1, *T*2, ..., *Tn* are a tree. Within *T*, *R* is called the ***root*** of *T*, and *T*1, *T*2, ..., *Tn* are called ***subtrees***

5.2. BASIC TERMINOLOGY OF TREES

Root Node: A root node of a tree is a node which is the top most level node in that tree. A directed tree has one node called its *root*, with in degree zero.

Node : A node in the tree has the item of information (data) plus the branches to other nodes.

Branch node (internal node) : All other nodes whose out degree is not zero are called as *branch* nodes

Terminal node (leaf node): In a directed tree, any node that has an out degree zero is a *terminal node*. The terminal node is also called as *leaf node* (or *external node*).

Directed tree: An acyclic directed graph is a *directed tree*.

In degree: The Number of nodes which are coming into a particular node is called in degree.

Out degree: The number of nodes which are going out from a particular node is called out degree.

Level of node: The level of any node is its path length from the root. The level of the root of a directed tree is zero, whereas the level of any node is equal to its distance from the root.

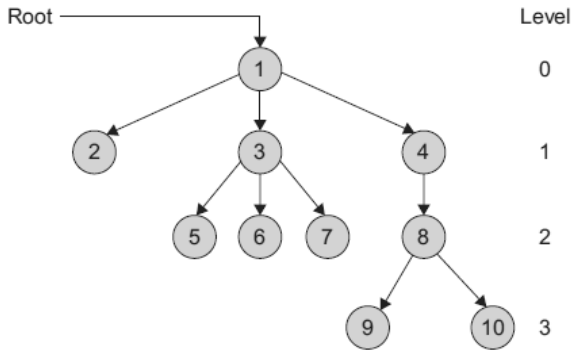


Figure 5.1. An Example of a TREE

The following is a list of terms for review using the above example tree:

Sub trees: The nodes labelled 2, 3, and 4 are the roots of the sub trees (children) of the node labelled 1. The nodes labelled 5, 6, and 7 are the roots of the sub trees (children) of the node labelled 3. The node labelled 8 is the root of the sub tree (child) of the node labelled 4. The nodes labelled 9 and 10 are the roots of the sub trees (children) of the node labelled 8.

Leaves: The nodes labelled 2, 5, 6, 7, 9, and 10 are the terminal nodes or leaf nodes.

Degree: The nodes labelled 1 and 3 have degree 3. The node labelled 8 has degree 2. The node labelled 4 has degree 1. All the leaf nodes have degree 0. The degree of the tree is 3, because the maximum degree of any node is 3.

Levels: The level number appears on the right of the tree. The level of the root is 0, the level of the nodes labelled 2, 3, and 4 is 1, the level of the nodes labelled 5, 6, 7, and 8 is 2, and that of the nodes labelled 9 and 10 is 3.

Family relationships: The node labelled 1 is the **parent** of the nodes labelled 2, 3, and 4. The node labelled 3 is the parent of the nodes labelled 5, 6, and 7. The node labelled 4 is the parent of the node labelled 8, which in turn is the parent of the nodes labelled 9 and 10. The nodes labelled 2, 3, and 4 are **siblings** similar to the nodes labelled 9 and 10. Note that the node labelled 8 is not the sibling of the nodes labelled 5, 6, and 7.

Paths and path lengths: Paths exist from all parents to children. A unique path exists from the root to each leaf node as shown in following diagram. Since any sub-path is a path, all the paths are represented.

$1 \rightarrow 2$	Length: 1
$1 \rightarrow 3 \rightarrow 5$	Length: 2
$1 \rightarrow 3 \rightarrow 6$	Length: 2
$1 \rightarrow 3 \rightarrow 7$	Length: 2
$1 \rightarrow 4 \rightarrow 8 \rightarrow 9$	Length: 3
$1 \rightarrow 4 \rightarrow 8 \rightarrow 10$	Length: 3

Height and depth: The height of the tree is 3, the maximum level. The depth of the nodes labelle d 2, 3, and 4 is 1. The depth of the nodes labelled 5, 6, 7, and 8 is 2. The depth of the nodes labelled 9 and 10 is 3, which is the same as the height of the tree. The depth of the nodes on the lowest level is always the same as the height of the tree.

Orderings: The pre order, in order, and post order orderings of the nodes are given in the Following sequence:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow 10$ (preorder)
$2 \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 8 \rightarrow 10 \rightarrow 4$ (inorder)
$2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 3 \rightarrow 9 \rightarrow 10 \rightarrow 8 \rightarrow 4 \rightarrow 1$ (postorder)

5.3. BINARY TREES

The node at the top of the hierarchy is called as **root node**. A binary tree is a special form of a tree in which a parent node can have a maximum of **two child** nodes. The following diagram shows the general form of a binary tree:

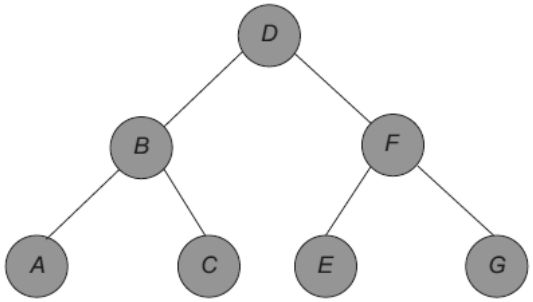


Figure 5.2. An Example of a BINARY TREE

Definition A binary tree

1. Is either an empty tree (or)
2. Consists of a node, called *root*, and two children, *left* and *right*, each of which is itself a binary tree.

5.3.1. Complete Binary Tree

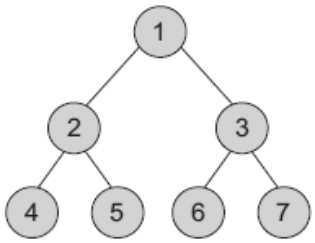
A binary tree of height *h* is complete if and only if one of the following holds good:

1. It is empty.
2. Its left subtree is complete of height *h* - 1 and its right subtree is completely full of height *h* - 2.
3. Its left subtree is completely full of height *h* - 1 and its right subtree is complete of height *h* - 1.

In other words, A binary tree is a *complete binary tree* if it contains the maximum possible number of nodes in all levels.

A binary tree is completely full if it is of height *h* and has $(2^h+1 - 1)$ nodes. In a full binary tree, each node has two children or no child at all. The total number of nodes in a full binary tree of height *h* is $2^h+1 - 1$ considering the root at level 0.

It can be calculated by adding the number of nodes of each level as in the following equation: $2^0 + 2^1 + 2^2 + ... + 2^h = 2^h+1 – 1$.



A Binary Tree has $(2^3+1) - 1 = 8 - 1 =$ 7 nodes.

Figure 5.3. A Binry Tree with seven nodes

A binary tree in sorted form holds great significance as it allows quick search, insertion and deletion operations.

5.3.2. Left and Right Skewed Binary Tree

If the right subtree is missing in every node of a tree, we call it a ***left skewed tree.*** If the left subtree is missing in every node of a tree, we call it as ***right skewed subtree.***

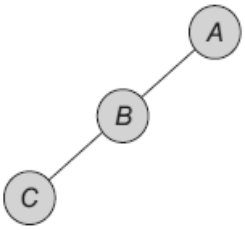


Figure 5.4 (a) A Left skewed tree

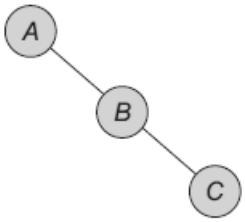


Figure 5.4 (b) A Right skewed tree

5.3.3. Strictly Binary Tree

If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called a *strictly binary tree*.

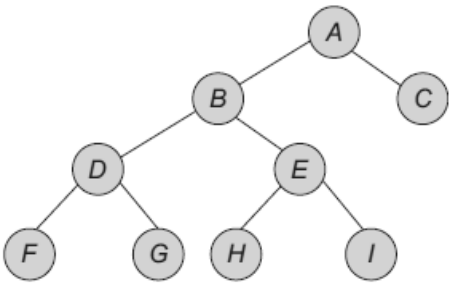


Figure 5.5. A Strictly Binary Tree

5.3.4. Extended Binary Tree

A binary tree *T* with each node having zero or two children is called an *extended binary tree*. The nodes with two children are called *internal nodes*, and those with zero children are called *external nodes*. Trees can be converted into extended trees by adding a node.

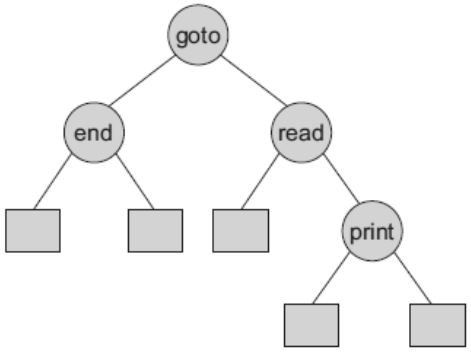


Figure 5.6. An Extended Binary Tree

5.3.5. Representation of Binary Tree

There are two standard representations of a binary tree given as follows:

1. Adjacency matrix (Sequential representation)
2. Adjacency list (Linked representation)

Array Representation of Binary Tree

One of the ways to represent a tree using an array is to store the nodes level-by-level, starting from the level 0 where the root is present. Such a representation requires sequential numbering of the nodes, starting with the nodes on level 0, then those on level 1, and so on.

A complete binary tree of height *h* has (2^h+1 - 1) nodes in it. The nodes can be stored in a one-dimensional array, *tree*, with the node numbered at the location *tree* (*i*). An array of size 2^h+1 - 1 is needed for the same.

