LEARNING MATERIAL

ON

DATA STRUCTURE

(3RD SEMESTER)

DEPARTMENT OF INFORMATION TECHNOLOGY

Prepared by
Ms. Manalisa Giri
Lect. IT
Govt.Polytechnic,Bhubaneswar

Subject Name : Data structure 3rd semester IT

1.0 INTRODUCTION

Section 1.1Data:

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing, by human or electronic machine. Data items refer to single unit of values. Data are characteristics or information, usually numerical that are collected through observation. In a more technical sense, data is a set of values of qualitative or quantitative variables about one or more persons or objects. They are quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

Information:

Information is organized or classified data, which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based.

For the decision to be meaningful, the processed data must qualify for the following characteristics −

- **Timely** − Information should be available when required.
- **Accuracy** − Information should be accurate.
- **Completeness** − Information should be complete.

Data types:

A *data type* is the most basic and the most common classification of data. It is this through which the compiler gets to know the form or the type of information that will be used throughout the code. Data Type is the kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of the given data type only. Some basic examples are integer, float, double, character, string etc

Section 1.2 Data structure

Data may be organized in many different ways**; the logical or mathematical model of a particular organization of data is called data**

structure. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structure introduction refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

A data structure should be seen as a logical concept that must address two fundamental concerns.

1. How the data will be stored, and

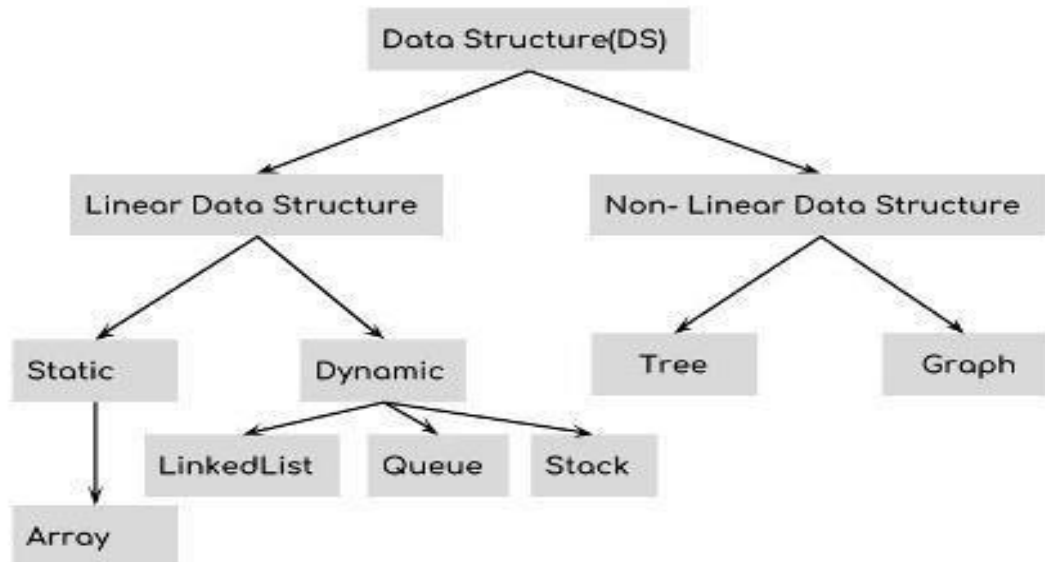
2. What operations will be performed on it.

Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include **lists, arrays, stacks, queues, trees, and graphs**.

Classification of Data Structures:

Data structures can be classified as

- Simple data structure
- Compound data structure
- Linear data structure
- Non linear data structure



Simple Data Structure: Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

Compound Data structure: Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as (i) Linear data structure and (ii) Non-linear data structure

Linear Data Structure: Linear data structures can be constructed as a

continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the data elements.

Non-linear Data Structure: Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the data items.

Data structure operations

The data appearing in our data structures are processes by means of certain operations. In fact the particular data structure that one chooses for a given situation depends largely on the frequency with which specific

operation s are performed. The followings are the operations that can be performed on Data Structures:

1. **Creation**: Creating a data structure according to the requirement.
   eg: an integer array of 5 values.
   int ar[5];   //ar is the name of array
2. **Insertion**: Inserting values into data structure. There can be three ways to insert elements into data structure: at the beginning, at the end and at the desired location.
3. **Traversal**: Accessing each record or Visiting each element of the data structure at least once.
4. **Search**: Finding the location with a given key value or Searching of an element in the given number of elements. The elements can be searched in two ways:

a. Linear Search: Simplest way of searching an element.
b. Binary Search: It works on divide and conquer rule.

5. **Sorting**: Rearranging the elements in a particular order, ascending or descending order. There are several sorting algorithms: a. Bubble Sort

b. Selection Sort
c. Quick Sort
d. Merge Sort
e. Heap Sort

6. **Merging**: Combining the data items of two sorted files into single file in the sorted form.
7. **Updation**: Updating the current value in the data structure with some new value.
8. **Deletion**: Deleting the undesired value from the data structure. There are 3 ways to delete a value from data structure. These are: from the beginning, from the end and from the given location.
9. **Copying:** To make a duplicate of the data structure.
10. **Concatenation**: Simple joining of two or more strings to make a single string.

Section 1.3 Abstract data type

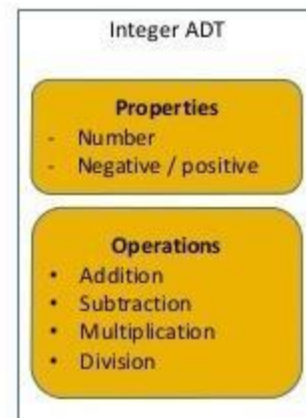
An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed

without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called information hiding. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

An **abstract data type** is defined by its behavior (semantics) from the point of view of a user, of the **data**, specifically in terms of possible values, possible operations on **data** of this **type**, and the behavior of these operations.

Example of Abstract Data Type (ADT)

- Integer
  - ...., -4, -3, -2, -1, 0, 1, 2, 3, 4 ...



Abstract Data Types

- Abstract data type (ADT)
  - An ADT is composed of
    - A collection of data
    - A set of operations on that data
  - Specifications of an ADT indicate
    - What the ADT operations do, not how to implement them
  - Implementation of an ADT
    - Includes choosing a particular data structure
    - A data structure is a construct that can be defined in a programming language to store a collection of data

Section 1.4 Algorithms and its complexities

An algorithm is a well defined list of steps for solving a particular problem. Sometimes, there is more than one way to solve a problem. Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.

- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

We need to learn to develop efficient algorithm for the processing of our data. It is required to compare the performance of different algorithms and choose the best one to solve a particular problem. While analyzing algorithms, we mostly focus on two major measures of the efficiency of an algorithm:

1. **Space**
2. **Time**

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

The Complexity of an **algorithm** is the function or measure which gives the amount of running time and/or space required by an **algorithm** for an input of a given size (n). Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

- **Time complexity of an algorithm** quantifies the amount of **time** taken by an **algorithm** to run as a function of the length of the input.
- Similarly, **Space complexity of an algorithm** quantifies the amount of **space** or **memory** taken by an **algorithm** to run as a function of the length of the input.

Time and space complexity depends on lots of things like

- Hardware, Operating system, Processors,.
- Input
- Programming language
- Coding skill
- compiler

However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Three **types** of **complexity** could be considered when analyzing algorithm performance.

- Worst-case **complexity** Best-case **complexity**, and Average-case **complexity**.

Only worst-case **complexity** has found to be useful.

The **best-case complexity** of the algorithm is the function defined by the minimum number of steps taken on any instance of size n. It represents the curve passing through the lowest point of each column.

(A)Time Complexity

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time**

complexity of an algorithm because that is the maximum time taken for any input size.

Now various notations used for Time Complexity are

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

(B)Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stacks space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic.

Section 1.5Time Space Tradeoff

In computer science, the algorithms are evaluated by the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (**time complexity**) or storage locations (**space complexity**).

The space-time tradeoff refers to a choice between algorithmic solutions of a data processing problem that allows one to decrease the running time of an algorithmic solution by increasing the space to store the

data and vice versa. We may take several examples of searching and sorting techniques to compare their time and space complexity to measure the Time-Space trade-off.

Example- while printing a telephone directory every year, a separate temporary file for new telephone customer is created and subsequently the directory is updated.

CHAPTER 2.0 STRING PROCESSING

Section 2.1 Basic terminology

Each programming language contains a character set which is used to communicate with the computer. This set includes the following:

Alphabets: A B C D E……Z, a b c d…..z

Digits: 0 1 2 3 4 5 6 7 8 9

Special Characters: + _ * / ? ( ) & % $ # @ ! = - ' "

A finite sequence S of zero or more characters is **called String**.

In computer programming, a **string** is traditionally a sequence of characters, either as a literal constant or as some kind of variable. A **string** is generally considered as a **data** type and is often implemented as an array **data structure** of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding. ... When a **string** appears literally in source code, it is known as a **string** literal or an anonymous **string**. Depending on the programming language and precise data type used, a variable declared to be a string may either cause storage in memory to be statically allocated for a predetermined maximum length or employ dynamic allocation to allow it to hold a variable number of elements.

The number of characters in a string is called its **length**. The string with zero characters is called the empty **string or null string.**

Examples 'THE END' length of the string is 7.

'TO BE NOT TO BE' length of the string is 15.

'' The length of the string is 0.

Let S_1 and S_2 are two strings.The string consisting of the characters of S_1 followed by the characters of S_2 is called Concatenation. S_1 and S_2 are denoted by "S_1 //S_2".

For Example

‘THE’ //’ END’ = ‘THEEND’

‘THE’ //’ E’ ‘ //’END’= ‘THE END’

The Small square is used for blank space.Clearly the length of “S_1 //S_2” is qual to the sum of the lengths of the strings S_1 and S_2.

A string Y is called a **substring** of a string S if there exist strings X and Z such that

S = X//Y//Z

If X is an empty string then Y is called an **initial substring** of S, and if Z is an empty string then Y is called a **terminal string**.

STORING STRING

Strings are stored in three types of structure

1. Fixed length structure (Record oriented)
2. Variable length structure with fixed maximum
3. Linked structure

1. In Fixed length storage each line of print is viewed as a record where all records have the same length. The disadvantages are

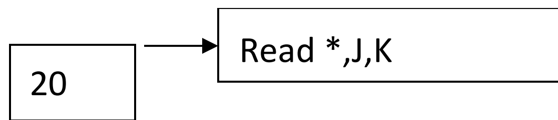
- Time is wasted reading an entire record
- Certain records may require more space than available
- Changing the misspelled word requires the entire record to be changed

2. The Storage of Variable length structure with fixed maximum can be done in two general ways:

(a) One can use a marker such as two dollar signs ($$) to signal the end of the string.

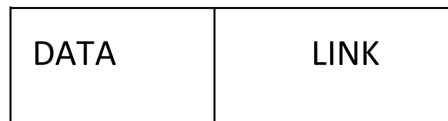
```
Read *,J,K$$
IF(J.LE.K) THEN $$
```

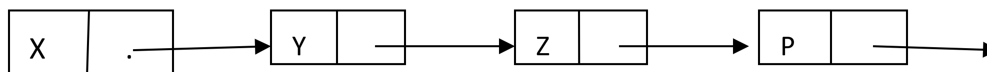
(b)One can list the length of the string- as an additional item in the pointer array. We define here the maximum length of the string. Here maximum length is 20.



This method of storing strings will obviously save space and used in memory when records are relatively permanent. The disadvantage is that storage are usually inefficient when the strings and their lengths are frequently being changed.

3.The Linked Storage we mean a linearly ordered sequence of memory cells called nodes, where each node contains two parts, one part contains **DATA** and the other part called a **LINK**, which points to the next node in the list.





Section 2.2 CHARACTER DATA TYPE:-

The character data type is of two data type.

(1) Constant

(2) Variable

Constant String:

The constant string is fixed & is written in either ‘single quote or “double quotation. It stores fixed sequence of character. Ex:- ‘THE END’ and “TO BE OR NOT TO BE”

These are string constants of length 7 and 18 characters respectively.

Variable String:

String variable are variables that contain not just numbers but also special characters such as “/”,”-“ or “.” and so on anything a keyboard may produce. String variable falls into 3 categories.

1. Static
2. Semi-Static

3. Dynamic

1.Static character variable:

Static character variable is defined before the program can be executed & can not change throughout the program Example:

2.Semi-static variable:

Whose length variable may as long as the length does not exist, a maximum value. A maximum value determine by the program before the program is executed.

3.Dynamic variable:

A variable whose length can change during the execution of the program.

Section 2.3 String Operation:

There are four different operations.

1. Sub string
2. Indexing
3. Concatenation
4. Length

1.Sub string:-

Group of conjunctive elements in a string (such as wards, purchases or sentences) called substring.

Accessing substring of a given string required 3 pieces of information.

a. The name of the string or the string itself.
b. The position of the first character of the substring in the given string.
c. The length of the substring of the last character of the substring.
   We called this operation SUBSTRING.

The format of substring as follows:

SUBSTRING (String, initial, length)

To denote the substring of string 'S' beginning in the position 'K' having a length

‘L‘.

SUBSTRING (S, K, L)

Let string ‘TO BE NOT TO BE’ and K=4,L=7

For e.g.; SUBSTRING (‘TO BE OR NOT TO BE‘, 4, 7)
SUBSTRING=’BE OR N’

SUBSTRING (THE END, 3, 4) SUBSTRING = ’E EN’.

2. INDEXING:-

Indexing also called pattern matching which refers to finding the position where a string pattern ‘P‘. First appears in a given string text ‘T‘, we called this operation index and write as **INDEX (text, pattern)**

If the pattern ‗P‘ does not appear in text ‗T‘ then index is assign the value 0; the argument & text and pattern can either string constant or string variable.

For e.g.; T contains the text.

‘**HIS FATHER IS THE PROFESSOR‘**

Then INDEX (T,‗THE‘) = 7

INDEX (T, ‘THEN‘) = 0

INDEX (T, ‘ESSO‘) = 23

3.Concatenation:-

Let $S1$ & $S2$ be the string then concatenation of S1 and S2 denoted by, $S1 \parallel S2$, each the string consist of the character of $S1$ followed by the characters of $S2$.

Ex:- S1= 'Sonali' S2 ' ' and S3 = 'Behera'

$S1 \parallel S2 \parallel S3$ = Sonali Behera

4.Length operation:-

The number of character in a string is called its length. We will write the format as

LENGTH (string).

For the length of a given string LENGTH (Computer). The length is 8.

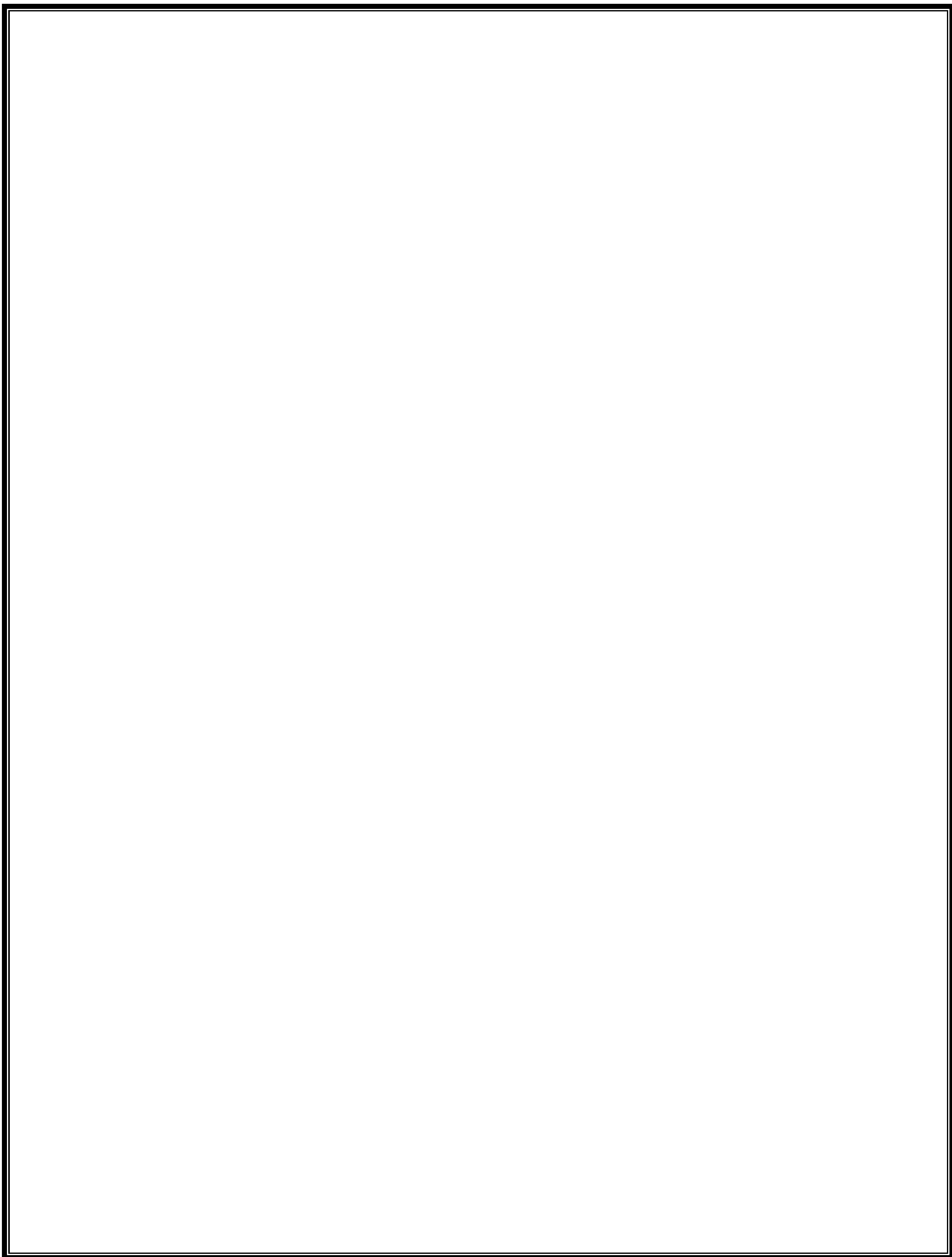
Basic language LEN (STRING)

Strlen (string)

LIST OF INBUILT C FUNCTIONS IN STRING.H FILE:

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2
strcmpi ( )	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
strchr ( )	Returns pointer to first occurrence of char in str1
strrchr ( )	last occurrence of given character in a string is found
strstr ( )	Returns pointer to first occurrence of str2 in str1

strrstr ( )	Returns pointer to last occurrence of str2 in str1
strdup ( )	Duplicates the string
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string
strset ( )	Sets all character in a string to given character
strnset ( )	It sets the portion of characters in a string to given character
strtok ( )	Tokenizing given string using delimiter



Strupper
(string)
String
upper
Strupr(‗
compute
r‘)
COMPUTER
String lower
Strlwr (‗COMPUTER‘)
COMPUTER

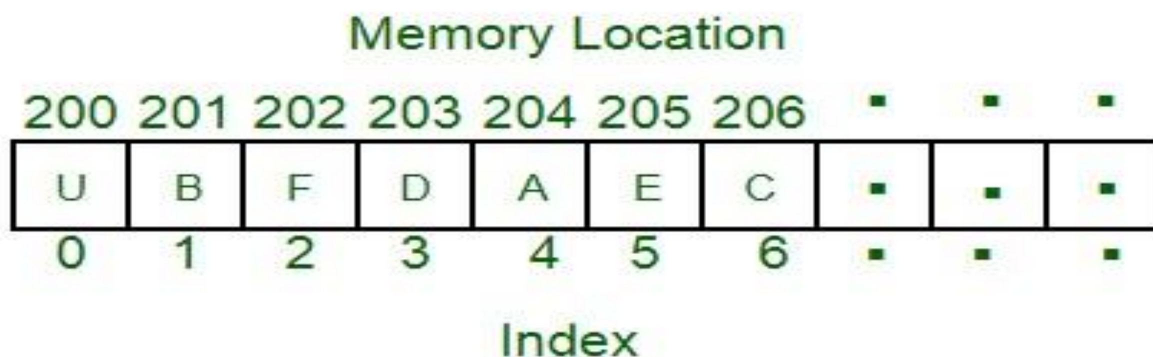
String
concatenating Strcnt String Reverse Strrev

CHAPTER 3.0 ARRAYS

Section 3.1 Introduction:

An **array** is a collection of homogeneous (same type) **data** items stored in contiguous memory locations. An **array** is a **data structure** for storing more than one **data** item that has a similar **data** type. The items of an **array** are allocated at adjacent memory locations. These memory locations are called elements of that **array**.

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e. the memory location of the first element of the array (generally denoted by the name of the array).



Why we need an array?

Array is particularly useful when we are dealing with lot of variables of the same type. For example, let's say I need to store the marks in math subject of 100 students. To solve this particular problem, either I have to create the 100 variables of integer type or create an array of int type with the size 100.

Obviously the second option is best, because keeping track of all the 100 different variables is a tedious task. On the other hand, dealing with array is simple and easy, all 100 values can be stored in the same array at different indexes (0 to 99).
For example, if an array is of type "int", it can only store integer elements and cannot allow the elements of other types such as double, float, char etc.

As we know the linear array consists of 'n' number of **homogeneous data elements** such that:

- The elements of the array are referenced respectively by an **index** set consists of n consecutive numbers.
- The elements of the array are stored respectively in **successive memory** locations.

The length or the number of data elements of the array can be obtained from the index set by the formula:

Length = UB – LB+1

Where UB is the largest index called **upper bound** and LB is the smallest index called the **lower bound** of the array.

Therefore length = UB when LB = 1

We use the subscript notation in brackets. The number K in LA[K] is called a subscript or an index and **LA[K] is called a subscripted variable**. Subscripts allow any element of LA to be referenced by its relative position in LA. Let DATA is the linear array having 06 elements.

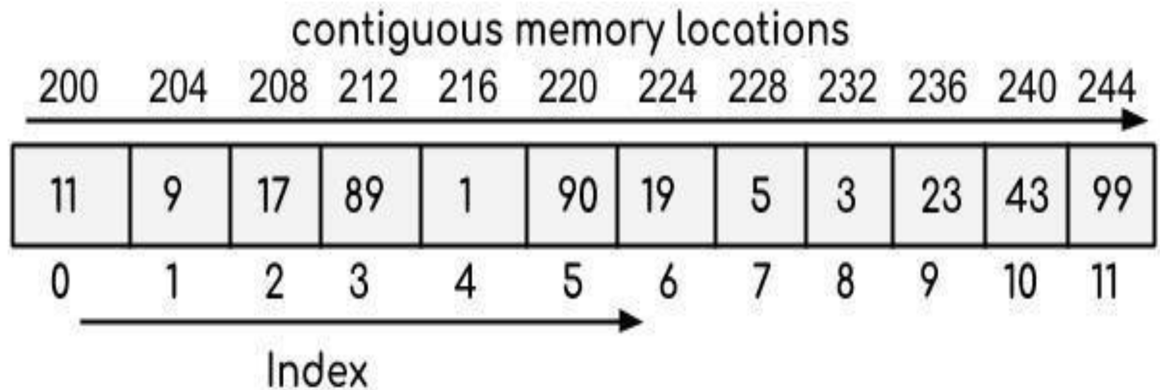
DATA

247	56	429	135	87	156
1	2	3	4	5	6

Here DATA[1] = 247, DATA[3] = 429 and DATA[6] = 156

Section 3.2 Representation Linear Array in Memory

The following diagram represents an integer array that has 12 elements. **The index of the array starts with 0**, so the array having 12 elements has indexes from 0 to 11.



Let LA is a linear array in the memory of the computer. Recall that the memory of computer is simply a sequence of addressed locations.

LOC (LA[K]) = address of element LA[K] of the array LA.

As previously noted, the elements of LA are stored in the successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by **Base (LA)**

and called the base address of LA. Using base address, the computer calculates the address of any element of LA by the following formula:

Let LA is a linear array, Base (LA) is the base address i. e the address of the first element of LA, the address of the Kth element of LA is defined by

LOC(LA[K]) = Base (LA) + (K – lower bound)
Here there is only single element in each location of the linear array LA.

When there is 'w' no. of elements are there in each location of the linear array LA, then the address of the Kth element of LA is defined by

LOC (LA[K]) = Base (LA) + w (K - lower bound)

Where w is the number of words per memory cell for the array LA. When the lower bound is '1' then the address of the Kth element of LA is defined by

LOC(LA[K]) = Base (LA) + (K – 1) and

LOC (LA[K]) = Base (LA) + w (K - 1) DATA

247	56	429	135	87	156
1	2	3	4	5	6

Now address of the 3rd location will be

LOC(LA[3]) = Base (LA) + (3– 1) =Base(LA) + 2 after

base two location

Section 3.4 Two dimensional and Multidimensional Array

Multidimensional Array

1. Array having more than one subscript variable is called **Multi-Dimensional array**.
2. Multi-Dimensional Array is also called as **Matrix.**
3. **Consider the Two-dimensional array –**

- Two Dimensional arrays are also called table or matrix; Two Dimensional arrays have two subscripts Variable.
- One Subscript Variable denotes the —**Row**‖ of a matrix
- Another Subscript Variable denotes the —**Column**‖ of a matrix
- Two dimensional arrays in which elements are stored column by column is called as **column major matrix.**
- Two-dimensional array in which elements are stored row by row is called as **row major matrix**.
- First subscript denotes number of rows and second subscript denotes the number of columns.
- The simplest form of the Multi-Dimensional Array is the Two Dimensional Array.

A **two Dimensional array** has a type such as integer [][] or String[][], with **two** pairs of square brackets. ... The elements of a **2D array** are arranged in rows and

columns, and the new operator for **2D arrays** specifies both the number of rows and the number of columns.

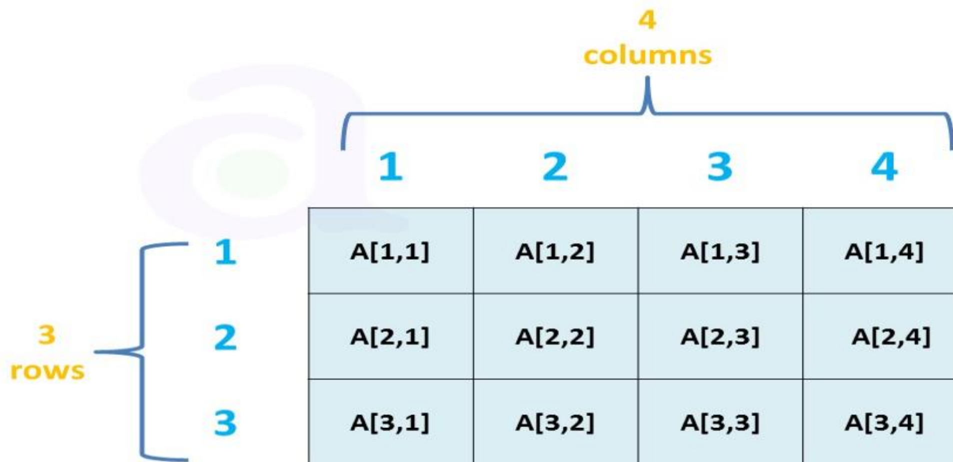


Fig: Two-dimension 3x4 array

Declaration and Use of Two Dimensional Array :

```
int a[3][4];
```

Use :

```
for(i=0;i<row;i
++)
  for(j=0;j<col;j++)
  {
  printf("%d",a[i][j]);
}
```

Meaning of Two Dimensional Array :

1. Matrix is having 3 rows (i takes value from 0 to 2)
2. Matrix is having 4 Columns (j takes value from 0 to 3)
3. Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.
4. Name of 2-D array is _**a**_ and each block is identified by the row & column Number.
5. Row number and Column Number Starts from 0.

Cell Location	Meaning
a[0][0]	0th Row and 0th Column
a[0][1]	0th Row and 1st Column
a[0][2]	0th Row and 2nd Column
a[0][3]	0th Row and 3rd Column
a[1][0]	1st Row and 0th Column
a[1][1]	1st Row and 1st Column
a[1][2]	1st Row and 2nd Column
a[1][3]	1st Row and 3rd Column
a[2][0]	2nd Row and 0th Column
a[2][1]	2nd Row and 1st Column
a[2][2]	2nd Row and 2nd Column
a[2][3]	2nd Row and 3rd Column

Let A be a two-dimensional MXN Array. The programming language will store the array A either

- Column by column called Column –major order.
- Row by row called Row–major order.

When there is ‘w’ no. of elements are there in each location of the linear array LA, then the address of the Kth element of LA is defined by

LOC(LA[K]) = Base (LA) + w (K – 1)

Where ‘w’ is the number of words per memory cell for the array LA.

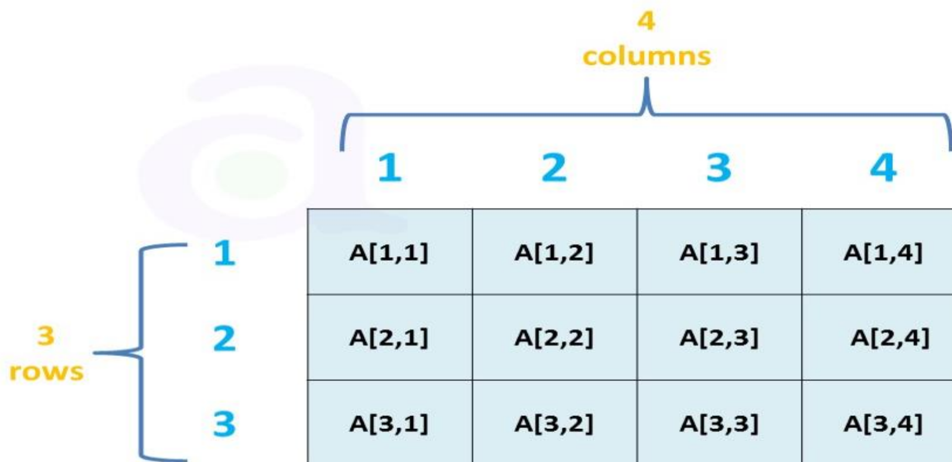


Fig: Two-dimension 3x4 array

Similar situations also holds good for any two dimentional MXN array 'LA'.The computer keeps track of Base(LA) – the address of the first element LA[1,1] of linear array.Now we can represent the matrix in two different forms:

Row major and column major to compute address LOC(LA[J,K]) of LA[J,K]

A(1,1)	COLUMN 1
A(2,1)	
A(3,1)	
A(1,2)	COLUMN 2
A(2,2)	
A(3,2)	
A(1,3)	COLUMN 3
A(2,3)	
A(3,3)	
A(1,4)	COLUMN 4
A(2,4)	
A(3,4)	

(A) Column –major order

A(1,1)	ROW 1

A(1,2)	
A(1,3)	
A(1,4)	
A(2,1)	ROW 2
A(2,2)	
A(2,3)	
A(2,4)	
A(3,1)	ROW 3
A(3,2)	
A(3,3)	
A(3,4)	

(B) Row–major order

A similar situation also holds for any two-dimensional MXN array 'LA'. We can compute the address LOC(LA[J,K]) of LA[J,K] using the formula

1.Row – major order **LOC(LA[J,K]) = Base(LA)+w[N(J - 1)+(K – 1)]**

2. Column – major order **LOC(LA[J,K]) = Base(LA)+w[M(K - 1)+(J – 1)]**

Where there are M rows and N columns in the two-dimensional array. We can find the address LOC(LA[J,K]) in time independent of J and K.