Example 1:

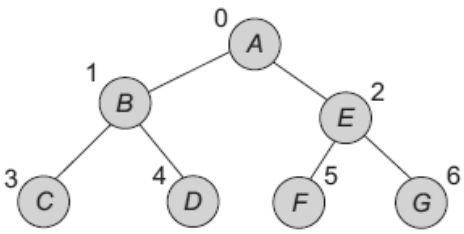


Figure 5.7. (a) A Complete Binary Tree **Figure 5.7.(b) Array Representation of 5.7 (a)**

Example 2:

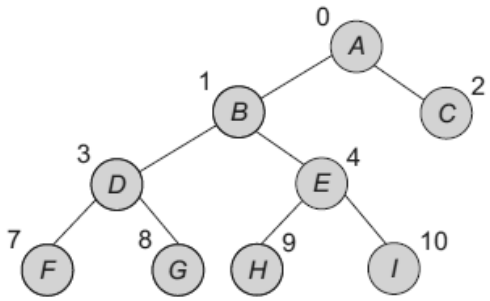


Figure 5.8. A Binary Tree with 11 Nodes

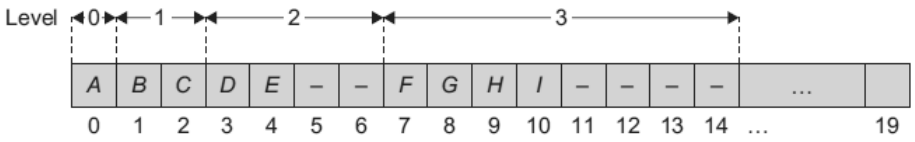


Figure 5.9. Array Representation of Figure 5.8

Example 3:

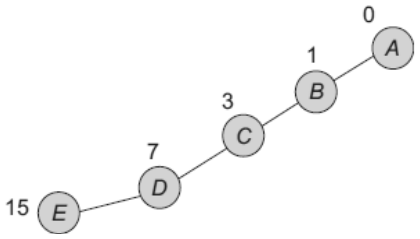


Figure 5.10. A Skewed Tree

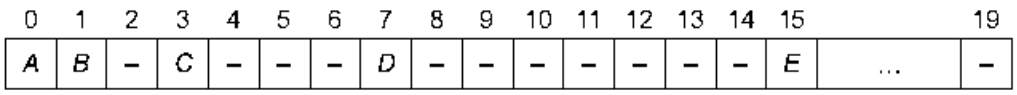


Figure 5.11. Array Representation of Figure 5.10

This representation of binary trees using an array seems to be the easiest. However, such a representation has certain drawbacks. In most of the representations, there will be a lot of unused space.

For complete binary trees, the representation seems to be good as no space in an array is wasted between the nodes. Other than full binary trees, majority of the array entries may be empty. It allows only static representation. The array size cannot be changed during the execution.

These problems can be overcome by the use of linked representation.

Linked Representation of Binary Tree

Binary tree has a natural implementation in a linked storage. In a linked organization, we wish that all the nodes should be allocated dynamically.

Hence, we need each node with data and link fields. Each node of a binary tree has both a left and a right subtree. Each node will have three fields—Lchild, Data, and Rchild.

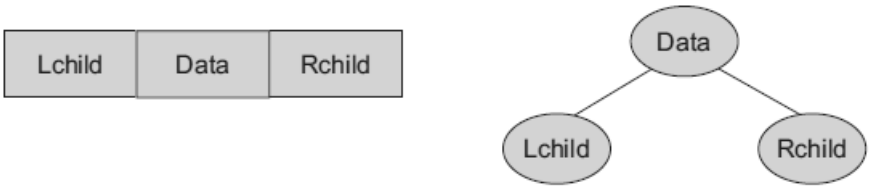


Figure 5.12. A Node with three fields

In this node structure, Lchild and Rchild are the two link fields to store the addresses of left child and right child of a node; data is the information content of the node.

With this representation, if we know the address of the root node, then using it, any other node can be accessed.

Each node of a binary tree (as the root of some subtree) has both left and right subtrees, which can be accessed through pointers as follows.

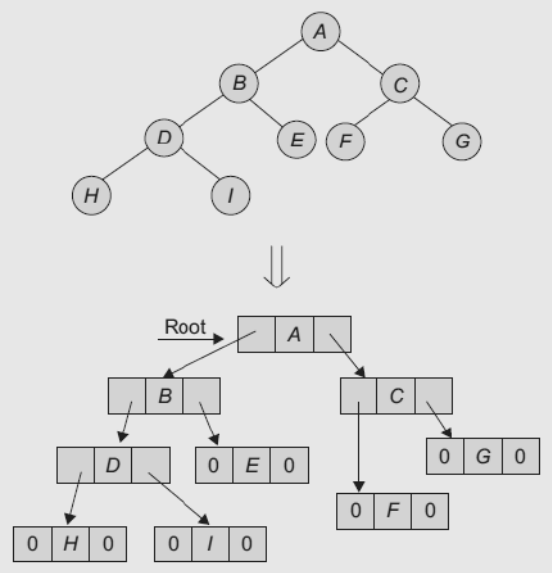


Figure 5.13. A Linked Representation of a Binary Tree

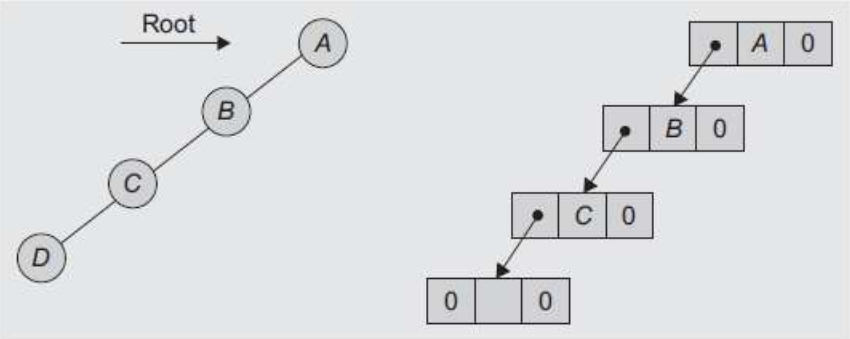


Figure 5.14. A Linked Representation of a Skewed Tree

Advantages

The merits of representing binary trees through liked representations are as follows:

1. The drawbacks of the sequential representation are overcome in this representation. We may or may not know the tree depth in advance. In addition, for unbalanced trees, the memory is not wasted.
2. Insertion and deletion operations are more efficient in this representation.
3. It is useful for dynamic data.

Disadvantages

The demerits of representing binary trees through linked representation are as follows:

1. In this representation, there is no direct access to any node. It has to be traversed from the root to reach to a particular node.
2. As compared to sequential representation, the memory needed per node is more. This is due to two link fields (left child and right child for binary trees) in the node.
3. The programming languages not supporting dynamic memory management would not be useful for this representation.

5.3.6. Binary Tree Abstract Data Types

Let us now define Binary Tree as an abstract data type (ADT), which includes a list of operations that process it.

```
ADT btree
1. Declare create()→btree
2. makebtree(btree, element,btree) →btree
3. isEmpty(btree) →boolean
4. leftchild(btree) →btree
5. rightchild(btree) →btree
6. data(btree) →element
7. for all l,r € btree, e € element, Let
8. isEmpty(create) = true
9. isEmpty(makebtree(l,e,r)) = false
10. leftchild(create()) = error
11. rightchild(create()) = error
12. leftchild(makebtree(l,e,r)) = l
13. rightchild(makebtree(l,e,r)) = r
14. data(makebtree(l,e,r)) = e
15. end
end btree
```

The following program code states the class definition and operations that process the tree ADT.

```
class TreeNode
{
public:
char Data;
TreeNode *Lchild;
TreeNode *Rchild;
};
class BinaryTree
{
private:
TreeNode *Root;
public:
BinaryTree(){Root = Null};
// constructor creates an empty tree
TreeNode * GetNode();
void InsertNode(TreeNode*);
void DeleteNode( TreeNode*);
};
```

The six functions with their domains and ranges are declared in lines 1 through 6. Lines 7 through 14 are the set of axioms that describe how the functions are related.

The create() operation creates an empty binary tree; the isEmpty()operation checks whether btree is empty or not and returns the Boolean value true or false, respectively; leftchild(btree) and rightchild(btree) return the left and right sub trees, respectively; data(btree) returns the data element.

5.3.7. Operations on binary tree

The basic operations on a binary tree can be as listed as follows:

1. Creation—Creating an empty binary tree to which the 'root' points
2. Traversal—Visiting all the nodes in a binary tree
3. Deletion—Deleting a node from a non-empty binary tree
4. Insertion—Inserting a node into an existing (may be empty) binary tree
5. Merge—Merging two binary trees
6. Copy—Copying a binary tree
7. Compare—Comparing two binary trees
8. Finding a replica or mirror of a binary tree

5.3.8. Conversion of General Tree to Binary Tree

Binary trees are the trees where the maximum degree of any node is two. Any general tree can be represented as a binary tree using the following algorithm:

1. All nodes of a general tree will be the nodes of a binary tree.
2. The root T of a general tree is the root of a binary tree.
3. To obtain a binary tree, we use a relationship between the nodes that can have the following two characteristics:
   (a) The first or the leftmost child–parent relationship
   (b) Node-next right sibling relationship

Use the following steps to obtain binary tree from general tree:

1. Connect (insert an arrow from) each node to its right sibling (if one exists).
2. Disconnect (remove arrows from) each node from (to) all but the leftmost child.

Example: Convert the following general tree into its corresponding binary tree.

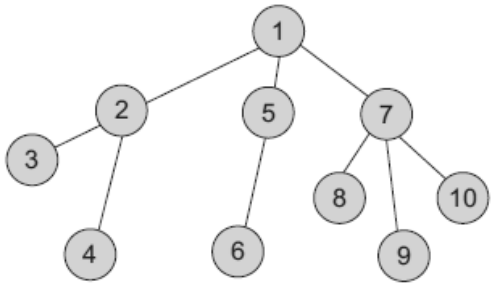


Figure 5.15. General Tree

Solution: In this tree, the leftmost child of 2 is 3 and the next right child of 2 is 4. The binary tree corresponding to the tree is obtained by connecting together all siblings of each node in the first figure and deleting all links from a node to its children except for the link to its leftmost child. The binary tree obtained is shown below:

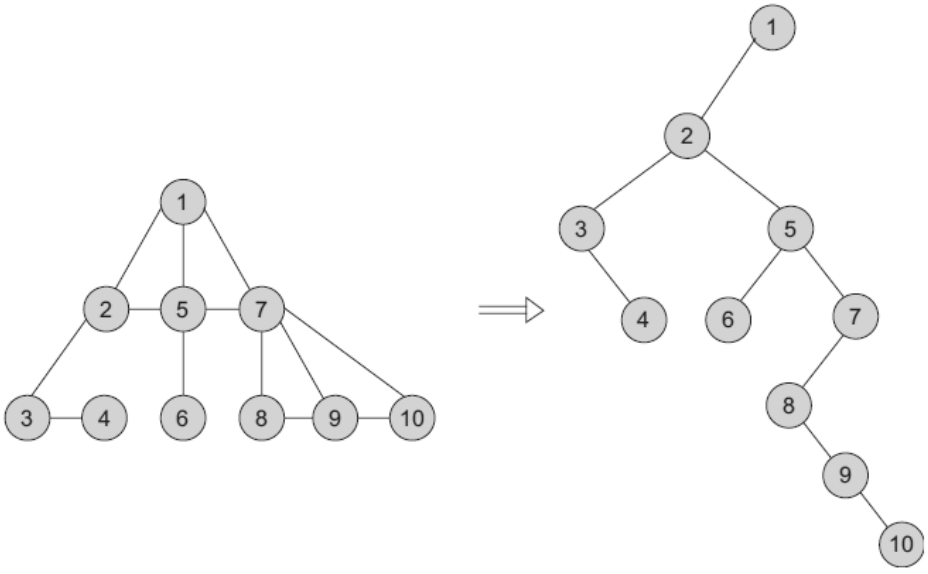


Figure 5.16. General Tree Converted into Binary Tree

5.3.9. Binary Tree Traversal

Traversal is a frequently used operation. Traversal of a tree means *stepping through the nodes of a tree* by means of the connections between parents and children, which is also called *walking the tree*, and the action is *a walk of the tree*. Traversal means visiting every node of a binary tree.