Pointers: A variable P is called a pointer if P points to an element in an array. P contains the address of an element in array. An array PTR is called a **pointer array if each element of PTR** is **a pointer.**

Section 3.3 OPERATIONS ON ARRAYS

Various operations that can be performed on an array

- Traversing
- Insertion
- Deletion
- Sorting
- Searching
- Merging

ARRAY TRAVERSAL ALGORITHM

Traversal in a **Linear Array** is the process of visiting each element once. **Traversal** operation **can** be used in counting the **array** elements, printing the values stored in an **array**, updating the existing values, increasing each element by value 2 or summing up all the element values.

In **traversing** operation of an **array**, each element of an **array** is accessed exactly for once for processing. This is also called visiting of an **array**.

Algorithm for Traversing a linear array:- Here LA is a linear array with lower bound LB and Upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA. PROCESS may be any operation which can be applied on each element of the Array.

Step 1 [Initialize counter]

Set K := LB

Step2 [Loop structure]

Repeat steps 3 and 4 while K <= UB

Step 3 [Visit element]

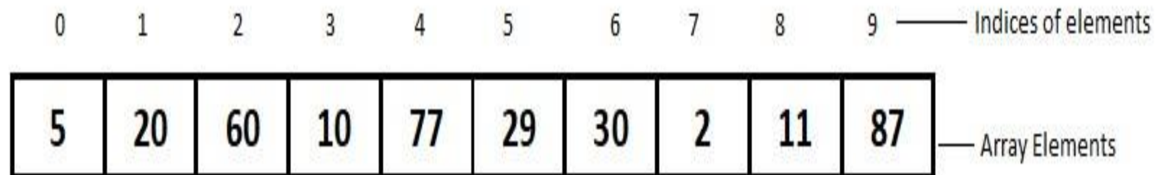
Apply PROCESS to LA[K]

Step 4 [Increase counter]

Set : K = K+1

Step 5 [finish]

Exit



Array of 10 integer elements

Here we can visit every location from the beginning and apply some operation on each and every location. The visit is made exactly once and till we reach at the end of the Array.

Algorithm for Inserting an element into a linear array:-

INSERT (LA,N,K,ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K<=N.This algorithm inserts an element **ITEM** into the Kth position in LA.

Step 1 [Initialize counter] Set

J :=N.

Step2 [Loop structure]

Repeat steps 3 and 4 while J>= K.

Step 3 [Move the Jth element downward] Set

LA[J+1] := LA[J].

Step 4 [Decrease counter] Set

:J= J-1.

Step 5 [Insert element]

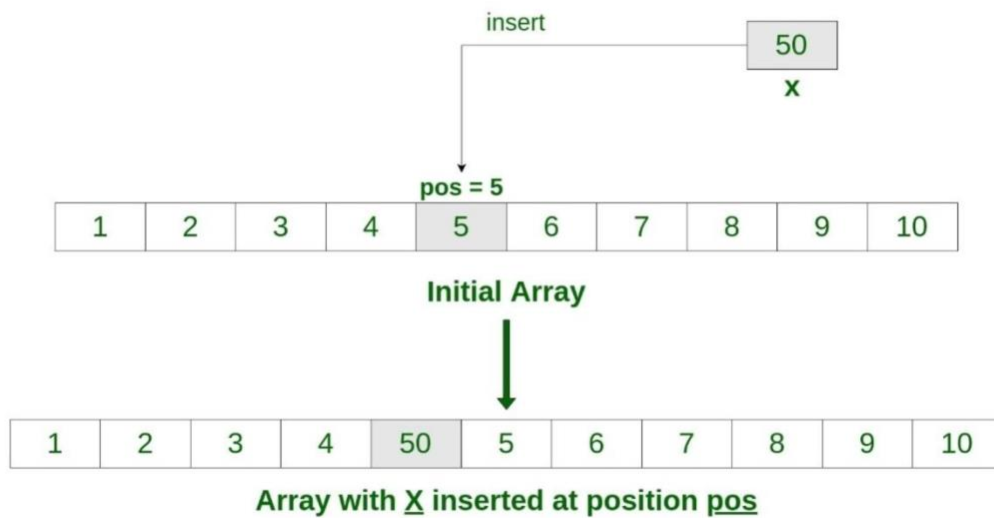
Set LA[K] := ITEM

Step 6 [Reset N]

Set N :=N+1.

Step 7 [Finish]

Exit.



Algorithm for deleting an element from a linear array:-

DELETE (LA,N,K,ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K<=N. This algorithm deletes the Kth element from linear array LA.

Step 1 [Move the Kth element from the array and store in ITEM] Set

ITEM:= LA[K].

Step 2 [Loop structure to move $J+1^{st}$ element upward]

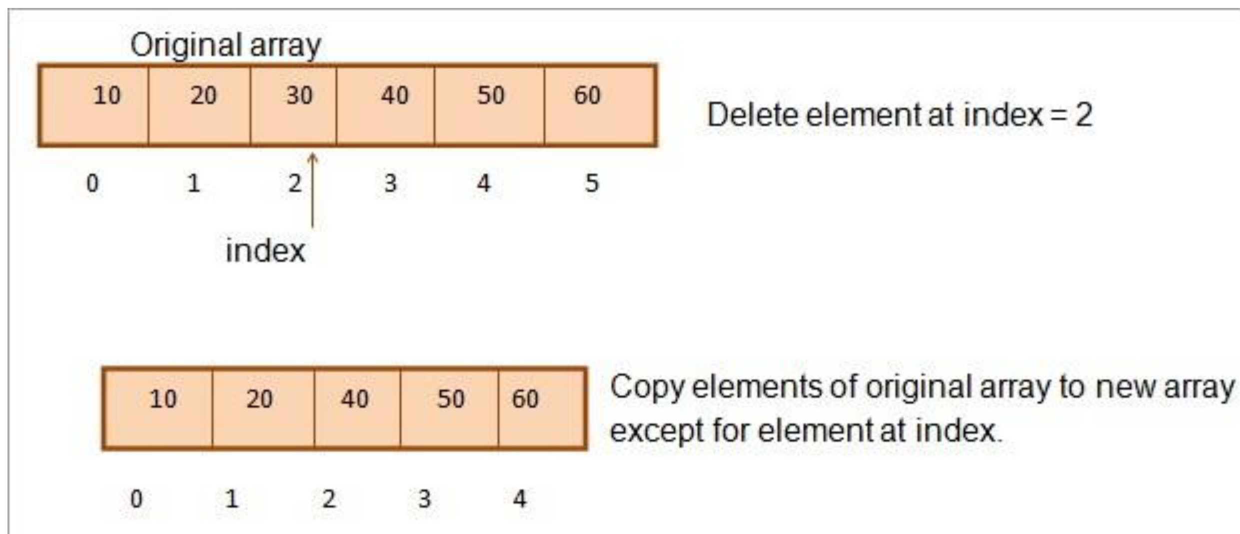
Repeat for J = K to N-1

Set LA[J]:= LA[J+1].

Step 3 [Reset the number N of Elements in LA]

Set N :=N-1. Step 4 [Finish]

Exit.



Section 3.5 Sparse Matrix

A **matrix** is a two-dimensional **data** object made of m rows and n columns, therefore having total m x n values. **If most of the elements of the matrix have 0 value, then it is called a sparse matrix.** ... Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

The **sparse matrix** is **represented** using three one-dimensional arrays for the non-zero values, the extents of the rows, and the column indexes. Compressed **Sparse** Column. The same as the Compressed **Sparse** Row method except the column indices **are** compressed and read first before the row indices.

Sparse Matrix Representations
A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

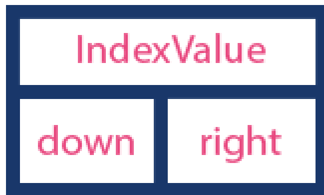
Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow.

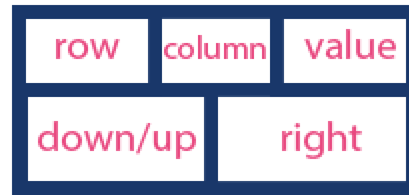
Linked Representation

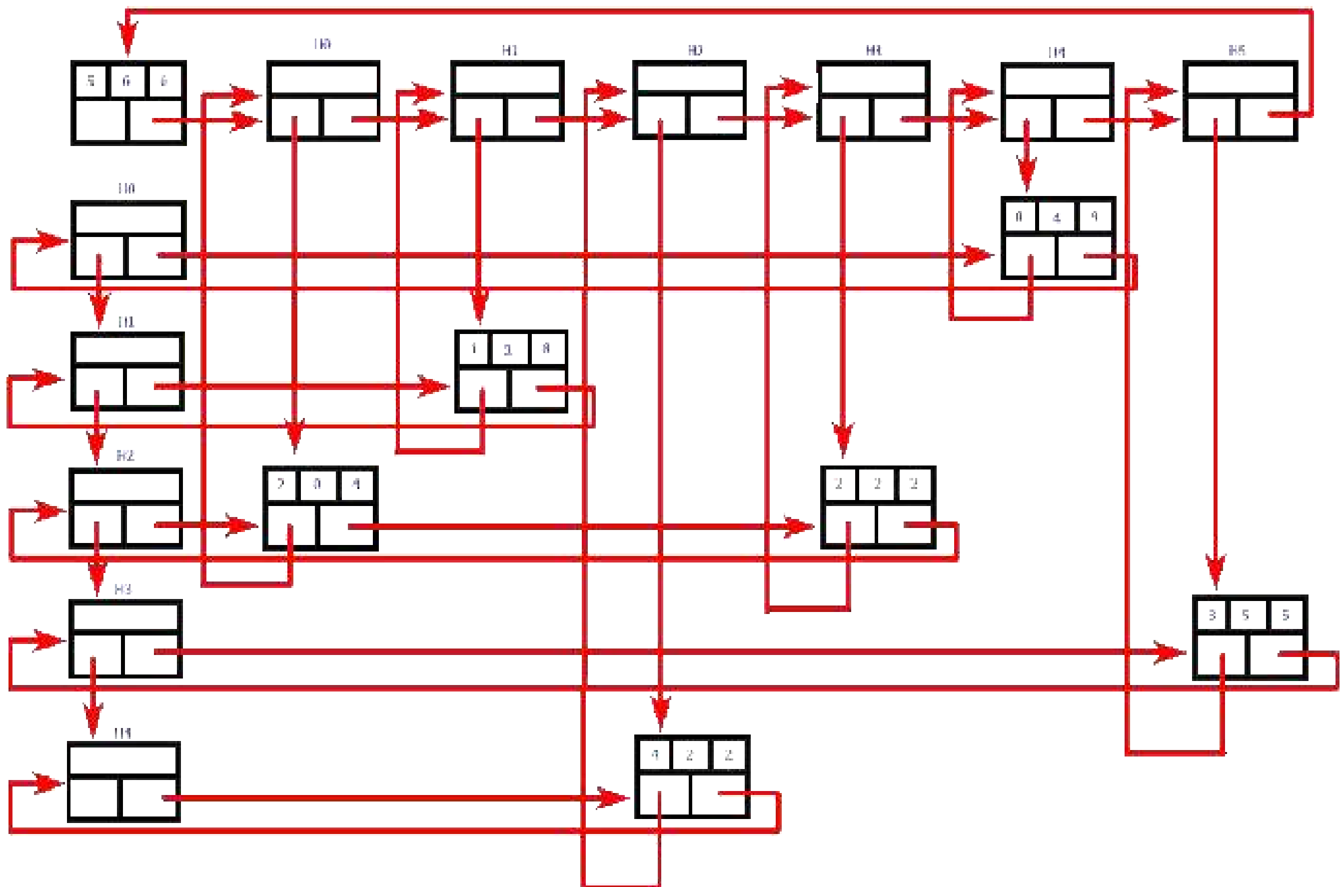
In linked representation, we use a linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image...

Header Node



Element Node





In the above representation, H0, H1,..., H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

Schaum’s outlines data structure

Seymour lipschutz

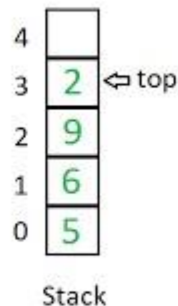
Mc graw-hill publication

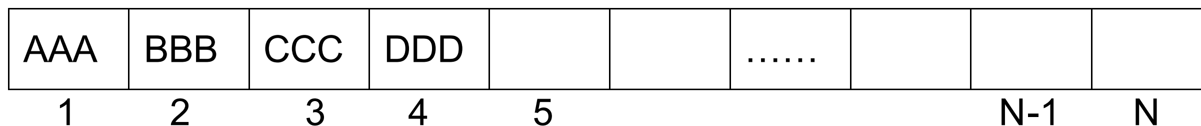
CHAPTER 4 STACKS AND QUEUES

A stack is a basic data structure that can be logically thought of as a linear structure represented by a real physical stack or pile, a structure where **insertion and deletio** of items takes place at one end called **top of the stack**.