There are many operations that are often performed on a tree such as search a node, print some information, insert a node, delete a node, and so on. All such operations need the traversal through a tree.

This operation is used to visit each node (exactly once). A full traversal of a tree visits nodes of a tree in a certain linear order. This linear order could be familiar and useful.

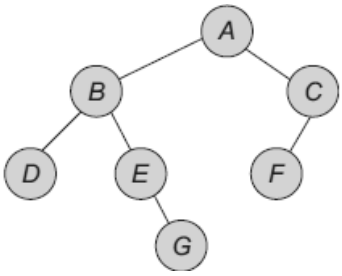
Let L, D, and R stand for moving left, data, and moving right, respectively, when at a node, then there are three possible combinations—LDR, LRD, DLR. These are called as I***norder, Postorder,*** and ***Preorder*** traversals because there is a natural correspondence between these traversals producing the infix, postfix, and prefix forms of an arithmetic expression, respectively.

Preorder Traversal

In this traversal, the root is visited first followed by the left sub tree in preorder and then the right sub tree in pre order. The tree characteristics lead to naturally implement the tree traversals recursively. It can be defined in the following steps:

Preorder (DLR) Algorithm

1. Visit the root node, say D.
2. Traverse the left sub tree of the node in preorder.
3. Traverse the right sub tree of the node in preorder.

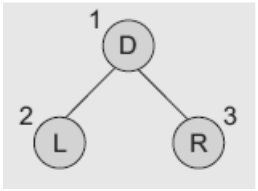


A B D E G C F

5.17. Binary tree and its preorder expression

The Preorder traversal says, 'visit a node, traverse left, and continue moving. When you cannot continue, move right and begin again or move back, until you can move right and stop'.

```
void BinaryTree :: Preorder(TreeNode*)
{
if(Root != Null)
{
cout << Root->Data;
Preorder(Root->Lchild);
Preorder(Root->Rchild);
}
}
```



Inorder Traversal

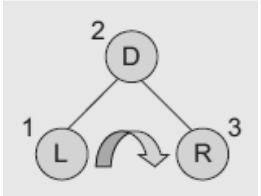
In this traversal, the left subtree is visited first in inorder followed by the root and then the right sub tree in inorder. This can be defined as the following:

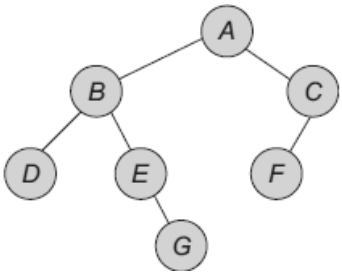
Inorder (LDR) Algorithm

1. Traverse the left sub tree of the root node in inorder.
2. Visit the root node.
3. Traverse the right sub tree of the root node in inorder.

The Inorder() function simply calls for moving down the tree towards the left until it can no longer proceed. So next, we visit the node, move one node to the right, and continue again. If we cannot move, move one node to the right and continue again. If we cannot move to the right, go back one more node and then continue. The inorder traversal is also called as symmetric traversal. This traversal can be written as a recursive function as follows:

```
void BinaryTree :: Inorder(TreeNode*)
{
if(Root != Null)
{
Inorder(RootÆLchild);
cout << RootÆData;
Inorder(RootÆRchild);
}
}
```





D B E G A F C

5.18. Binary tree and its Inorder expression

Postorder Traversal

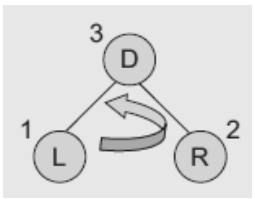
In this traversal, the left subtree is visited first in postorder followed by the right sub tree in post order and then the root. This is defined as the following:

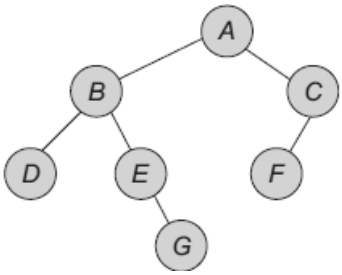
Postorder (LRD) Algorithm

1. Traverse the root's left child (subtree) of the root node in postorder.
2. Traverse the root's right child (subtree) of the root node in postorder.
3. Visit the root node.

The Postorder traversal says, "traverse left and continue again. When you cannot continue, move right and begin again or move back until you can move right and visit the node."

```
void BinaryTree :: Postorder(TreeNode*)
{
if(Root != Null)
{
Postorder(Root->Lchild);
Postorder(Root->Rchild);
cout << Root->Data;
}
}
```





D G E B F C A

5.19. Binary tree and its Postorder expression

5.4. BINARY SEARCH TREES

The binary search tree (BST) is one of the fundamental data structures extensively used for searching the target in a set of ordered data. BSTs are widely used for retrieving data from databases, look-up tables, and storage dictionaries.

It is the most efficient search technique having a time complexity that is logarithmic to the size of the set. There are two cases with respect to BST construction.

The first case is a set of keys and the probabilities with which they are searched, which is known in advance.

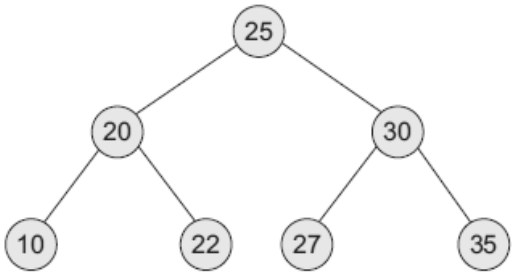
The second is when knowledge about the keys is not available in advance and the keys occur dynamically. These two cases lead to the following two kinds of search trees:

1. *Static BST*—is one that is not allowed to update its structure once it is constructed. In other words, the static BST is an offline algorithm, which is presumably aware of the access sequence beforehand.
2. *Dynamic BST*—is one that changes during the access sequence. We assume that the dynamic BST is an online algorithm, which does not have prior information about the sequence.

A Binary Search Tree is the only data structure left that not only has an efficient searching algorithm but also efficient insertion and deletion algorithms.

A BST can be defined as a key-based tree with the following properties:

1. Every element has a key, and no two elements have the same key (i.e., keys are unique).
2. The keys (if any) in the left sub tree are smaller than the key in the root.
3. The keys (if any) in the right sub tree are greater than the key in the root.
4. Each sub tree in itself is a BST.



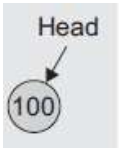
5.20. Sample binary search tree

Building Binary Search Tree

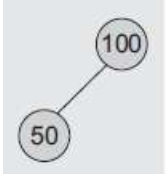
Build a BST from the following set of elements—100, 50, 200, 300, 20, 150, 70, 180, 120, 30—and traverse the tree built in inorder, postorder, and preorder.

Solution The BST is constructed through the following steps:

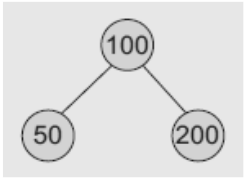
Step 1: Initially, Root = Null. Now let us insert 100.



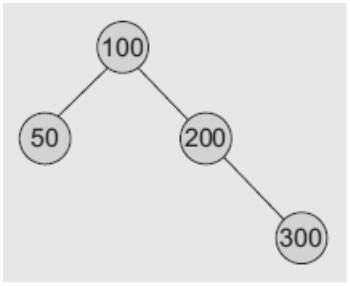
Step 2: Insert 50. As it is less than the root, that is, 100, and its left child is Null, we insert it as a left child of the root.



Step 3: Insert 200. As it is greater than the root, that is, 100, and its right child is Null, we insert it as a right child of the root.

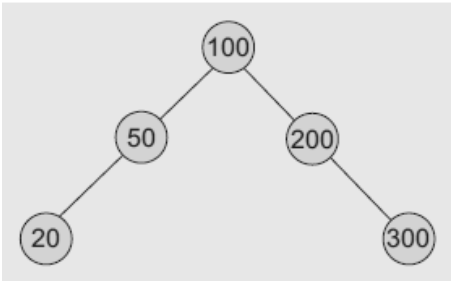


Step 4: Insert 300. As it is greater than the root, that is, 100, we move right to 200. It is greater than 200, and its right child is Null, so we insert it as a right child of 200.

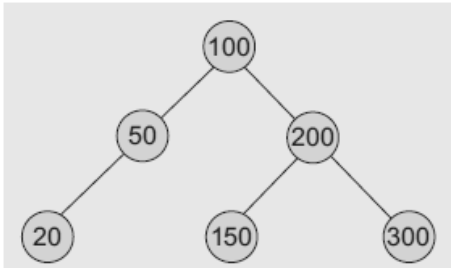


Similarly, we insert the other nodes.

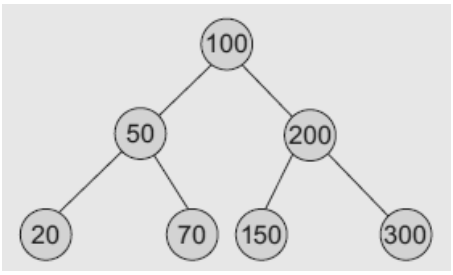
Step 5: Insert 20.



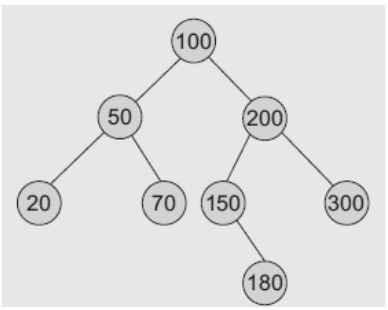
Step 6: Insert 150.



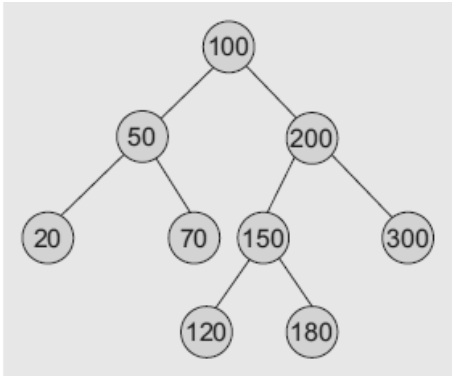
Step 7: Insert 70.



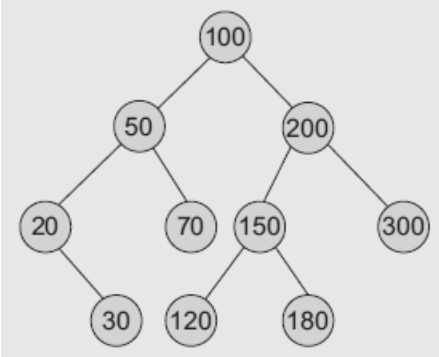
Step 8: Insert 180.



Step 9: Insert 120.



Step 10: Insert 30.



Traverse the built tree in inorder, postorder, and preorder and display the sequence of numbers.

Preorder: 100 50 20 30 70 200 150 120 180 300

Inorder: 20 30 50 70 100 120 150 180 200 300

Postorder: 30 20 70 50 120 180 150 300 200 100.

Binary Tree and Binary search Tree

A BST is a special case of the binary tree. The comparison of both yields the following points:

1. Both of them are trees with degree two, that is, each node has utmost two children. This makes the implementation of both easy.
2. The BST is a binary tree with the property that the value in a node is greater than any value in a node's left subtree and less than any value in a node's right subtree.
3. The BST guarantees fast search time provided the tree is relatively balanced, Whereas for a binary tree, the search is relatively slow.

Given a binary tree with no duplicates, we can construct a BST from a binary tree. The process is easy; one can traverse the binary tree and construct a BST for it by inserting each node in an initially empty BST.

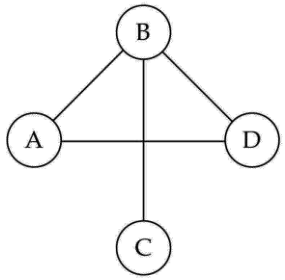
GRAPH

5.5. INTRODUCTION

Non-linear data structures are used to represent the data containing a network or hierarchical relationship among the elements.

Graphs are one of the most important nonlinear data structures. In non-linear data structures, every data element may have more than one predecessor as well as successor. Elements do not form any particular linear sequence.

Definition: A graph *G* as simply a way of encoding pairwise relationships among a set of objects. Thus, *G* consists of a pair of sets *(V, E)*—a collection *V* of *nodes* and a collection *E* of *edges*, each of which "joins" two of the nodes. We thus represent an edge $e \in E$ as a two-element subset of *V*: $e = \{u, v\}$ for some $u, v \in V$, where we call *u* and *v* the *ends* of *e*.



V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

Figure: 5.21. Simple Graph

5.5.1. Basic Terminologies of Graph

Adjacent Nodes

If an edge $e \in E$ is associated with a pair of nodes (a, b) where $a, b \in V$, then it is said that the edge *e* joins or connects the nodes *a* and *b*. Any two nodes that are connected with an edge are called as ***adjacent nodes***.

(5 is adjacent to 7 and 7 is adjacent from 5)

Figure 5.22. Adjacent Nodes

Directed and Undirected Graphs

In a graph *G* (*V*, *E*), an edge that is directed from one node to another is called a d***irected edge***, whereas an edge that has the no specific direction is called an ***undirected edge***. A graph where every edge is directed is called as a ***directed graph*** **or** ***diagraph***. A graph where every edge is undirected is called as an ***undirected graph***.

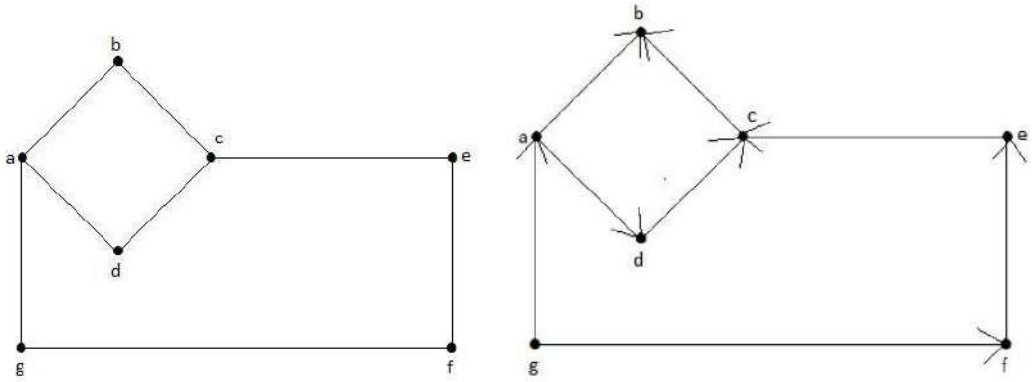


Figure 5.23 (a) Undirected Graph and (b) Directed Graph

Mixed Graph

If some of edges are directed and some are undirected in a graph, the graph is called as a *mixed graph*.

Initial and Terminal Node

Let (*V*, *E*) be a graph and let $e \in E$ be a directed edge associated with the ordered pair of nodes (*a*, *b*). Then, the edge *e* is said to be initiating or originating in the node *a* and terminating or ending in the node *b*. The nodes *a* and *b* are also called the ***initial*** **and** ***terminal nodes*** respectively, of the edge *e*.

Loop or Sling

An edge of a graph that joins a node to itself is called a ***loop*** **(sling).**

Incident

An edge $e \in E$ that joins the nodes *a* and *b*, be it directed or undirected, is said to be ***incident*** to the nodes *a* and *b*, respectively.

Parallel Edges and Multigraph

The following graph has only one edge between any pair of nodes. In the directed edges, the two possible edges between the pair of nodes that are opposite in direction are considered distinct. In some directed as well as undirected graphs, there may exist more than one edge **incident** to the same pair of nodes, say *a* and *b*.

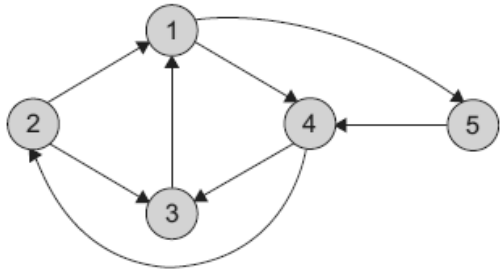


Figure 5.24. Simple Graph

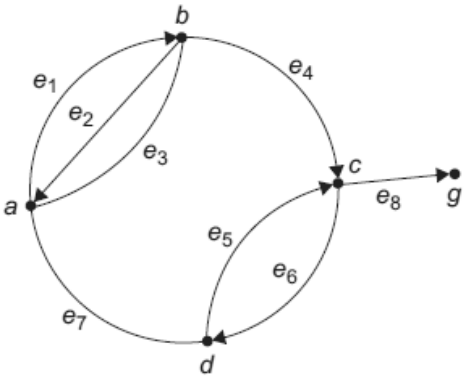


Figure 5.25. Multi Graph

In figure 5.25, the edges *e*1, *e*2, and *e*3 are incident to vertices *a* and *b*. Such edges are called as *parallel edges*. Here, *e*1, *e*2, and *e*3 are three parallel edges.

In addition, *e*5 and *e*6 are two parallel edges. Any graph that contains parallel edges is called a ***multigraph***. On the other hand, a graph that has no parallel edges is called a ***simple graph***.

Weighted Graph

A graph where weights are assigned to every edge is called a ***weighted graph***. Weights can also be assigned to vertices.

A graph of areas and streets of a city may be assigned weights according to its traffic density.

A graph of areas and connecting roads may be assigned weights such that the distance between the cities is assigned to edges and area population is assigned to vertices.

Null Graph and Isolated Vertex

In a graph, a node that is not adjacent to any other node is called an ***isolated node***. A graph containing only isolated nodes is called a ***null graph.*** Hence, the set of edges is an empty set in a null graph.

Let V = set of students, E = {there exists an edge incident to two students if they share books}. Let $V = \{a, b, c\}$. If no two students among a, b, and c shares books, then the graph $G = \{V, E\}$ is represented as shown below.

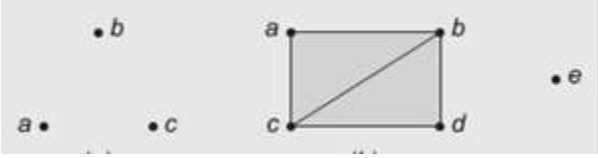


Figure 5.26. Null Graphs and Isolated Vertex

Here, G is a null graph, and a, b, and c are isolated vertices. $V = \{a, b, c, d, e\}$ and $E = \{(a, b), (a, c), (b, c), (c, d), (b, d)\}$, and e is an isolated vertex.

Degree of Vertex

In a directed graph, for any node V, the number of edges that have V as its initial node is called the *outdegree* of the node V.

In other words, the number of edges incident from a node is its ***outdegree*** **(***outgoing degree*), and the number of edges incident to it is an ***indegree*** (*incoming degree*). The sum of indegree and outdegree is the total degree of a node (*vertex*).

In an undirected graph, the total degree or degree of a node is the number of edges incident to the node. The isolated vertex degree is zero.

Connectivity

A graph is said to be connected if and only if there exists a path between every pair of vertices. The graph $G = (V, E)$ drawn (in Figure 5.27) with $V = \{a, b, c, d, e\}$ is a disconnected graph. It contains two connected components. A connected graph has a single connected component. The graph shown in Figure 5.28 is a connected graph.

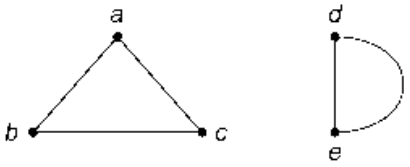


Figure 5.27. Disconnected Graph

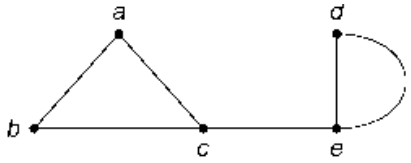


Figure 5.28. Connected graph

Acyclic Graph

A simple graph that does not have any cycles is called *acyclic graph*. Such graphs do not have any loops.