A stack is an Abstract Data Type (ADT), commonly used in most programmin languages. It is named stack as it behaves like a real-world stack, for example – a dec of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place o remove a card or plate from the top of the stack only. Likewise, Stack ADT allows a data operations at one end only. At any given time, **we can only access the to element of a stack.**





AAA	BBB	CCC	DDD			……			
1	2	3	4	5				N-1	N

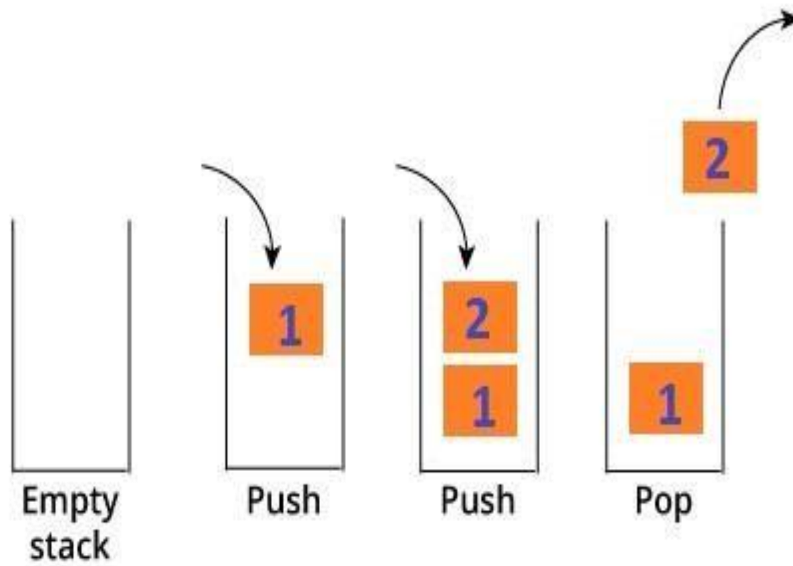
(TOP)

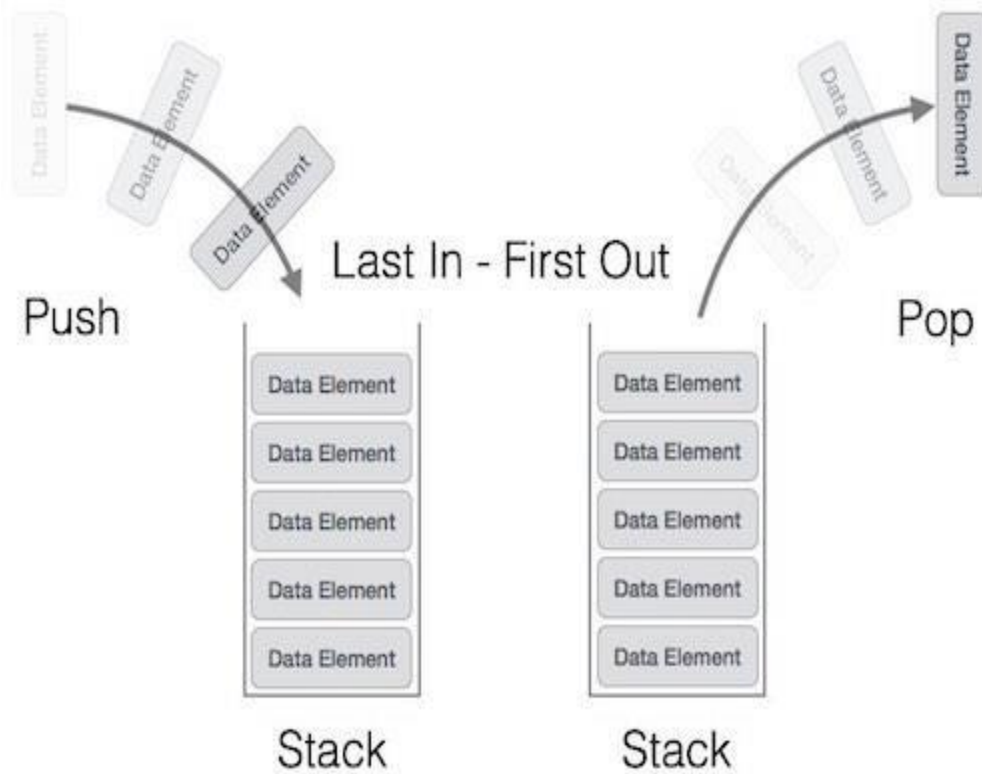
Stack has N no. of locations and TOP points the top most element in the stack.

Stack operations may involve the following two primary operations −

- **push()** − Pushing (storing) or inserting an element into the stack.
- **pop()** – Deleting/Removing (accessing) an element from the stack.

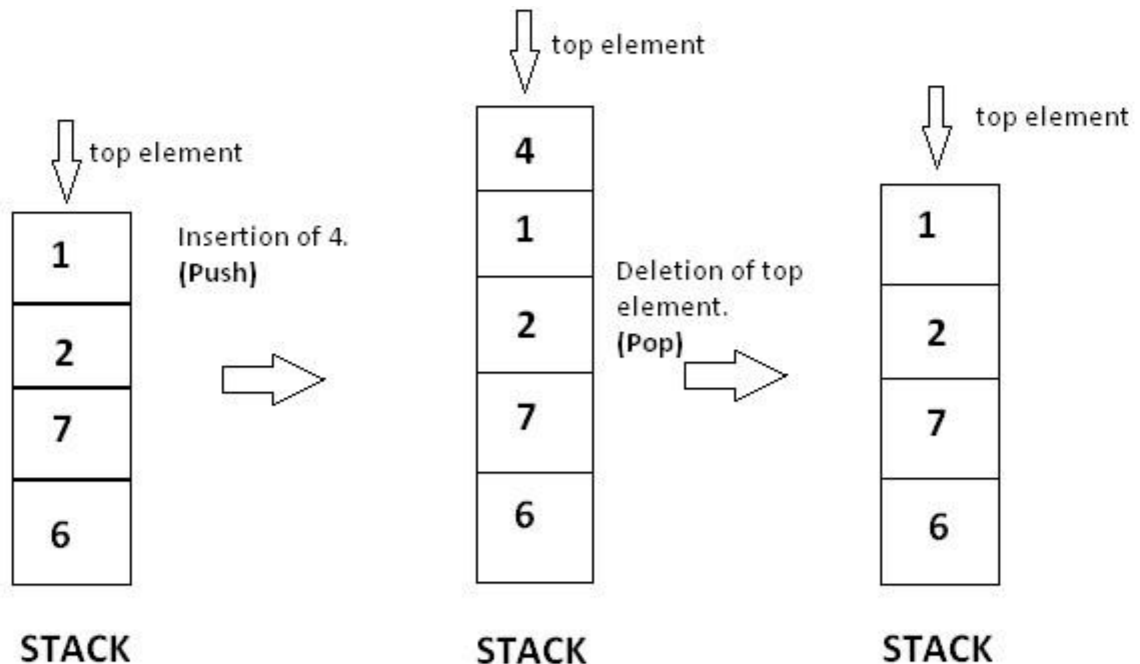
The following diagram depicts a stack and its operations –





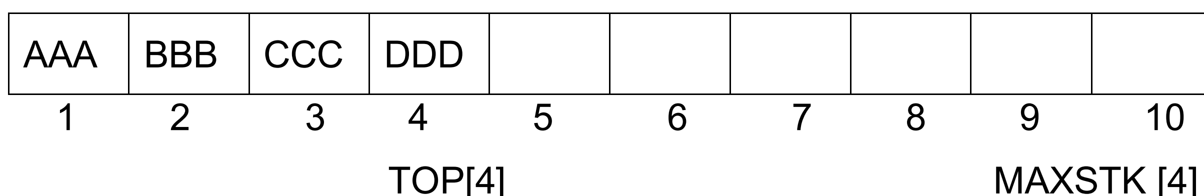
Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** or **First in last out (FILO)** principle. The last item to be inserted into a stack is the first one to be deleted from it.

Stack can be visualized by



Section 4.2 Array representation of stack

Stack will be maintained by a linear array **STACK;** a pointer variable **TOP,** which contains the location of the top element of the stack and a variable **MAXSTK** which gives the maximum number of elements that can be held by the stack. The condition
TOP = 0 or TOP = NULL will indicate the stack is empty.



AAA	BBB	CCC	DDD						
1	2	3	4	5	6	7	8	9	10

TOP[4] MAXSTK [4]

The operation of adding (pushing) an item onto a stack and the operation of removing (popping) of an item from a stack may be implemented respectively by the procedure **PUSH() and POP()** algorithm. While executing PUSH operation, it is required to see whether space is there for the new data item. If no space then the condition of **“overflow”** arises.

Similarly for POP operation, one must first check whether there is element in the stack to be deleted. If no data item found in the stack then the condition is “**underflow”**.

Algorithm 1: PUSH (STACK, TOP, MAXSTK, ITEM) This procedure pushes an item on to a stack.

1. [Stack already filled]? If TOP = MAXSTK, then Print: OVERFLOW, and Return.
2. [Increase TOP by 1].
   Set TOP := TOP + 1.
3. . [Inserts ITEM in new TOP position].
   Set STACK [TOP] := ITEM
4. Return.

Algorithm 2: POP (STACK, TOP, ITEM)

This procedure deletes the TOP element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed] If TOP = 0, then Print: UNDERFLOW and Return.
2. [Assign TOP element to ITEM].
   Set ITEM :=STACK [TOP].
3. [Decrease TOP by 1].
   Set TOP := TOP – 1
4. Return.
   Exit

Section 4.3 Arithmetic Expressions;Polish Notation

An **expression** is a collection of operators and operands that represents a specific value. In above definition, operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc., Operands are the values on which the operators can perform the task.

An **arithmetic expression** is an **expression** built up using numbers, **arithmetic** operators (such as +,* , -, / and) and parentheses, "(" and ")". **Arithmetic expressions** may also make use of exponents.

An **expression** is a combination of one or more operands, zero or more **operators**, and zero or more pairs of parentheses. There are three **kinds** of **expressions**:

1. An **arithmetic expression** evaluates to a single **arithmetic** value.
2. A character **expression** evaluates to a single value of **type** character.

There are three kinds of expressions:

- An *arithmetic expression* evaluates to a single arithmetic value.
- A *character expression* evaluates to a single value of type character. A *logical* or *relational expression* evaluates to a single logical value.

The *operators* indicate what action or operation to perform.

The *operands* indicate what items to apply the action to. An operand can be any of the following kinds of data items:

- Constant
- Variable
- Array element
- Function
- Substring
- Structured record field (if it evaluates to a scalar data item) An expression

Table: Arithmetic Operation

Operator	Meaning

**	Exponentiation
*	Multiplication
/	Division
-	Subtraction or Unary Minus
+	Addition or Unary Plus

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

 Infix Notation Prefix (Polish) Notation Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. **a - b + c,** where operators are used **in**between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗

3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Conversion of Infix Expressions to Prefix and Postfix

The first technique that we will consider uses the notion of a fully parenthesized expression. Recall that **A + B * C** can be written as **(A + (B * C))** to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Look at the right parenthesis in the subexpression (B * C) above. If we were to move the multiplication symbol to that position and remove the matching left parenthesis, giving us B C *, we would in effect have converted the subexpression to postfix notation. If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result (see figure).

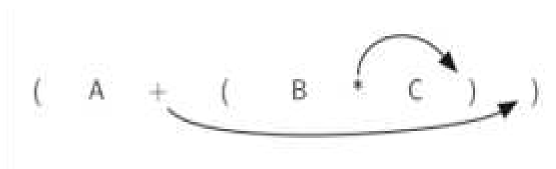


Figure : Moving Operators to the Right for Postfix Notation

Infix expression: A+B*C **Postfix expression: ABC*+**

If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation (see figure). The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.

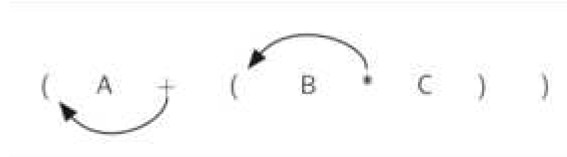


Figure 7: Moving Operators to the Left for Prefix Notation

Infix expression: A+B*C **Prefix expression: +A*BC**

Example:

(A+B)/(C-D) = [+AB] / [-CD] = /+AB-CD (prefix)

(A+B)/(C-D) = [AB+] / [CD-] = AB+CD-/ (postfix)

Evaluation of a Postfix Expression

Suppose P is an arithmetic expression in postfix notation. The following algorithm which uses a STACK to hold operands , evaluates P.

Algorithm: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

- Step 1. Add a right parenthesis ")" at the end of P.
- Step 2. Scan P from left to right and repeat **steps 3 and 4 for each** element of P until the symbol ")" is encountered.
- Step 3. If an operand is encountered put it on STACK.
- Step 4. If an operator $\otimes$ is encountered then:

  (a) Remove the two top elements of STACK where A is the top element and B is the next-to-top element. (b) Evaluate B $\otimes$ A.

  (c) Place the result of (b) back to STACK.

  [End of IF structure]

  [End of Step 2 loop]

- Step 5. Set VALUE equal to the top element on STACK. Step 6. Exit.

Example: Consider the following arithmetic expression Q in infix notation
Q : 5*(6+2)-12/4. Let us convert it into postfix form 'P" and then evaluate it by the algorithm.

'P' written in postfix notation. **P: 5, 6, 2, + , *, 12, 4, / ,- ,)**

While evaluating, the following shows the stack contents

	Symbol scanned	STACK
1	5	5
2	6	5,6
3	2	5,6,2
4	+	5,8
5	*	40
6	12	40,12
7	4	40,12,4
8	/	40,3
9	-	37
10	)	
The final VALUE in stack is 37.		

Section 4.4 Application of Stack

The followings are the application of stack

1. **Expression Evaluation and their Conversion**
2. **Recursion and Function Call**
3. Backtracking
4. Parenthesis Checking
5. Memory Management

RECURSION

1. "Recursion" is technique of solving any problem by calling same function again and again until some breaking (base) condition where recursion stops and it

starts calculating the solution from there on. For eg. calculating factorial of a given number

2. Suppose ‘P’ is a procedure containing either a **Call statement to itself** or **a Call statement to a second procedure** that may eventually result in a Call statement back to the original procedure ‘P’. Then ‘P’ is called a Recursive procedure. So that the program **will not continue to run indefinitely**, a recursive procedure must have the following two properties:

    - There must be a certain criterion called base criteria for which the procedure does not call itself.
    - Each time the procedure call itself (directly or indirectly), it must be closure to the base criteria.

The recursive procedure with these two properties is said to be well-defined

3. Thus, in recursion last function called needs to be completed first.
4. Now Stack is a LIFO data structure i.e. (Last In First Out) and hence it is used to implement recursion.
5. The High-level Programming languages, such as Pascal, C etc. that provides support for recursion use stack.
6. In each recursive call, there is need to save the
    1. current values of parameters,
    2. local variables and
    3. the return address (the address where the control has to return from the call).
7. Also, as a function calls to another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.

Example:

Let us calculate factorial 4 using **non recursive** definition

Procedure FACTORIAL (FACT, N)

Step1.If N=0

Then FACT=1 and Return.

Step2. Set FACT=1.

Step3. Repeat for K = 1 to N

Set FACT := K*FACT.

Step4. Return.

Let us calculate factorial 4 using **recursive** definition

Procedure FACTORIAL (FACT, N)

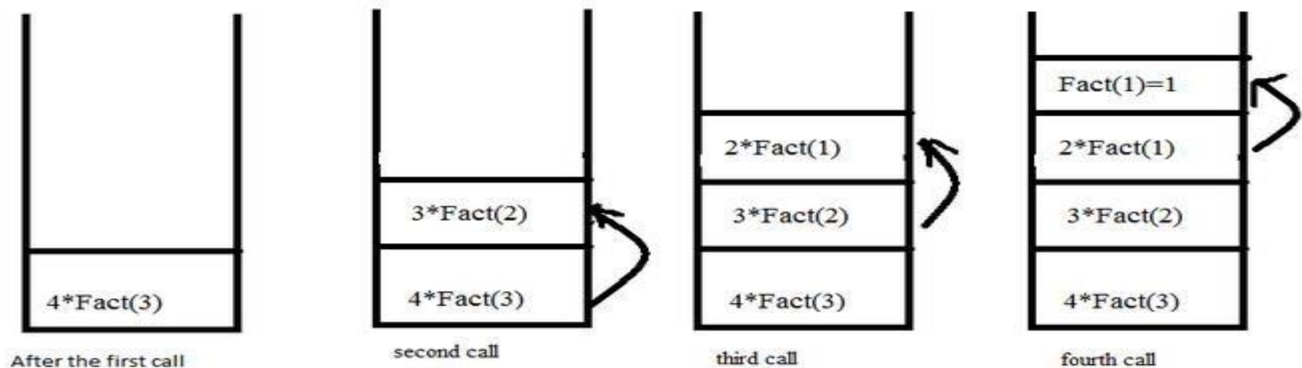
Step1.If N=0

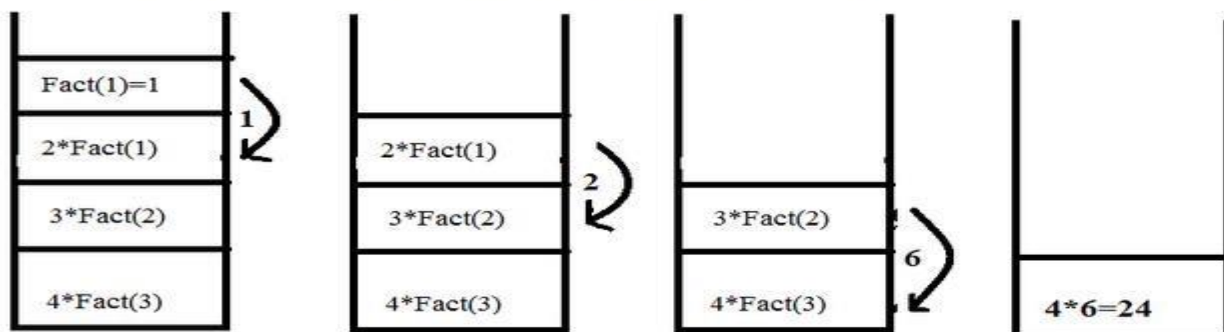
Then FACT=1 and Return.