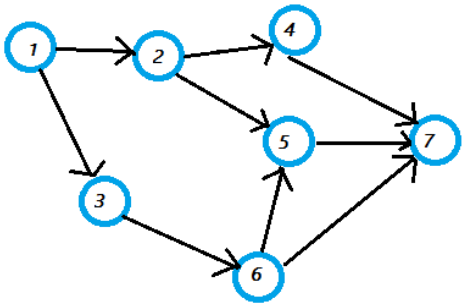


Figure 5.29. Acyclic Graph

5.5.2. Applications of Graph Theory

Graph theory has its applications in diverse fields of engineering

- **Electrical Engineering** − The concepts of graph theory is used extensively in designing circuit connections. The types or organization of connections are named as topologies. Some examples for topologies are star, bridge, series, and parallel topologies.
- **Computer Science** − Graph theory is used for the study of algorithms. For example,
    - Kruskal's Algorithm
    - Prim's Algorithm
    - Dijkstra's Algorithm
- **Computer Network** − The relationships among interconnected computers in the network follows the principles of graph theory.
- **Science** − The molecular structure and chemical structure of a substance, the DNA structure of an organism, etc., are represented by graphs.
- **Linguistics** − The parsing tree of a language and grammar of a language uses graphs.

5.5.3. Graph Abstract Data Type

We can denote the graph as $G = (V, E)$. Let us define the graph ADT. We need to specify both sets of vertices and edges. Basic operations include creating a graph, inserting and deleting a vertex, inserting and deleting an edge, traversing a graph, and a few others.

A graph is a set of vertices and edges $\{V, E\}$ and can be declared as follows:

graph

create() –> Graph
insert_vertex(Graph, v) –> Graph
delete_vertex(Graph, v) –> Graph
insert_edge(Graph, u, v) –> Graph
delete_edge(Graph, u, v) –> Graph
is_empty(Graph) –> Boolean;

end graph

These are the **primitive operations** that are needed for storing and processing a graph.

Create

The create operation provides the appropriate framework for the processing of graphs. The create() function is used to create an empty graph. An empty graph has both V and E as null sets.

Insert Vertex

The insert vertex operation inserts a new vertex into a graph and returns the modified graph.

When the vertex is added, it is isolated as it is not connected to any of the vertices in the graph through an edge. If the added vertex is related with one (or more) vertices in the graph, then the respective edge(s) are to be inserted.

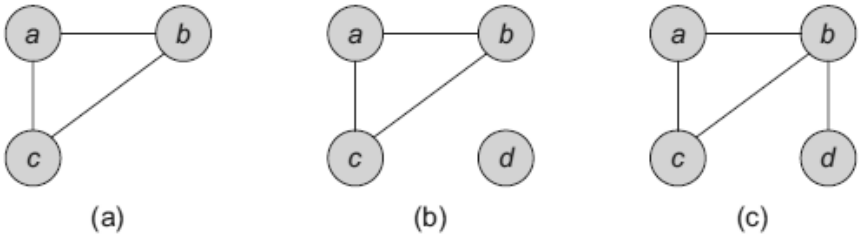


Figure 5.30. Inserting Vertex and Edge into a Graph

The figure 5.30 shows a graph G (V, E), where $V = \{a, b, c\}$ and $E = \{(a, b), (a, c), (b, c)\}$, and the resultant graph after inserting the node d. The resultant graph G is shown in figure 5.30 (b). It shows the inserted vertex with resultant $V = \{a, b, c, d\}$. We can show the adjacency relation with other vertices by adding the edge as in figure 5.30 (c). So now, E would be $E = \{(a, b), (a, c), (b, c), (b, d)\}$ as shown below.

Delete Vertex

The delete vertex operation deletes a vertex and all the incident edges on that vertex and returns the modified graph.

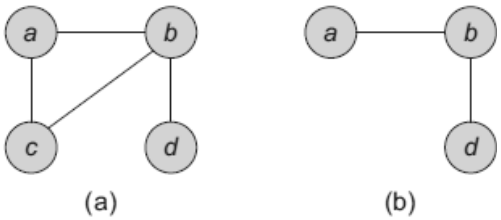


Figure 5.31. Deleting Vertex and Edge from a Graph

Consider the Figure 5.31(a), it shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$, and the resultant graph after deleting the node c is shown in figure 5.31 (b) with $V = \{a, b, d\}$ and $E = \{(a, b), (b, d)\}$.

Insert Edge

The insert edge operation adds an edge incident between two vertices. In an undirected graph, for adding an edge, the two vertices u and v are to be specified, and for a directed graph along with vertices, the start vertex and the end vertex should be known.

The following figure 5.32(a) shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$ and the resultant graph after inserting the edge (c, d) with $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d), (c, d)\}$is shown figure 5.32(b).

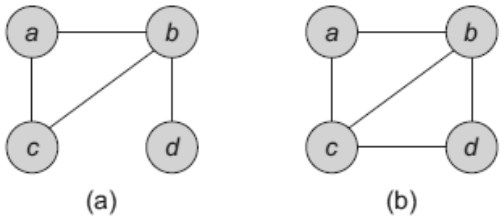


Figure 5.32. Inserting an Edge to a Graph

Delete Edge

The delete edge operation removes one edge from the graph. Let the graph G be $G(V, E)$. Now, deleting the edge (u, v) from G deletes the edge incident between vertices u and v and keeps the incident vertices u, v.

The following figure 5.33(a) shows a graph $G(V, E)$, where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$. The resultant graph after deleting the edge (b, d) is shown in figure 5.33(b) with $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c)\}$.

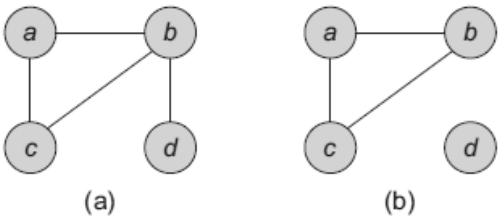


Figure 5.33. Deleting an Edge from Graph

Is_empty

The is_empty() operation checks whether the graph is empty and returns true if empty else returns false. An empty graph is one where the set V is a null set.

5.5.4. Representation of Graphs

We need to store two sets *V* and *E* to represent a graph. Here *V* is a set of vertices and *E* is a set of incident edges. These two sets basically represent the vertices and adjacency relationship among them.

There are two standard representations of a graph given as follows:

1. Adjacency matrix (Sequential representation) and
2. Adjacency list (Linked representation)

Adjacency Matrix

Adjacency matrix is a square, two-dimensional array with one row and one column for each vertex in the graph. An entry in row *i* and column *j* is 1 if there is an edge incident between vertex *i* and vertex *j*, and is 0 otherwise.

For a graph $G = (V, E)$, suppose $V = \{1, 2, \ldots, n\}$. The adjacency matrix for *G* is a two dimensional *n* X *n* Boolean matrix ***A*** and can be represented as

$$A[i][j] = \{1 \text{ if there exists an edge } <i, j> \\ 0 \text{ if edge } <i, j> \text{ does not exist}\}$$

The adjacency matrix ***A*** has a natural implementation as in the following:

- ***A[i][j]*** is 1 (or true) if and only if vertex *i* is adjacent to vertex *j*.
- If the graph is undirected, then ***A[i][j]*** = ***A[j][i]*** = 1.
- If the graph is directed, we interpret 1 stored at ***A[i][j]***, indicating that the edge from *i* to *j* exists and not indicating whether or not the edge from *j* to *i* exists in the graph.

The graphs *G*1, *G*2, and *G*3 are represented using the adjacency matrix , among which *G*2 is a directed graph.

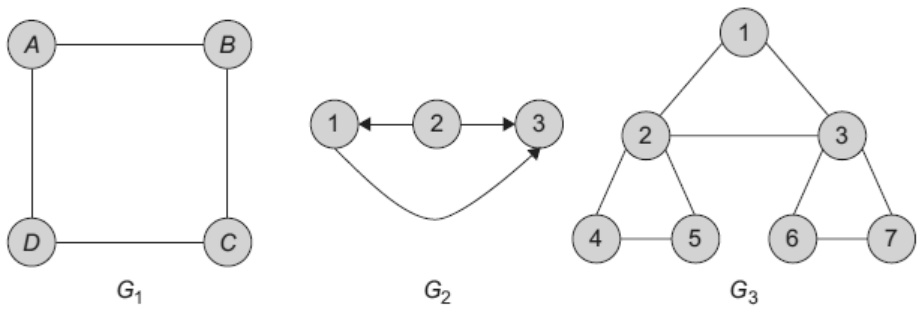


Figure 5.34. Graphs *G*1, *G*2 (Digraph), and *G*3

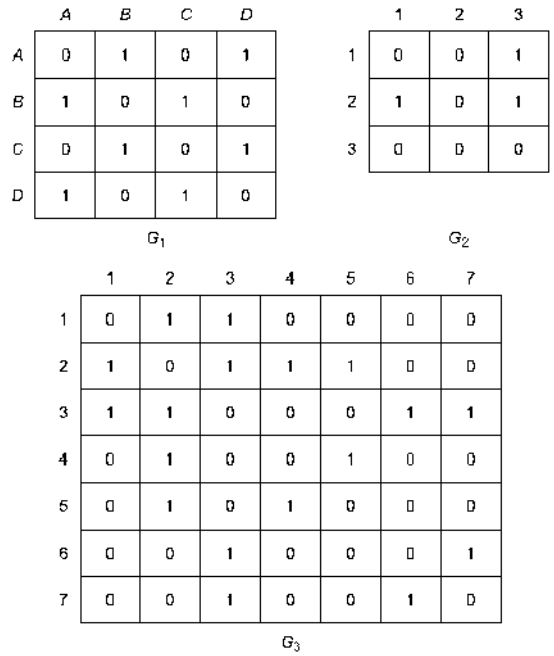


Figure 5.35. Adjacency Matrices for *G*1, *G*2, and *G*3

Adjacency List

In this representation, the *n* rows of the adjacency list are represented as *n*-linked lists, one list per vertex of the graph. The adjacency list for a vertex *i* is a list of all vertices adjacent to it. One way of achieving this is to go for an array of pointers, one per vertex.

For example, we can represent the graph *G* by an array Head, where Head[i] is a pointer to the adjacency list of vertex *i*.

For list, each node of the list has at least **two fields: vertex and link**. The vertex field contains **the vertex id**, and the link field stores **a pointer to the next node** that stores another vertex adjacent to *i*.