Step2. Call FACTORIAL (FACT, N-1).

Step3. Set FACT := N * FACT.

Step4. Return.

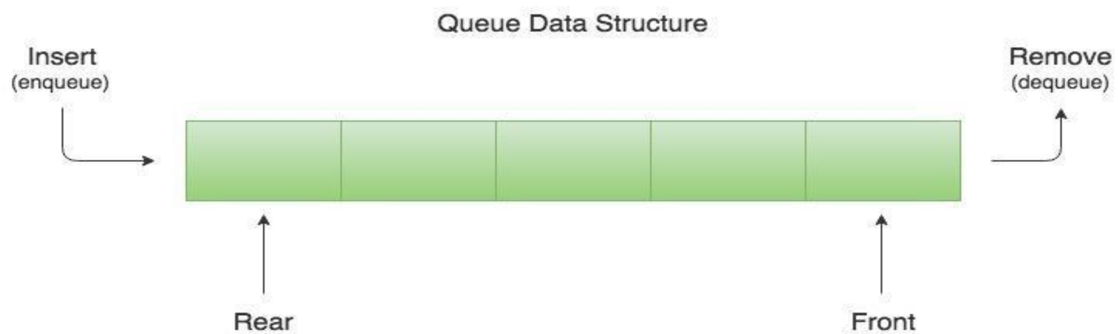




Section 4.5 QUEUES

A queue is a linear list of elements in which deletions can take place only at one end called the **front** and insertions can take place only at the other end called the **rear** end. Queues are also called First in First out (FIFO) lists, since the first element in the queue will be the first element out of the queue. In other words, the order in which elements enter in a queue is the order in which they leave.

Example: Automobiles waiting to pass through an intersection form a queue, i.e in which the first car in line is the first car through; People waiting in a line at the bank form a queue.



Representation of queue

Queues may be represented by means of

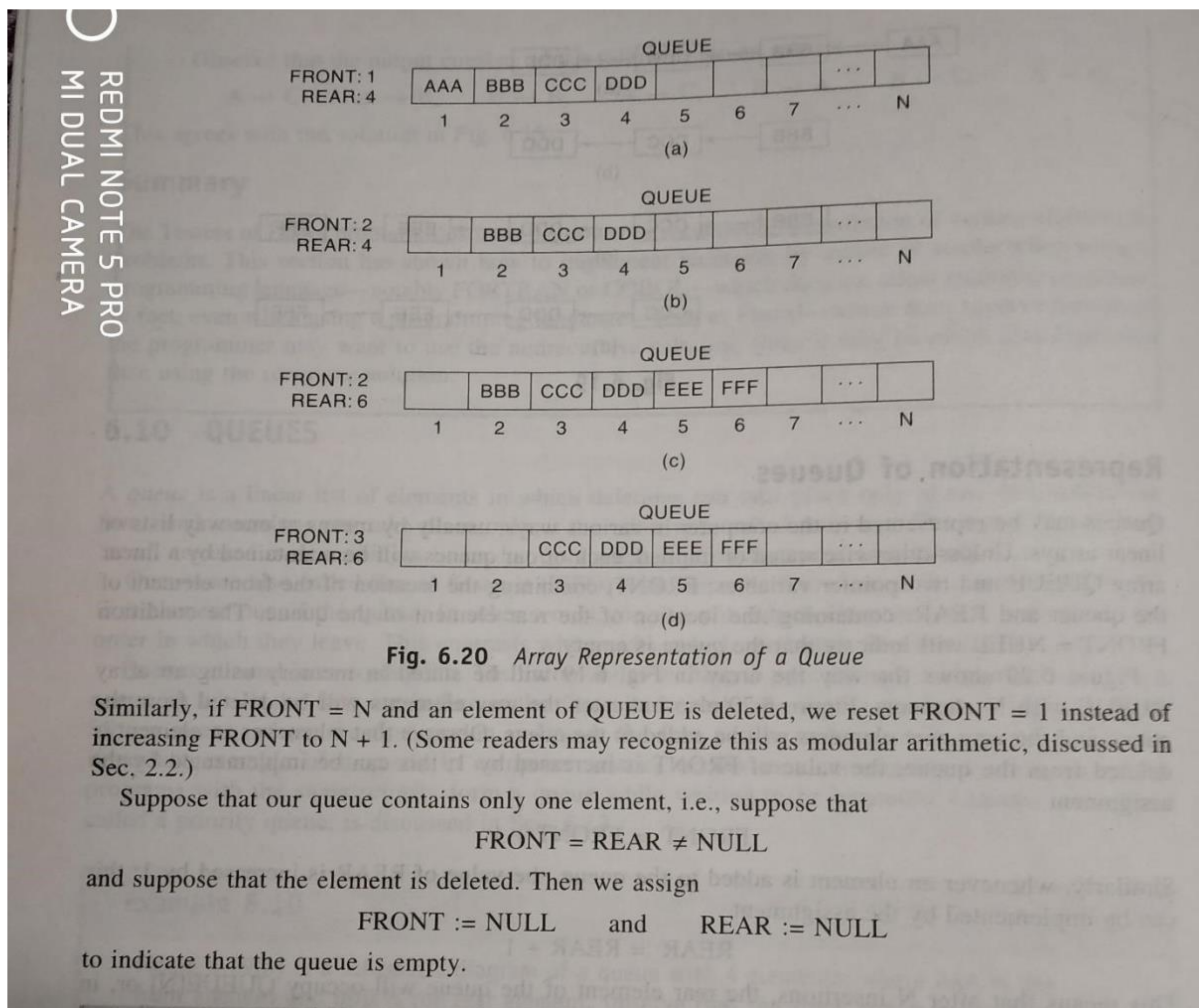
(a) Array representation

(a) Linked list representation

Queues will be maintained by a linear array QUEUE and two pointer variables:

1. FRONT: containing the location of the front element of the queue
2. REAR: containing the location of the rear element of the queue

The condition FRONT = NULL will indicate that the queue is empty. Following figure shows the way the array will be stored in memory using an array QUEUE with N elements. It also indicates the way elements will be deleted from the queue and the way new elements will be added to the queue.



REDMI NOTE 5 PRO
MI DUAL CAMERA

Fig. 6.20 *Array Representation of a Queue*

Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1. (Some readers may recognize this as modular arithmetic, discussed in Sec. 2.2.)

Suppose that our queue contains only one element, i.e., suppose that

$$FRONT = REAR \neq NULL$$

and suppose that the element is deleted. Then we assign

$$FRONT := NULL \quad \text{and} \quad REAR := NULL$$

to indicate that the queue is empty.

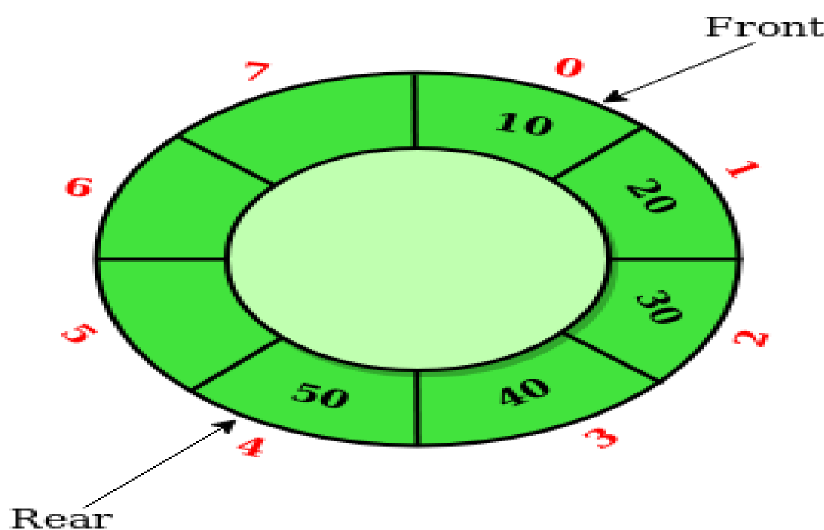
Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment:

FRONT: = FRONT + 1

Similarly, whenever an new element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment:

REAR: = REAR +1

After N insertions the rear pointer will be incremented and the QUEUE will occupy all N locations i.e QUEUE[N].

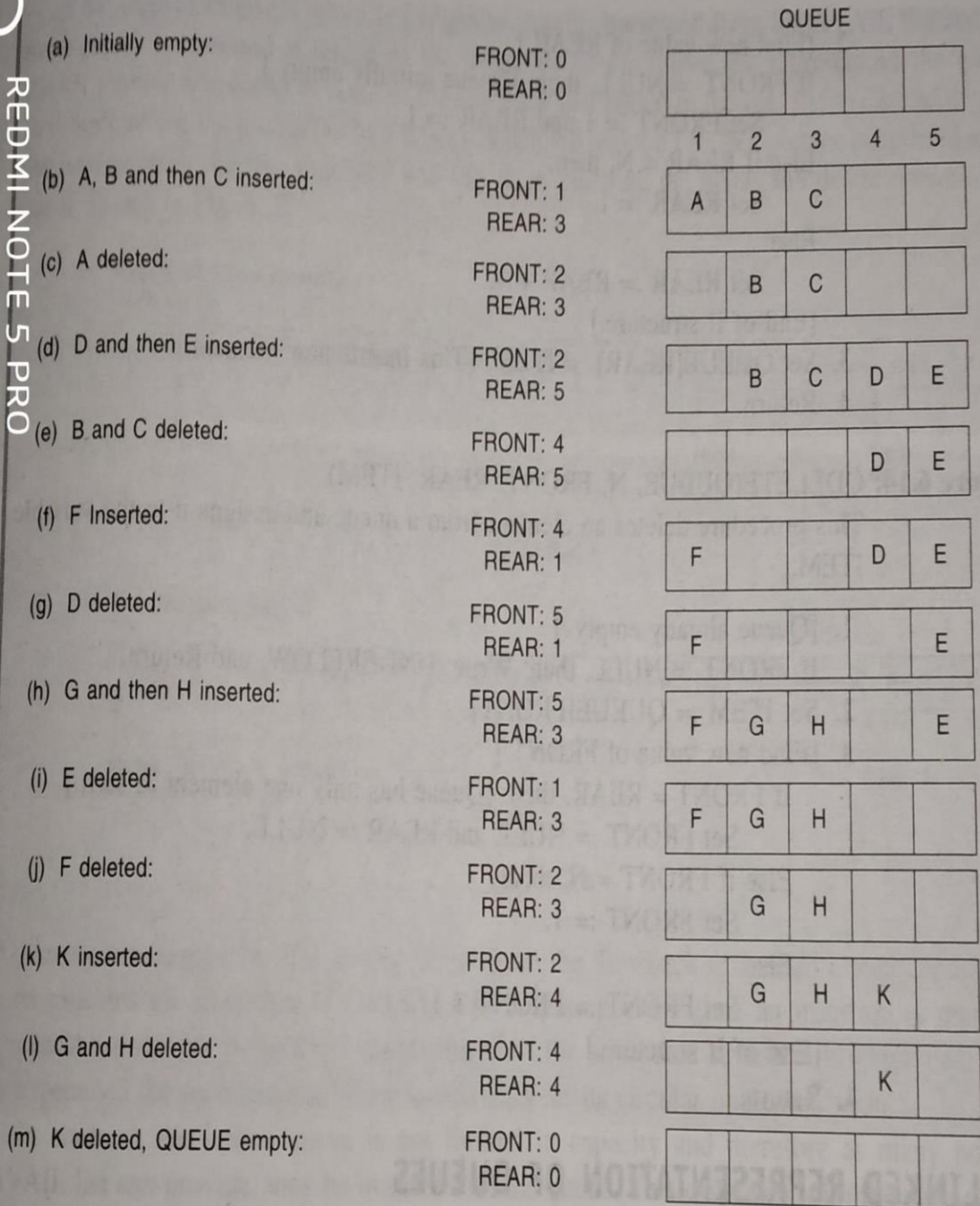


An ITEM is to be inserted and REAR = N we should not think that it is full, rather we simply move the entire queue to the beginning of the array changing FRONT and REAR pointer accordingly.

We assume QUEUE is circular i.e QUEUE[1] comes after QUEUE[N] in array. Specifically instead of REAR to N+1, we reset REAR = 1 and QUEUE[REAR] := ITEM

Similarly if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1. For a special case when FRONT= REAR ≠ NULL and suppose that element is deleted, then we assign

FRONT :=NULL and REAR := NULL and indicate that Queue is empty. For example let us have a Queue with N = 5 memory locations and several operations are observed.



REDMI NOTE 5 PRO
MI DUAL CAMERA

Fig. 6.21

We are now prepared to formally state our procedure QINSERT (Procedure 6.13), which inserts a data ITEM into a queue. The first thing we do in the procedure is to test for overflow, that is, to test whether or not the queue is filled.

Next we give a procedure QDELETE (Procedure 6.14), which deletes the first element from a queue, assigning it to the variable ITEM. The first thing we do is to test for underflow, i.e., to test whether or not the queue is empty.

Algorithm: QINSERT(QUEUE,N,FRONT,REAR,ITEM)

This procedure inserts an element ITEM into the queue.

Step1.[Check Queue already full?]

If FRONT = 1 and REAR = N

Or if FRONT = REAR + 1

Then write “OVERFLOW” and return

Step 2. [Find the new value of REAR] If

FRONT = NULL [for empty queue] Then

Set FRONT :=1 and REAR :=1.

Else if REAR = N

Then Set REAR :=1

Else Set REAR := REAR + 1

Step 3. [Inserts the new element]

Set QUEUE[REAR] := ITEM.

Step 4. Return.

Algorithm: QDELETE(QUEUE,N,FRONT,REAR,ITEM)

This procedure delets an element from the queue and assigns it to the variable ITEM..

Step1.[Check Queue already empty?]

If FRONT = NULL

Then write “UNDERFLOW” and return

Step 2. [Take out the element to be deleted] Set ITEM

:= QUEUE[FRONT]. Step 3. [Find the new value of

FRONT] if FRONT = REAR [Queue has only one

element] then Set FRONT = NULL and REAR =

NULL.

Else If FRONT = N

Then Set FRONT := 1.

Else Set FRONT := FRONT + 1.

Step 4. Return.