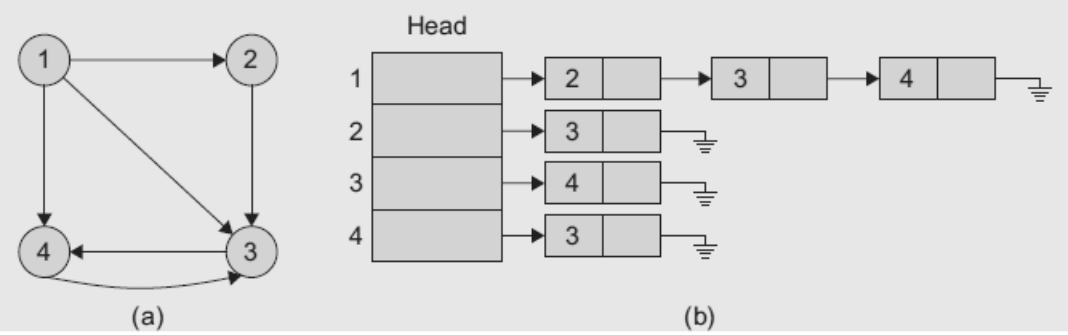


Figure 5.36. Adjacency List Representation of a Graph

5.5.5. Graph Traversal

To solve many problems modeled with graphs, we need to visit all the vertices and edges in a systematic fashion called *graph traversal*. We shall study **two types**

1. Depth-first traversal (DFS)
2. Breadth-first traversal (BFS)

Traversal of a graph is commonly used to search a vertex or an edge through the graph; hence, it is also called a *search technique*. Consequently, depth-first and breadth-first traversals are popularly known as *depth-first search* (DFS) and *breadth-first search* (BFS), respectively.

Depth-first Search

In DFS, as the name indicates, from the currently visited vertex in the graph, we keep searching deeper whenever possible.

Depth-first search works by selecting one vertex, say v of *G* as a start vertex; v is marked as visited. Then, each unvisited vertex adjacent to v is searched using the DFS recursively. Once all the vertices that can be reached from v have been visited, the search for v is complete. If some vertices remain unvisited, we select an unvisited vertex as a new start vertex and then repeat the process until all the vertices of *G* are marked as visited.

The recursive algorithm for DFS.

```
1. for v = 1 to n do
        visited[v] = 0 {unvisited}
2.      i = 1 {Let us start at vertex 1)
3.      DepthFirstSearch(i)
        begin
              visited[i] = 1
              for each vertex j adjacent
        to i do
              if(visited[j] = 0) then
              DepthFirstSearch(j)
        end
4. Stop
```

Consider the graph G and its adjacency list in following figure 5.37.

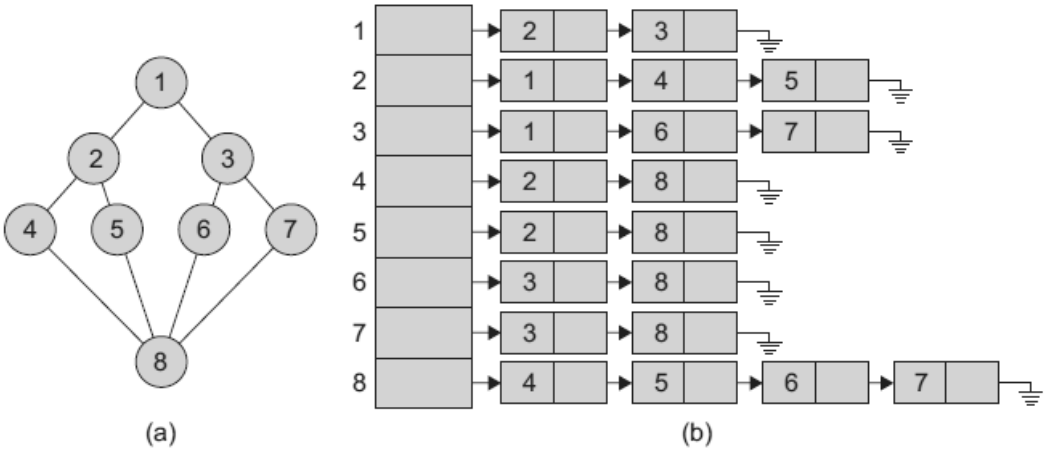


Figure 5.37. DFS - Adjacency List Representation of a Graph

Let us initiate a traversal using Adjacency list representation of graph *G* from the vertex 1 and the order of traversal will be 1, 2, 4, 8, 5, 6, 3, 7.

Breadth-first Search

The BFS differs from DFS in a way that all the unvisited vertices adjacent to i are visited after visiting the start vertex i and marking it visited. The approach is called 'breadth-first' because from the vertex i that we visit, we search as broadly as possible by next visiting all the vertices adjacent to i.

This search algorithm uses ***a queue*** to store the vertices of each level of the graph as and when they are visited. These vertices are then taken out from the queue in sequence, that is, first in first out (FIFO), and their adjacent vertices are visited until all the vertices have been visited. The algorithm terminates when the queue is empty.

The working of the BFS is given in following. The algorithm initializes the Boolean array visited[] to 0 (false), that is, marks each vertex as unvisited.

```
Breadth-first search (vertex j)
1. Let us start search at vertex j
2. Mark all vertices as unvisited
        for i = 1 to n do
        visited[i] = 0
3. Mark j as visited
        visited[j] = 1
4. Add j in queue
5. while not queue empty do
        begin
        i = delete from queue
        for all vertices j adjacent to i
        do
                begin
                if(not visited[j] = 1)
                Add j in queue
                visited[j] = 1
                end
        end
6. Stop
```

Let us consider following graph for BFS. Let us traverse the graph using a non- recursive algorithm that uses a queue. Let 1 be the start vertex. Initially, the queue is empty, and the initial set of visited vertices, V = Empty Set.

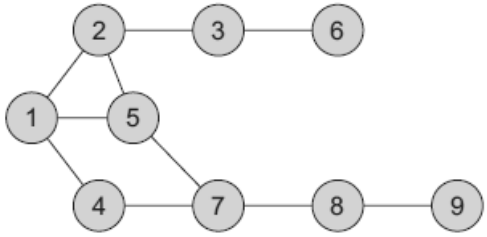
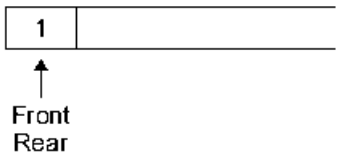
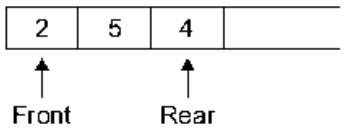


Figure 5.38. Example Graph for BFS

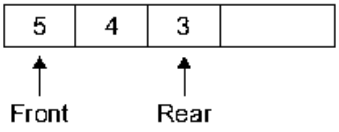
Step 1: Add 1 to the queue. Mark 1 as visited. V = {1}.



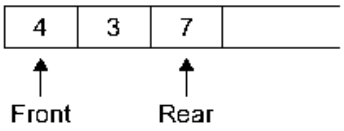
Step 2: As the queue is not empty, vertex = delete() from queue, and we get 1. Add all the un-visited adjacent vertices of 1 to the queue. In addition, mark them as visited. Now, V = {1, 2, 5, 4}.



Step 3: As the queue is not empty, vertex = delete() and we get 2. Add all the adjacent, un-visited vertices of 2 to the queue and mark them as visited. Now V = {1, 2, 5, 4, 3}.



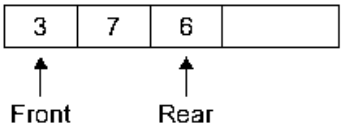
Step 4: As the queue is not empty, vertex = delete () from queue, and we get 5. Now, add all the adjacent, un-visited vertices adjacent to 5 to the queue and mark them as visited. Now, $V = \{1, 2, 5, 4, 3, 7\}$.



Step 5: As the queue is not empty, vertex = delete() from queue, and we get 4. Now, add all the adjacent, not visited vertices adjacent to 4 to the queue. The vertices 1 and 7 are adjacent to 4 and hence are already visited. Now the next element we get from the queue is 3.

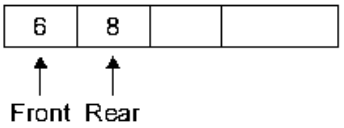
Now, we add all the un-visited vertices adjacent to 3 to the queue, making

$V = \{1, 2, 5, 4, 3, 7, 6\}$.



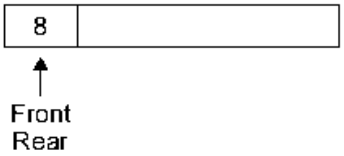
Step 6: As the queue is not empty, vertex = delete() and we get 7. Add all the adjacent, un- visited vertices of 7 to the queue and mark them as visited.

Now, $V = \{1, 2, 5, 4, 3, 7, 6, 8\}$.

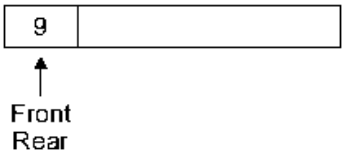


Step 7: As the queue is not empty, vertex = delete(), and we get 6. Then, add all the un-visited adjacent vertices of 6 to the queue and mark them as visited.

Now $V = \{1, 2, 5, 4, 3, 7, 6, 8\}$.



Step 8: As queue is not empty, vertex = delete() and we get 8. Add its adjacent un-visited vertices to the queue and mark them as visited. *V* = {1, 2, 5, 4, 3, 7, 6, 8, 9}.



Step 9: As the queue is not empty, vertex = delete() = 9. Here, note that no adjacent vertices of 9 are un-visited.

Step 10: As the queue is empty, we stop.

The sequence in which the vertices are visited by the BFS is 1, 2, 5, 4, 3, 7, 6, 8, 9

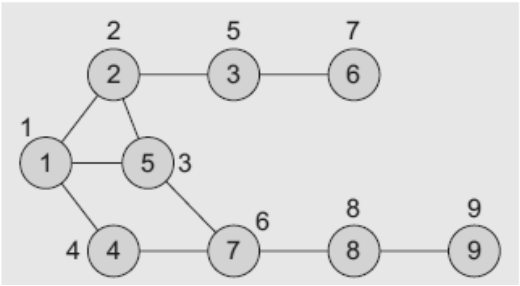


Figure 5.39. Breadth-first search sequence for the graph

5.5.6. Dijkstra's Shortest Path Algorithm

A *weighted graph* is a graph where the values are assigned to the edges and the length of a path is the sum of the weight of the edges in the path. We let w(i, j) denote the weight of edge (i, j).

In a weighted graph, we often need to find the shortest path. The shortest path between two given vertices is the path having minimum length. This problem can be solved by one of the greedy algorithms, by Edger W. Dijkstra, often called as *Dijkstra's algorithm*.

Consider a directed graph G = {A, B}. Each edge has a non-negative length. One of the nodes is the source vertex. Suppose we are to determine the shortest path from a to the destination vertex z.

Let us use two sets of vertices, visited and unvisited. Let V denote the set of visited vertices that contains the vertices that have already been chosen and the minimal distance from the source is already known for every vertex in V.

The set U contains all other vertices whose minimal distance from the source is not yet known.

Let an array Dist hold the length of the shortest distance and the array Path hold the shortest path between the source and each of the vertices. At each step, Dist[i] shows the shortest distance between a and i, and Path[i] shows the shortest path between a and i.