Section 4.5 Priority Queue

In computer science, a **priority queue** is an abstract data type similar to regular **queue** or stack data structure in which each element additionally has a "**priority**" associated with it. In a **priority queue**, an element with high **priority** is served before an element with low **priority**. The elements are deleted and processed comes from the following rules: 1. An element with high priority is processed before an element with low priority.

2. If two elements have the same priority, they are processed according to the order in which they were added to the queue, while in other implementations, ordering of elements with the same priority is undefined.

Priority Queue is represented and maintained in two different ways:

(a) One-way List representation of a priority queue

(b) Array representation of a priority queue

(A) One-way List representation of a priority queue in memory is made as follows;

- Each node in the list will contain three items of information
  - An information field INFO
  - A Priority number PRN
  - Link number LINK
- A node X precedes a node Y in the list
  - When X has higher priority than Y or
  - When both have the same priority but X was added to the list before Y i.e the order in the one way list corresponds to the order of the priority queue.

Addition and Deletion operations in a priority queue is more complicated.

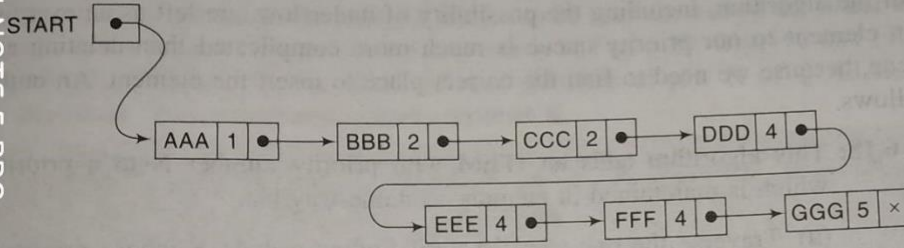


Fig. 6.27

START 5

AVAIL 2

	INFO	PRN	LINK
1	BBB	2	6
2			7
3	DDD	4	4
4	EEE	4	9
5	AAA	1	1
6	CCC	2	3
7			10
8	GGG	5	0
9	FFF	4	8
10			11
11			12
12			0

Fig. 6.28

The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue. The outline of the algorithm follows.

(B)Array Representation of Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority. Each such Queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. If each Queue is allocated the same amount of space, a twodimensional array QUEUE can be used instead of the linear arrays.

Front[K] and REAR[K] contain respectively the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

Example 6.14

Consider the priority queue in Fig. 6.27. Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers. Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig. 6.29. Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the list. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

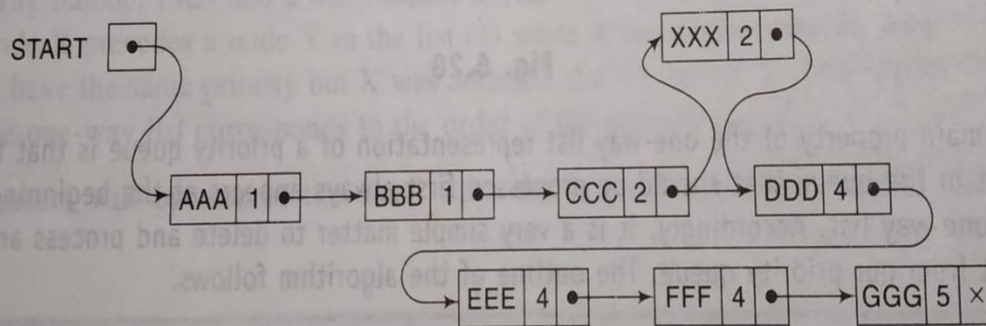


Fig. 6.29

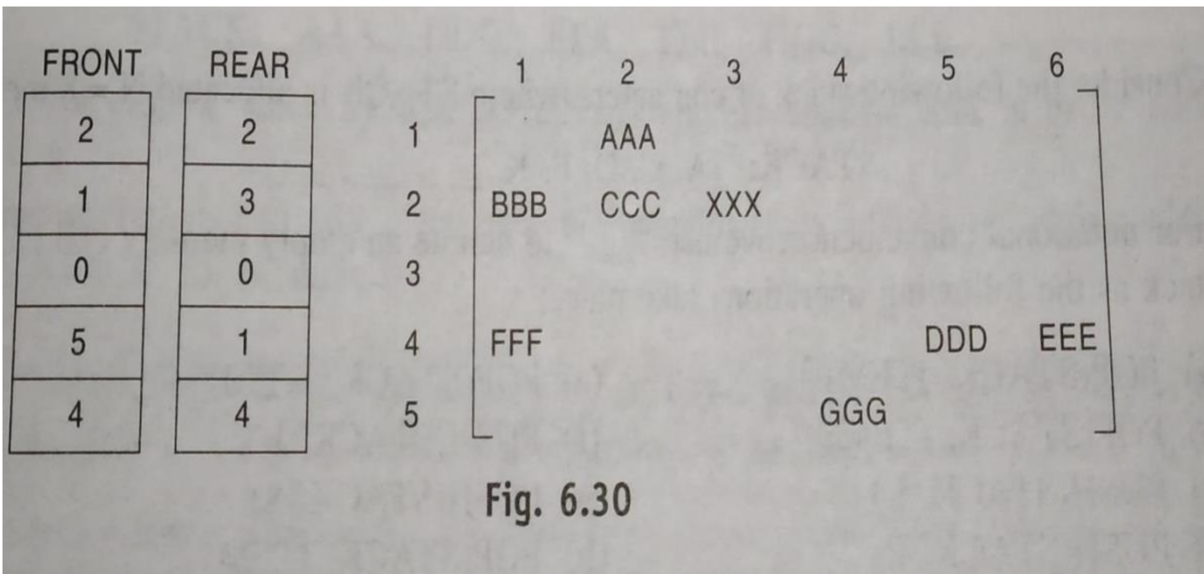


Fig. 6.30

Now we can compare stack and queue.

#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.

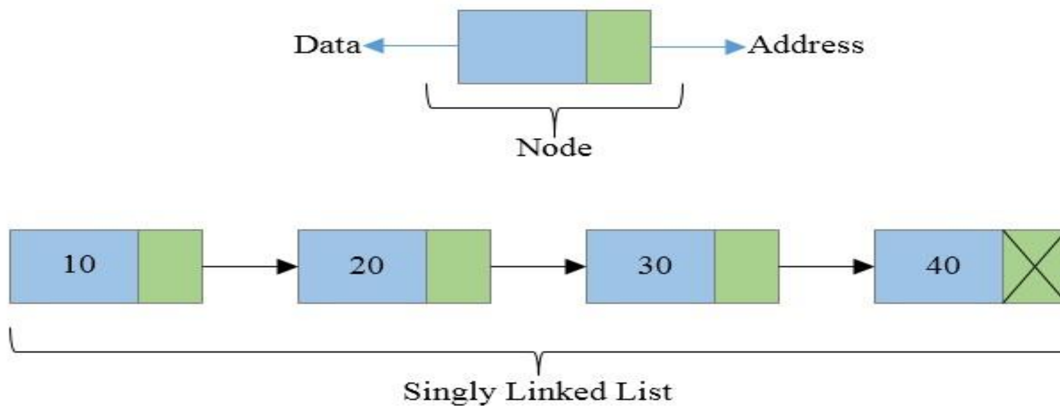
CHAPTER 5.0 LINKED LIST

Section 5.1 Introduction:

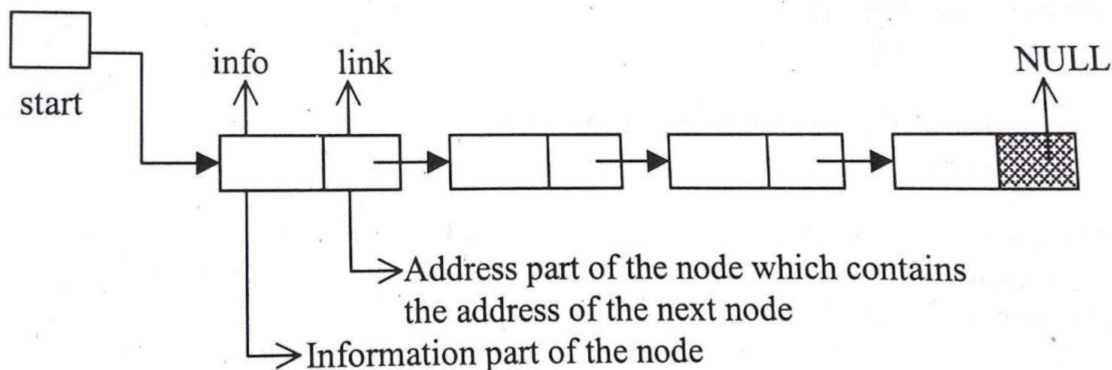
A **linked list** is a linear **data structure**, in which the elements are not stored at contiguous memory locations. A linked list or one-way list is a dynamic data structure consists of **data elements** (called a **nodes**) where the linear order is given by means of pointers. Each node is made up of two items:

1. Information of the element or data, and

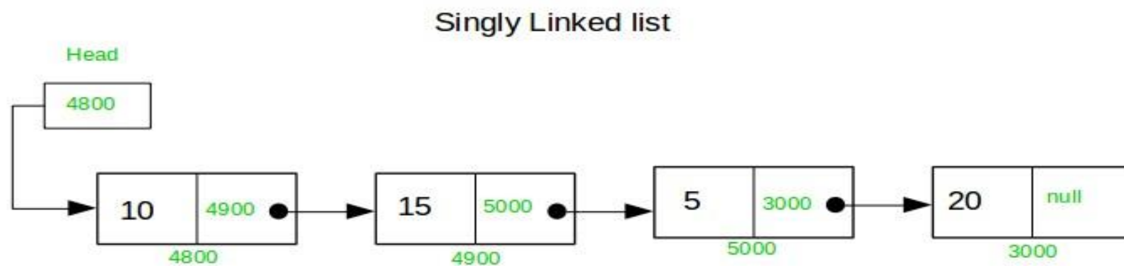
2. A reference/ link (or next pointer), which points to the address of the next **node**.



A linked list is a collection of **nodes** where each **node** is connected to the next **node** through a pointer.A linked list is a sequence of data structures, which are connected together via links



The data structure studied i.e Array, stack and queue where (A) Insert and Delete operation is not done as per convenience or as required by user. It has to follow either FIFO or FILO scheduling and (B) forecast of memory requirement before writing a program is not easy. It may be excess memory and get wasted or shortage of memory, for which a new data structure is evolved called Linked list.



Section 5.2 Representation of linked list in memory:

Let LIST be a linked list maintained in memory. List requires two linear arrays:

1.INFO

2.LINK

Such that **INFO[K]** contains information/data part and **LINK[K]** contains next pointer field of a node of LIST.

LIST requires a variable name such as:

START contains the beginning of the list and

NULL indicates the end of the list. We choose NULL=0.

The linked list indicates that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK, and that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.

Example 5.2

Figure 5.4 pictures a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters, or, in other words, the string, as follows:

START = 9, so INFO[9] = N is the first character.
LINK[9] = 3, so INFO[3] = O is the second character.
LINK[3] = 6, so INFO[6] = □ (blank) is the third character.
LINK[6] = 11, so INFO[11] = E is the fourth character.
LINK[11] = 7, so INFO[7] = X is the fifth character.
LINK[7] = 10, so INFO[10] = I is the sixth character.
LINK[10] = 4, so INFO[4] = T is the seventh character.
LINK[4] = 0, the NULL value, so the list has ended.

In other words, NO EXIT is the character string.

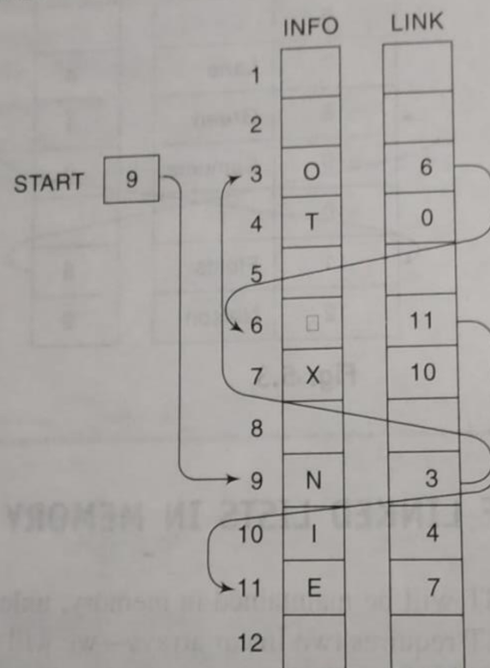


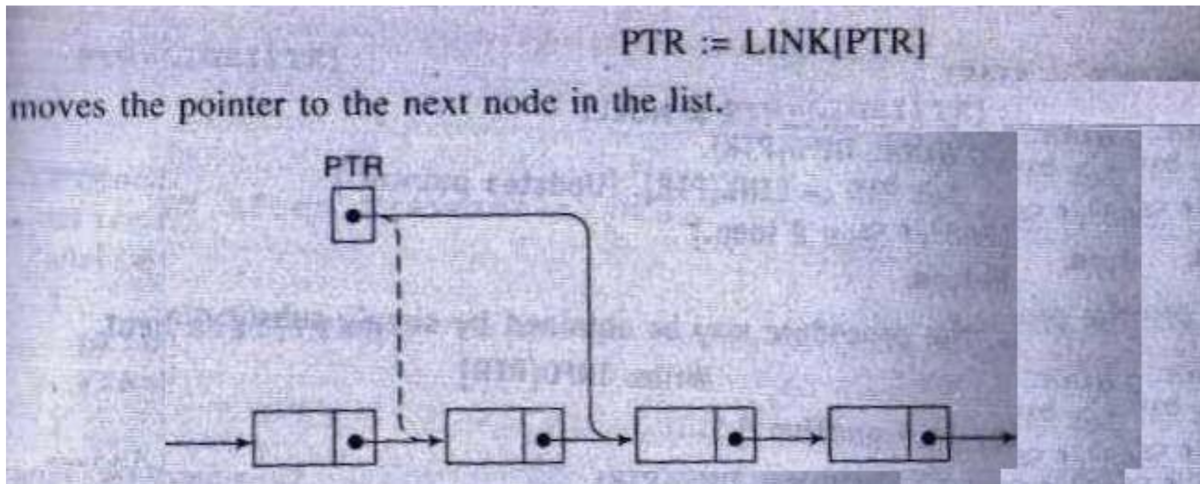
Fig. 5.4

Section 5.3 Traversing and Searching a linked list:

Let List be a linked list in memory stored in a linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of the list. Suppose we want to traverse LIST in order to process **each node exactly once.**

A pointer variable PTR which points to the node that is currently being processed. Accordingly, LINK[PTR] points to the next node to be processed. Thus, we have the assignment

PTR := LINK[PTR]



Algorithm (Traversing a linked list)

Let LIST be a linked list in memory, this algorithm traverse LIST applying & operation PROCESS to each element of LIST. The variable PTR to point to the nodes currently being processed.

Step 1 [initialize pointer PTR]

Set PTR :=START

Step 2 [Loop structure]

Repeat step 3 and 4 while PTR ≠ NULL

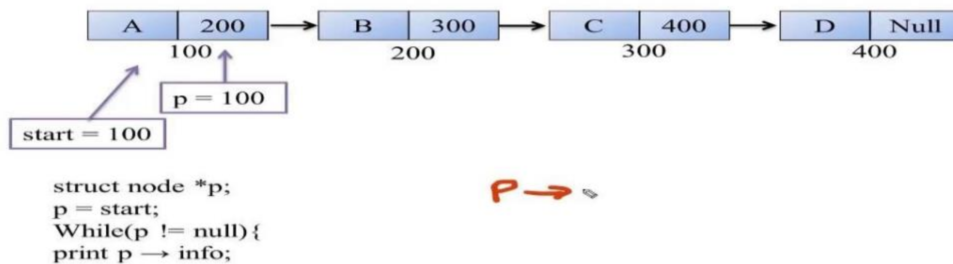
Step 3 [Visit the element]

Apply PROCESS to **INFO[PTR]**

Step 4 [PTR now points to the next node] SET
PTR=LINK [PTR]

End of step 2 loop

Step 5 Exit



Searching a linked list

Let LIST be linked list in memory. Suppose a specific ITEM of information is given. It is required to find the location LOC of the node where ITEM first appears in LIST.

Algorithm for searching linked list:-

SEARCH (INFO, LINK, START, ITEM, LOC)

LIST is a linked list in memory, this algorithm finds the location LOC of the node where ITEM first appears in LIST or sets, LOC=NULL.

Step 1 [initialize pointer PTR]

Set PTR :=START

Step 2 [Loop]

Repeat step 3 while PTR ≠ NULL

Step 3 [Compare and move forward till ITEM is found]

If ITEM = INFO[PTR] then Set LOC = PTR **&**

exit

Else

Set PTR = LINK[PTR] [PTR now points to next node]

[End of if structure & End of step 2 loop]

Step 4 [Search is unsuccessful]

Set LOC := NULL Step
5 Exit

Section 5.4 Memory allocation; Garbage Collection:

In computer science, **garbage collection** is a type of memory management. It automatically cleans up unused objects and pointers in memory, allowing the resources to be used again. Garbage collection may also be done at compile-time, when a program's source code is compiled into an executable program. In this method, the compiler determines which resources in memory will never be accessed after a certain time. It can then add instructions to automatically deallocate those resources from memory. While this is an effective way to eliminate unused objects,The **purpose** of **garbage collection** is to identify and discard those objects that are no longer needed by the application, in order for the resources to be reclaimed and reused.

Garbage collection relieves the programmer from performing manual memory management where the programmer specifies what objects to deallocate and return to the memory system and when to do so. Other similar techniques include stack allocation, region inference, memory ownership, and combinations of multiple techniques. Garbage collection may take a significant proportion of total processing time in a program and, as a result, can have significant influence on performance.

While doing operations in a linked list we have three important features:

(a) Memory Allocation (b) Garbage collection (c) Overflow and Underflow

(a) Memory Allocation

Whenever a new node is created, memory is allocated by the system. This memory is taken from list of those memory locations which are free i.e. not allocated. This list is called **AVAIL List, or list of available space, or free storage list** . Similarly, whenever a node is deleted, the deleted space becomes reusable and is added to the list of unused space i.e. to AVAIL List. This unused space can be used in future for memory allocation. The list has its own pointer ‘AVAIL’ and the data structure can be denoted as :

LIST(INFO, LINK,START,AVAIL)

Memory allocation is of two types-

1. Static Memory Allocation
2. Dynamic Memory Allocation

1. Static Memory Allocation:

When memory is allocated during compilation time, it is called ‘Static Memory Allocation’. This memory is fixed and cannot be increased or decreased after allocation. If more memory is allocated than requirement, then memory is wasted. If less memory is allocated than requirement, then program will not run successfully. So exact memory requirements must be known in advance.

2. Dynamic Memory Allocation:

When memory is allocated during run/execution time, it is called ‘Dynamic Memory Allocation’. This memory is not fixed and is allocated according to our requirements. Thus in it there is no wastage of memory. So there is no need to know exact memory requirements in advance.

(b) **Garbage Collection-**

Whenever a node is deleted, some memory space becomes reusable. This memory space should be available for future use. One way to do this is to immediately insert the free space into availability list. But this method may be time consuming for the operating system. So another method is used which is called ‘Garbage Collection’. This method is described below: In this method the OS collects the deleted space time to time onto the availability list. This process happens in two steps. In first step, the OS goes through all the lists and tags all those cells which are currently being used. In the second step, the OS goes through all the lists again and collects untagged space and adds this collected space to availability list. The garbage collection may occur when small amount of free space is left in the system or no free space is left in the system or when CPU is idle and has time to do the garbage collection.

(c) **Overflow & Underflow-**

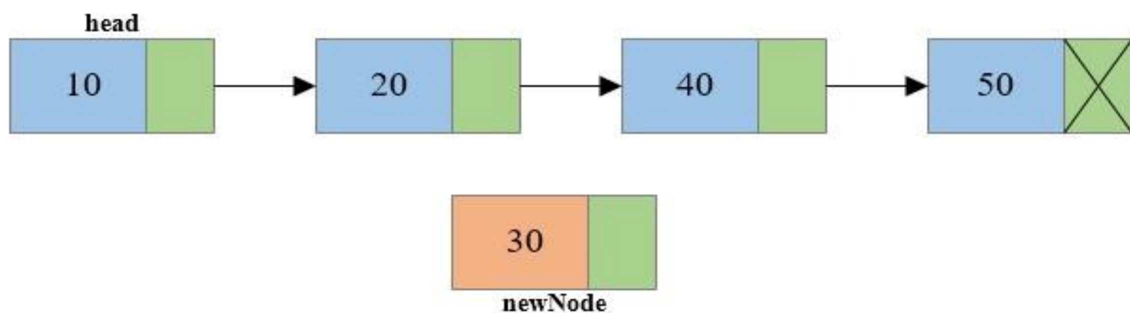
Overflow happens at the time of insertion. If we have to insert new space into the data structure, but there is no free space i.e. availability list is empty, then

this situation is called ‘Overflow’. The programmer can handle this situation by printing the message of OVERFLOW.As we know overflow will occur with our linked lists when **AVAIL=NULL**, there is an insertion.

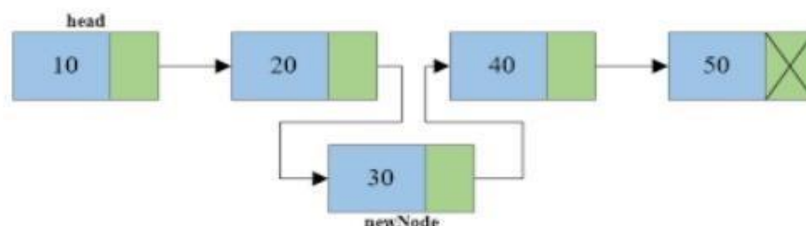
Underflow happens at the time of deletion. If we have to delete data from the data structure, but there is no data in the data structure i.e. data structure is empty, then this situation is called ‘Underflow’. The programmer can handle this situation by printing the message of UNDERFLOW. As we know underflow will occur with our linked lists when **START=NULL**, there is a deletion.

Section 5.5 Insertion into a linked list

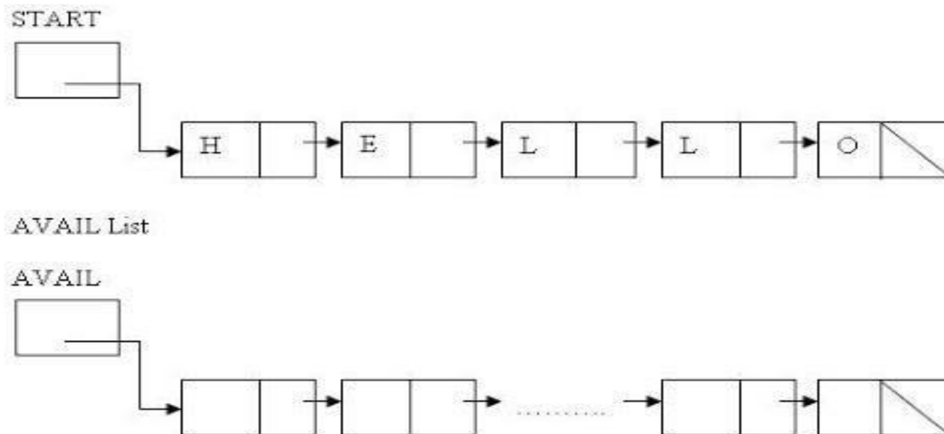
Let LIST be a linked list with successive nodes having values 20 and 40.We want to insert Node having value 30 in between them



Insertion in Singly Linked List



While inserting a node into a linked list, the new node will come from the AVAIL list. The first node of the AVAIL list will be used for the new node to be inserted.



Insertion operation can be performed at three situations

1.Insertion of a node at the beginning of the List.

2.Insertion of a node after the node with a given location of the List.

3. Insertion of a node into a sorted Linked List.

All the algorithms will use a node in the AVAIL list and include the following steps:

(a) Check to see if space is available in the AVAIL list and if not then OVERFLOW

If AVAIL = NULL

Then Print 'OVERFLOW'

(b) Remove a node from AVAIL list. Using a variable **NEW** to keep track of the location of the new node

NEW := AVAIL and AVAIL := LINK[AVAIL]

(c) Copying new information into the new node

INFO[NEW] := ITEM

(A) Inserting the node at the beginning of a list:-

INSERT (INFO, LINK, START, AVAIL, ITEM)

Step 1 [Check for over flow] If

AVAIL = NULL, then write

over flow & exit

Step 2 [REMOVE first node from AVAIL LIST]

Set NEW := AVAIL &

AVAIL := LINK [AVAIL]

Step 3 [Copy is new data into new node]

Set INFO [NEW] := ITEM

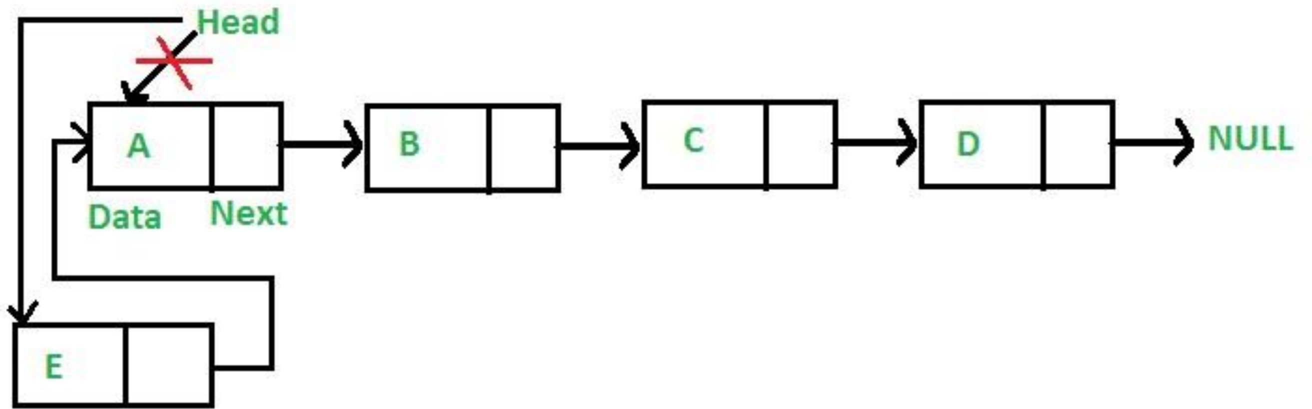
Step 4 [New node now points to original first node]

Set LINK [NEW] := START

Step 5 [Changes start so its point to new node]

Set START = NEW

Step 6 Exit


Head
A
B
C
D
NULL
Data
Next
E

(B)Inserting the node after a given node of a list: -

Suppose we are given the value of LOC where either LOC is the location of a node A in the linked list or LOC = NULL. The algorithm inserts ITEM into LIST so that ITEM follows node A or when LOC = NULL, so that ITEM is the first node.

Let N denote the new node whose location is NEW. If LOC = NULL, then N is inserted as the first node in the list otherwise node N points to node B (which originally followed node A) by the assignment: LINK[NEW] := LINK[LOC]

And we let node A point to the new node N by the assignment:

LINK[LOC] := NEW

Algorithm INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

This algorithm inserts ITEM. so that ITEM follows the node with location [LOC] or insert ITEM as the first node when LOC = NULL

Step 1 [Check over flow] if AVAIL = NULL, then write **overflow & exit**.

Step 2 [Remove first node from AVAIL list]

Set NEW := AVAIL & AVAIL := LINK [AVAIL]

Step 3 [Copy is new data into new node]

Set INFO [NEW] := ITEM

Step 4 [insert as first node] if LOC = NULL, then Set LINK [NEW] := START & START := NEW

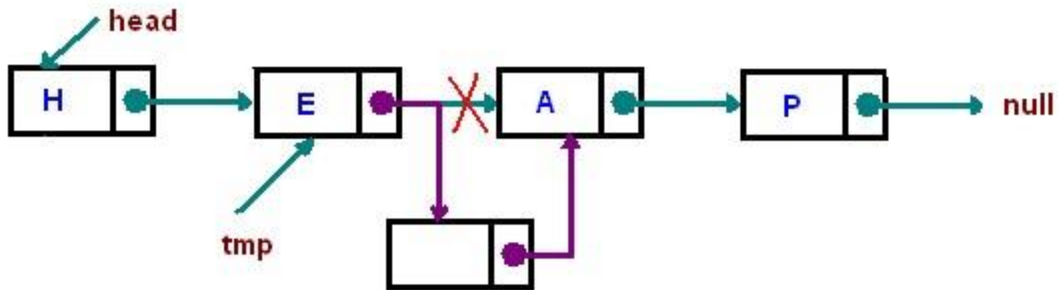
Else

[INSERT after node with location LOC]

Set LINK [NEW] := LINK [LOC] and LINK [LOC] := NEW

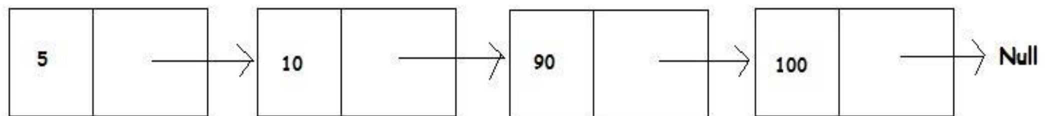
[End of if]

Step 5 Exit

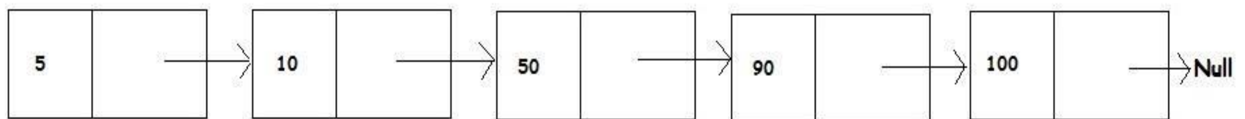


(C) Inserting the node into a Sorted linked list:-

The nodes having the INFO part are in ascending order. The node having INFO part as 50 is inserted in proper location



Insert 50 into the sorted Linked List



Insert into a sorted Linked List (OpenGenus)

Section 5.5 Deletion from a linked list:-

Let LIST be a linked list with a node N between nodes A and B. Suppose node N is to be deleted from the linked list. The linked list is maintained in memory in the form

LIST(INFO,LINK,START,AVAIL)

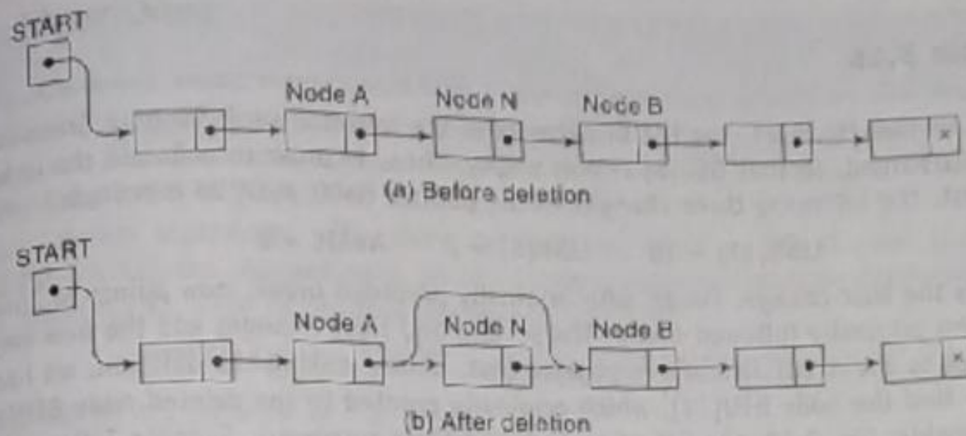


Fig. 5.22

Figure 5.22 does not take into account the fact that, when a node N is deleted from our list, we will immediately return its memory space to the AVAIL list. Specifically, for easier processing, it will be returned to the beginning of the AVAIL list. Thus a more exact schematic diagram of such a deletion is the one in Fig. 5.23. Observe that three pointer fields are changed as follows:

(1) The nextpointer field of node A now points to node B, where node N previously pointed.
(2) The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.
(3) AVAIL now points to the deleted node N.