Initially, **a** is the only vertex in V. At each step we add to V, another vertex, for which the shortest path from **a** has been determined. The array Dist[] is initialized by setting Dist[i] to the weight of the edge from a to i if it exists and to ∞ if it does not.

To determine which vertex to add to V at each step, we apply the criteria of choosing the vertex j with the smallest distance recorded in Dist such that j is not the visited one.

When we add j to V (set of visited vertices), we must update the entries of Dist by checking, for each vertex k that is not in V, whether a path through j and then directly to k is shorter than the previously recorded distance of k.

That is, we replace Dist[k] by Dist[j] + weight of the edge from j to k if the value of the latter quantity is lesser. Here, j is the currently selected vertex. Let k be a vertex whose distance is updated.

If the distance is updated, then the path is also updated. Then, path[k] becomes the path of j followed by k.

In brief,
if Dist[k] > (Dist[j] + weight<j,k>)
then
Dist[k] = Dist[j] + w <j,k>
and
Path[k] = Path[j] U{k}

The following algorithm for computing the shortest path from the source vertex to the destination Vertex.

1. Let G = (A, B) where A = set of vertices
2. Initially, let V = {a} and U = V − {a}
3. Let U be the unvisited and V be the visited vertices
4. Let Dist[t] = w[(a, t)] for every vertex a OE A
5. Select the vertex in U that has the smallest value Dist[x]. Let x denote this vertex.
6. If x is the vertex we wish to reach from a, goto 9. If not, let

   V = V − {x} and U = U − {x}
7. For every vertex t in A, compute Dist[t] with respect to V as,

   Dist[t] = min{Dist[t], Dist[t] + w (x,t)}
8. Repeat steps 5, 6, and 7
9. Stop

Let us consider the following graph, and let us compute the shortest path between a and all other vertices using this algorithm.

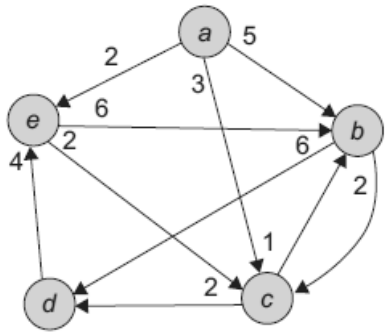


Figure 5.40. Directed weighted graph

Step 1: As initial step, The set $V = \{a\}$, where a is the source vertex and $U = \{b, c, d, e\}$ is the set of unvisited vertices. Dist[ ] = {−, 5, 3, ∞, 2}. This array can also be written as

	b	c	d	e
Distance	5	3	∞	2

This Dist[ ] array represents the current shortest distance between a and other vertices. Path = {Ø, *ab*, *ac*, Ø, *ac*}

Step 2: Now, the distance to vertex e is the shortest, so e is added to set V. We get, $V = \{a,e\}$; Let us update Dist array now.

	b	c	d	e
Distance	5	3	6	2

The weight of the edge between the current selected vertex e and d is 4 and the distance from a to e is 2; hence the distance between a and d becomes 6 as it is less than ∞.

Hence, the path is also updated for vertex d by the path of current selected vertex, that is, the path of e. Path = {Ø, *ab*, *ac*, *aed*, *ae*}

Step 3: Now the distance to vertex c among the unvisited vertices is the shortest. Hence, c is current selected vertex which gets to V. Therefore $V = \{a, e, c\}$. Let us update Dist array now.

	b	c	d	e
Distance	4	3	5	2

Here, the shortest distance between the source *a* to *b* and *d* are updated as,

Dist[*b*] = min{5, Dist[*c*] + w(*c*, *b*)}

= min{5, 3 + 1}

= 4

and

Dist[*d*] = min{6, Dist[*c*] + w(*c*, *d*)}

= min {6, 3 + 2}

= 5

As the shortest distance of *b* and *d* are updated, their respective paths are also updated as in the following expression:

Path = {Ø, *acb*, *ac*, *acd*, *ae*}

The path vector can also be shown as follows:

Path	*b*	*c*	*d*	*e*
	acb	*ac*	*acd*	*ae*

Step 4: Now *b* is the vertex that has the shortest distance and is unvisited. Hence,

V = {*a*, *e*, *c*, *b*}

Distance	*b*	*c*	*d*	*e*
	4	3	5	2

Here, none of the shortest distances is updated. Hence, the path also remains unchanged.

Path	*b*	*c*	*d*	*e*
	acb	*ac*	*acd*	*ae*

Step 5: Now *d* is the next selected vertex, and the final distance and path vectors are the same as stated.

Hence, the shortest distances between *a* and {*b, c, d, e*} are {4, 3, 5, 2}, respectively. In addition, the shortest path between *a* and {*b, c, d, e*} are {*acb, ac, acd, ae*}, respectively.

In the final two steps, adding the vertices *b* and *d* to *V* yield the paths and distances as shown in following figure.

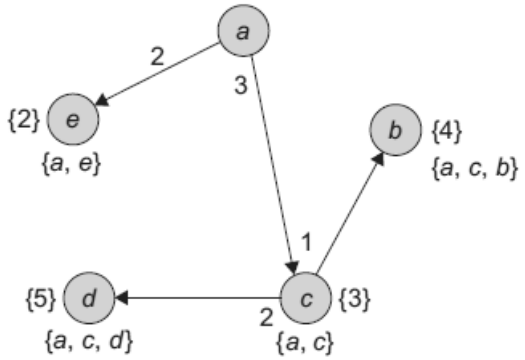


Figure 5.41. Shortest Paths and Distances

TWO MARKS QUESTIONS AND ANSWERS

1. **Define a Tree Data Structure.**

A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r, called the root, and zero or more nonempty (sub) trees T1, T2,…,Tk, each of whose roots are connected by a directed edge from r.

2. **Define a binary tree.**

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left sub-tree and right sub-tree.

3. **Define a full binary tree.**

A full binary tree is a tree in which all the leaves are on the same level and every non-leaf node has exactly two children.

4. **Define a complete binary tree.**

A complete binary tree is a tree in which every non-leaf node has exactly two children not necessarily to be on the same level

5. **State the properties of a binary tree.**

- The maximum number of nodes on level n of a binary tree is 2n-1, where n≥1.
- The maximum number of nodes in a binary tree of height n is 2n-1, where n≥1.
- For any non-empty tree, nl=nd+1 where nl is the number of leaf nodes and nd is the number of nodes of degree 2.

6. **State the merits of linear representation of binary trees.**

- Storage method is easy and can be easily implemented in arrays
- When the location of a parent/child node is known, other one can be determined easily
- It requires static memory allocation so it is easily implemented in all programming Language.

7. Define a binary search tree.

A binary search tree is a special binary tree, which is either empty or it should satisfy the following characteristics:

- Every node has a value and no two nodes should have the same value i.e) the values in the binary search tree are distinct
- The values in any left sub-tree is less than the value of its parent node
- The values in any right sub-tree is greater than the value of its parent node
- The left and right sub-trees of each node are again binary search trees

8. Why it is said that searching a node in a binary search tree is efficient than that of a simple binary tree?

In binary search tree, the nodes are arranged in such a way that the left node is having less data value than root node value and the right nodes are having larger value than that of root. Because of this while searching any node the value of the target node will be compared with the parent node and accordingly either left sub branch or right sub branch will be searched. So, one has to compare only particular branches. Thus searching becomes efficient.

9. What is the use of threaded binary tree?

In threaded binary tree, the NULL pointers are replaced by some addresses. The left pointer of the node points to its predecessor and the right pointer of the node points to its successor.

10. What is an expression tree?

An expression tree is a tree which is build from infix or prefix or postfix expression. Generally, in such a tree, the leaves are operands and other nodes are operators.

11. What do you mean by balanced trees?

Balanced trees have the structure of binary trees and obey binary search tree properties. Apart from these properties, they have some special constraints, which differ from one data structure to another. However, these constraints are aimed only at reducing the height of the tree, because this factor determines the time complexity.

12. Define splay tree.

A splay tree is a binary search tree in which restructuring is done using a scheme called splay. The splay is a heuristic method which moves a given vertex v to the root of the splay tree using a sequence of rotations.

13. What is the idea behind splaying?

Splaying reduces the total accessing time if the most frequently accessed node is moved towards the root. It does not require to maintain any information regarding the height or balance factor and hence saves space and simplifies the code to some extent.

14. List the types of rotations available in Splay tree.

Let us assume that the splay is performed at vertex v, whose parent and grandparent are p and g respectively. Then, the three rotations are named as:

- Zig: If p is the root and v is the left child of p, then left-left rotation at p would suffice. This case always terminates the splay as v reaches the root after this rotation.
- Zig-Zig: If p is not the root, p is the left child and v is also a left child, then a left- left rotation at g followed by a left-left rotation at p, brings v as an ancestor of g as well as p.
- Zig-Zag: If p is not the root, p is the left child and v is a right child, perform a left-right rotation at g and bring v as an ancestor of p as well as g.

15. What is a heap?

A heap is a partially ordered data structure, and can be defined as a binary tree assigned to its nodes, one key per node, provided the following two conditions are met,

The tree's shape requirement-The binary tree is essentially complete, that is all the leaves are full except possibly the last level, where only some rightmost leaves will be missing.

The parental dominance requirement-The key at each node is greater that or equal to the keys of its children

16. What are the advantages of using a heap?

Heaps are especially suitable for implementing priority queues. Priority queue is a set of items with orderable characteristic called an item's priority, with the following operations

- Finding an item with the highest priority
- Deleting an item with highest priority
- Adding a new item to the set

17. Define Graph.

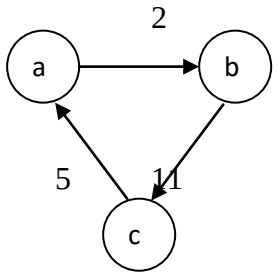
A graph G consist of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set for edge E to a set of pairs of elements of V. It can also be represented as G=(V, E).

18. Define adjacent nodes.

Any two nodes which are connected by an edge in a graph are called adjacent nodes. For example, if an edge x ε E is associated with a pair of nodes (u,v) where u, v ε V, then we say that the edge x connects the nodes u and v.

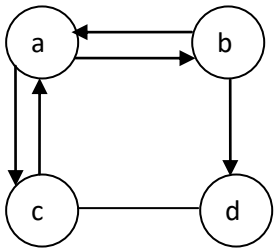
19. Define a weighted graph.

A graph is said to be weighted graph if every edge in the graph is assigned some weight or value. The weight of an edge is a positive value that may be representing the distance between the vertices or the weights of the edges along the path.



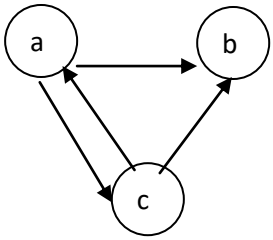
20. Define parallel edges.

In some directed as well as undirected graph certain pair of nodes are joined by more than one edge, such edges are called parallel edges.



21. Define total degree of a graph.

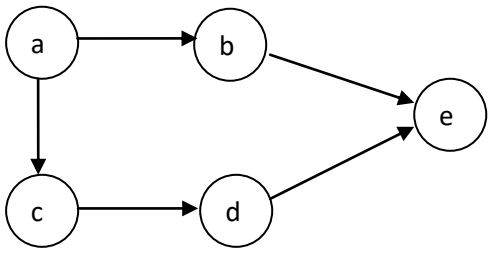
The sum of the In degree and out degree of a node is called the total degree of the node. Ex : total degree of a node c = 1 +2 =3



22. Define a path in a graph.

A path in a graph is defined as a sequence of distinct vertices each adjacent to the next, except possibly the first vertex and last vertex is different.

The path in a graph is the route taken to reach the terminal node from a starting node.



The path from 'a' to 'e' are

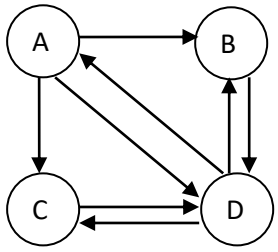
P1 = ((a,b),(b,e))

P2 = ((a,c),(c,d),(d,e))

23. What is the use of Dijksra's algorithm?

Dijkstra's algorithm is used to solve the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph, find the shortest path to all its other vertices. The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may have edges in common.

24. Give the adjacency matrix representation for the following



A	B	C	D
0	1	1	1
0	0	0	1
0	0	0	1
1	1	1	0

25. What is meant by strongly connected in a graph?

An undirected graph is connected, if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

26. When is a graph said to be weakly connected?

When a directed graph is not strongly connected but the underlying graph is connected, then the graph is said to be weakly connected.

27. What is an undirected acyclic graph?

When every edge in an acyclic graph is undirected, it is called an undirected acyclic graph. It is also called as undirected forest.

28. What is a minimum spanning tree?

A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at the lowest total cost.

A spanning tree of a connected graph G, is a tree consisting of edges and all the vertices of G.

In minimum spanning tree T, for a given graph G, the total weights of the edges of the spanning tree must be minimum compared to all other spanning trees generated from G.

-Prim's and Kruskal is the algorithm for finding Minimum Cost Spanning Tree.

29. What is the use of Kruskal's algorithm and who discovered it?

Kruskal's algorithm is one of the greedy techniques to solve the minimum spanning tree problem. It was discovered by Joseph Kruskal when he was a second-year graduate student.

30. Prove that the maximum number of edges that a graph with n Vertices is n*(n-1)/2.

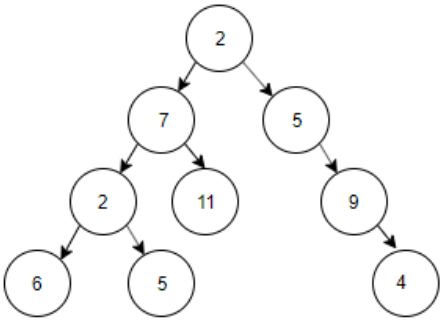
Choose a vertex and draw edges from this vertex to the remaining n-1 vertices. Then, from these n-1 vertices, choose a vertex and draw edges to the rest of the n-2 Vertices. Continue this process till it ends with a single Vertex.

Hence, the total number of edges added in graph is

$(n-1)+(n-2)+(n-3)+\ldots+1 = n*(n-1)/2$.

SOLVED PROBLEMS:

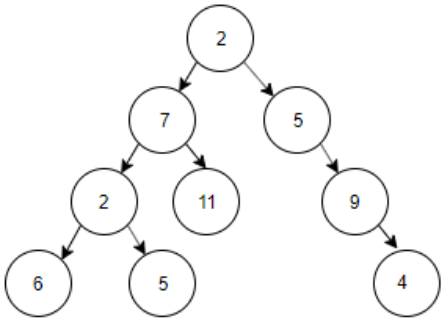
1. **For the tree below, write the pre-order traversal.**



Solution:

Pre order traversal follows NLR (Node-Left-Right). The Pre-order traversal of above tree is: 2, 7, 2, 6, 5, 11, 5, 9, 4.

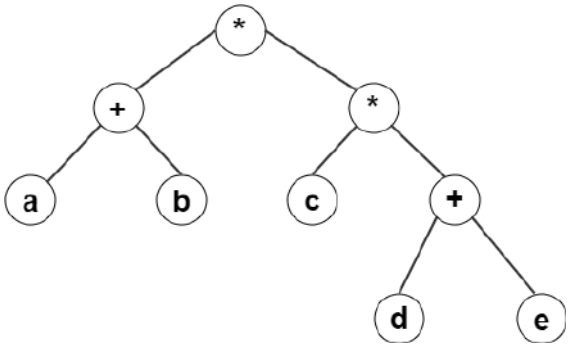
2. **For the tree below, write the post-order traversal.**



Solution:

Post order traversal follows LRN (Left-Right-Node). The Post-order traversal of above tree is: **2, 5, 11, 6, 7, 4, 9, 5, 2.**

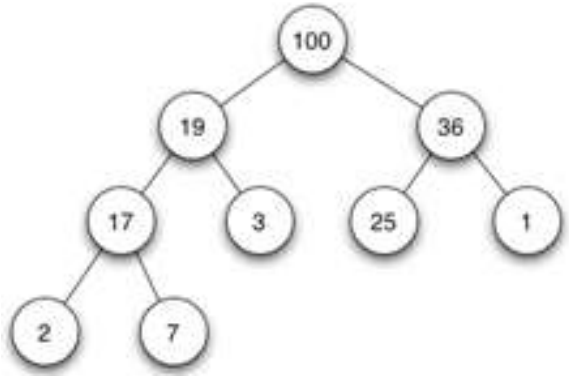
3. **What is the postfix expression for the following expression tree?**



***Solution*:**

If the given expression tree is evaluated, the postfix expression **a b + c d e + * *** is obtained.

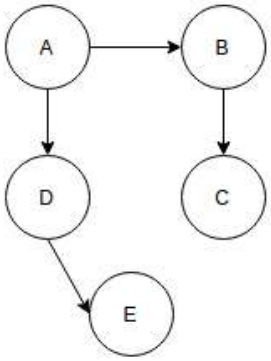
4. **If we implement heap as maximum heap for the following tree, adding a new node of value 15 to the left most node of right subtree . What value will be at leaf nodes of the right subtree of the heap?**



Solution:

As 15 is less than 25 so there would be no violation and the node will remain at that Position. So leaf nodes with value 15 and 1 will be at the position in right sub tree.

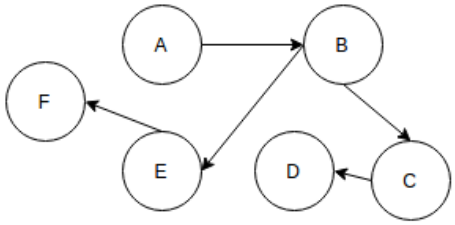
5. What would be the DFS traversal of the given Graph?



Solution:

In this case two answers are possible including ABCED and ADEBC.

6. Find the topological sorting of the given graph.

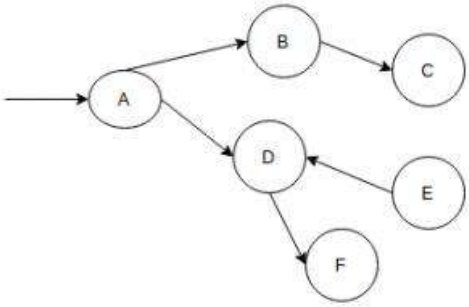


Solution:

Topological sorting is a linear arrangement of vertices such that for every directed edge ***uv*** from vertex u to vertex v, u comes before v in the ordering.

So, Answer is A B C D E F, A B F E D C and A B E C F D.

7. What sequence would the BFS traversal of the given graph yield?

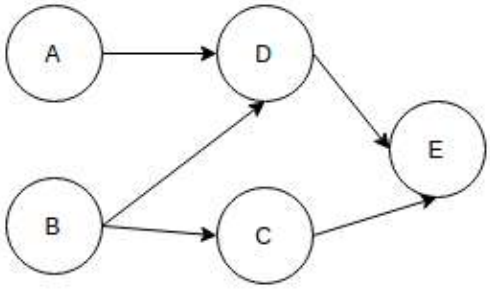


Solution:

In BFS, a node gets explored and then the neighbors of the current node get explored, before moving on to the next levels.

So, Answer is A B D C F.

8. In the following DAG, find out the number of required Stacks in order to represent it in a Graph Structured Stack.



Solution:

Paths ADE, BDE and BCE are possible. So, the number of required Stacks in order to represent it in a Graph Structured Stack is 3.

9. Which of the following graphs are isomorphic to each other?

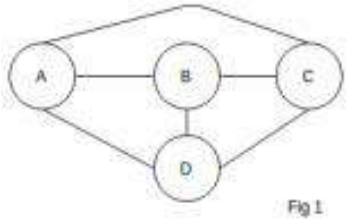


Fig 1

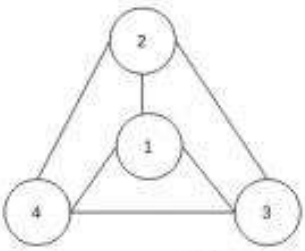


Fig 2

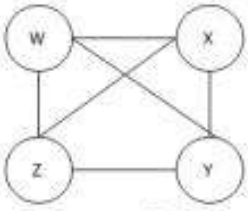
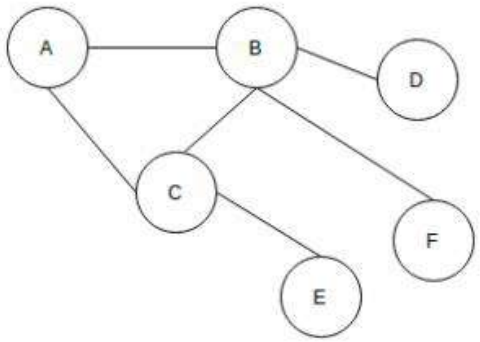


Fig 3

Solution:

All three graphs are complete graphs with 4 vertices. So, all graphs are isomorphic to each other.

10. In the given graph which edge should be removed to make it a Bipartite Graph?



Solution:

The resultant graph would be a Bipartite Graph having {A, C, E} and {D, B} as its subgroups. So, edge A-C should be removed.