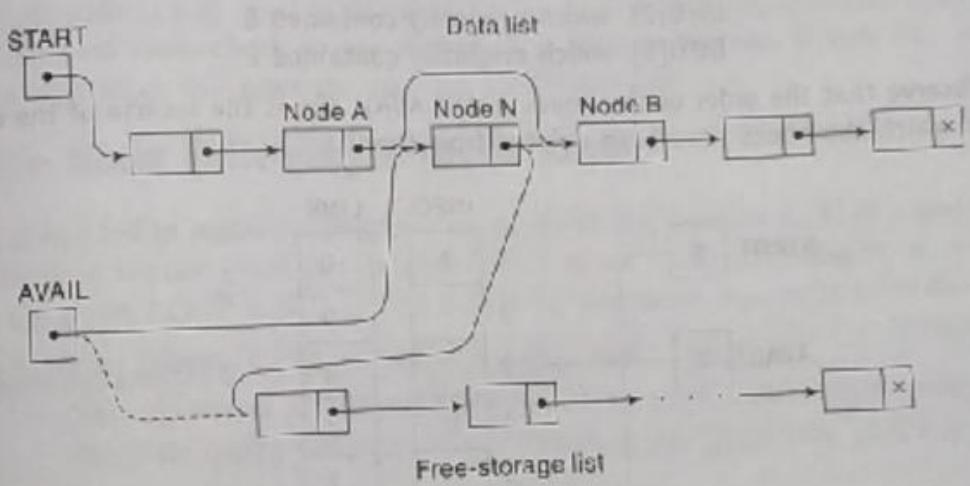


Fig. 5.23

There are also two special cases. If the deleted node N is the first node in the list, then START will point to node B; and if the deleted node N is the last node in the list, then node A will contain the NULL pointer.

Deletion Algorithms

Algorithms which delete nodes from linked lists come up in various situations. We discuss two of them here. The first one deletes the node following a given node, and the second one deletes the node with a given ITEM of information. All our algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START, AVAIL).

All of our deletion algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list. Accordingly, all of our algorithms will include the following pair of assignments, where LOC is the location of the deleted node N:

LINK[LOC] := AVAIL and then AVAIL := LOC

These two operations are pictured in Fig. 5.25.

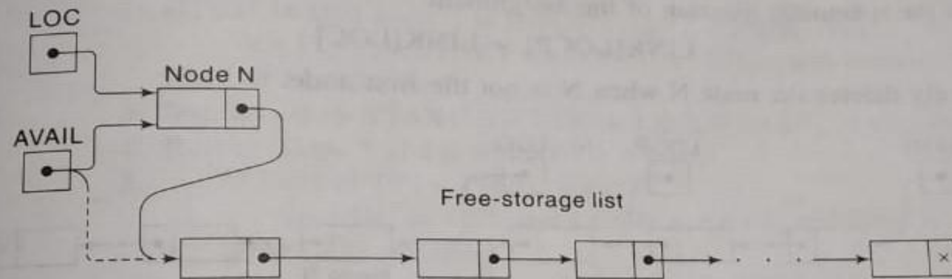


Fig. 5.25 *LINK[LOC] := AVAIL and AVAIL := LOC*

Some of our algorithms may want to delete either the first node or the last node from the list. An algorithm that does so must check to see if there is a node in the list. If not, i.e., if START = NULL, then the algorithm will print the message UNDERFLOW.

DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

This algorithm deletes the node N with location LOC, LOCP is the location of the node which precedes N or when N is the first node, LOCP = NULL.

Step 1 [Delete first node]
If LOCP = NULL,
then Set START := LINK [START]
Else
Set LINK [LOCP] := LINK [LOC]
[Deletes node N] [End of if]

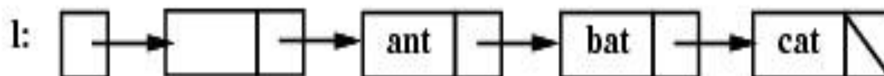
Step 2 [Return deleted node to the AVAIL LIST] Set
LINK [LOC] := AVAIL & AVAIL := LOC

Step 3 Exit

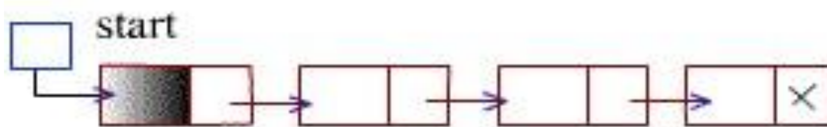
Section 5.5 Header linked list:-

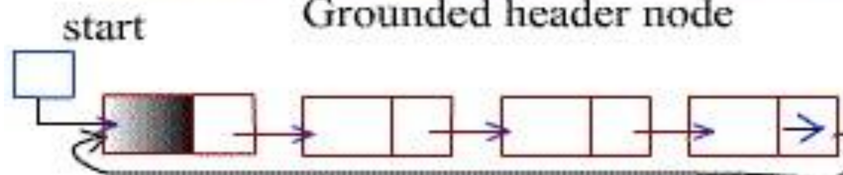
A header linked list is a special type of linked list, which always contains a special node called **header node** at the beginning of the list so in a header linked list will not point to first node of the list. But start will contain the address of the header node.

A **header** node is a special node that is found at the beginning of the **list**. A **list** that contains this type of node, is called the **header**-**linked list**. This type of **list** is useful when information other than that found in each node is needed.



There are two types of header linked list

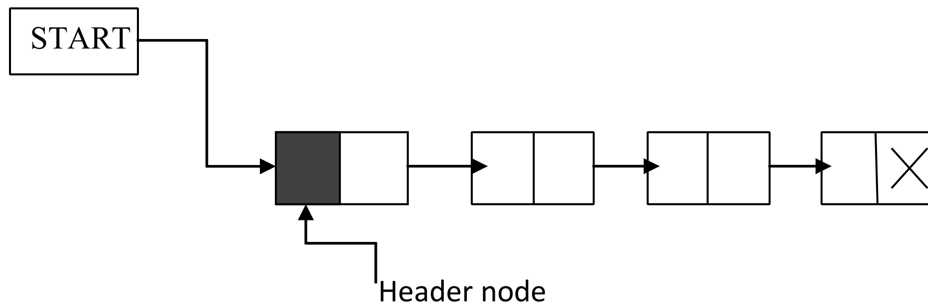




1.**Grounded header linked list**:-

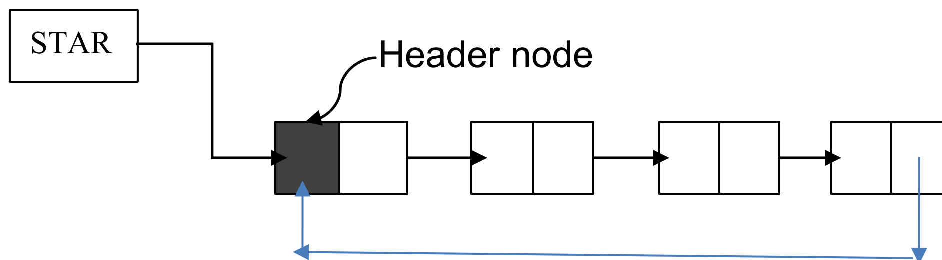
It is a header linked list where last node contains the NULL pointer. Observer that the list pointer START always points to the header node.

Then LINK [START] = NULL indicates that a grounded header linked list is empty.



2. Circular header linked list:-

It is a header linked list where the last node points back to the header node.



In Circular linked list when LINK [START] = START, it indicates that the circular linked list is empty.

Circular header list are frequently used instead of ordinary linked list because many operation are much easier to implement header list.

This comes from the following two properties, all circular header lists.

- The NULL pointer is not used & hence contains valid address.
- Every ordinary node has a predecessor. So the first node may not required a special case.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed. Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

74 75 76

Chapter 6.0 TREE

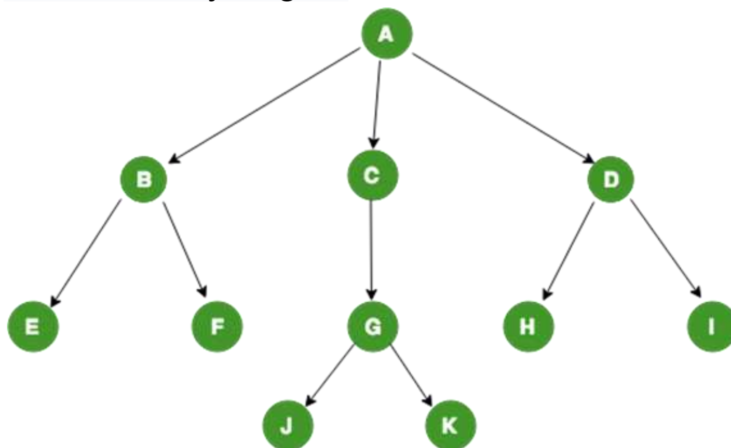
Section 6.1 Basic terminology

A **tree** is a nonlinear **data structure**, compared to arrays, linked lists, stacks and queues which are linear **data structures**. A **tree** can be empty with no nodes or a **tree** is a **structure** consisting of one node called the root and zero or one or more **subtrees.**

The **tree data structure** has roots, branches and leaves, but it is drawn upside-down. A **tree** is a hierarchical **data structure** which can represent relationships between different nodes.

An **example** of a **tree** is a genealogical diagram that shows the parents and offspring of many generations, the family **tree**.

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Tree Terminologies

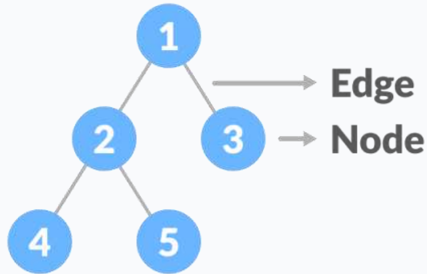
Node

A **node** is a structure which may contain a value or condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more **child nodes**, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the child's **parent node** (or superior). A node has at most one parent, but possibly many *ancestor nodes*, such as the parent's parent. Child nodes with the same parent are **sibling nodes**.

A node is an entity that contains a key or value and pointers to its child nodes. The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes. The node having at least a child node is called an **internal node**.

Edge

It is the link between any two nodes



.

Properties of a Tree

- A tree can contain no nodes or it can contain one special node called the **root** with zero or more subtrees.
- Every edge of the tree is directly or indirectly originated from the root.
- Every child has only one parent, but one parent can have many children.

Root

The topmost node in a tree is called the **root node**. Depending on definition, a tree may be required to have a root node (in which case all trees are non-empty), or may be allowed to be empty, in which case it does not necessarily have a root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin, since as a data structure, one can only pass from parents to children.

Path

A sequence of consecutive edges is called a path.

Height of a Node

The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

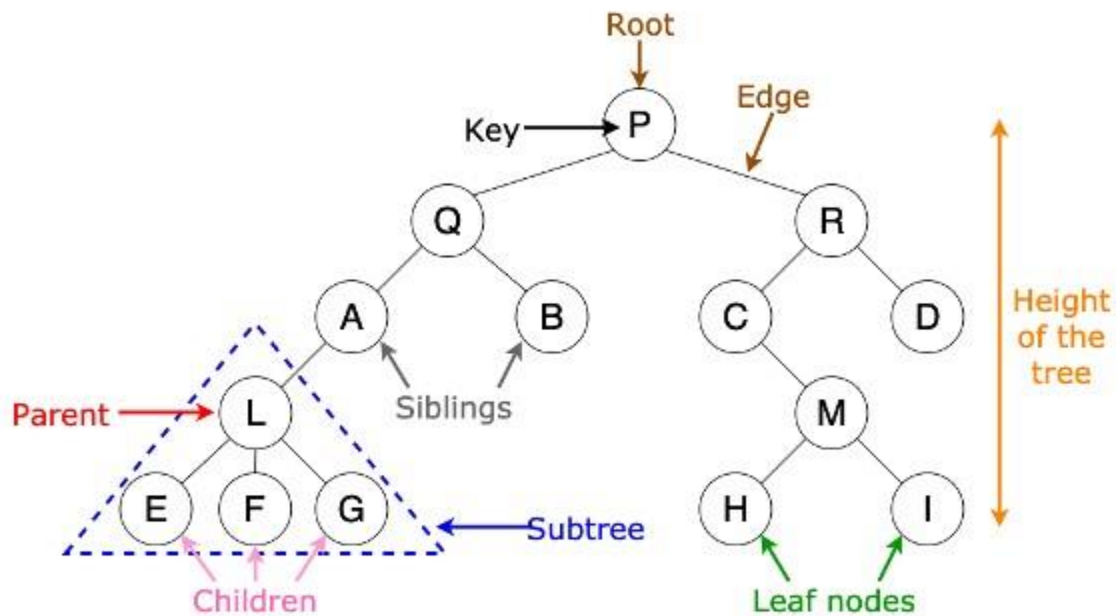
The height of a Tree is the height of the root node or the depth of the deepest node. It is one more than the largest level number of the tree

Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.



Types of tree

1. General tree
2. Binary tree
3. Binary search tree

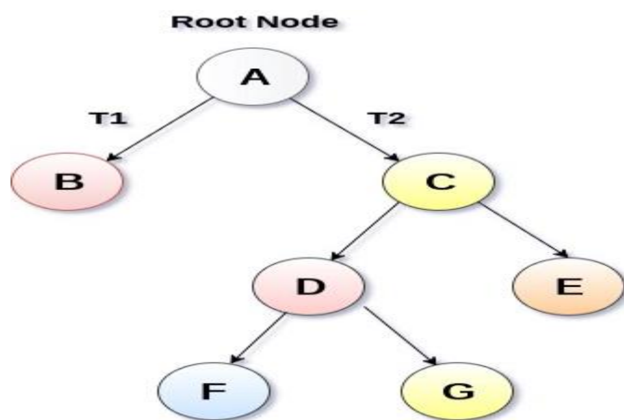
4. AVL tree
5. B-tree

Section 6.2 Binary tree

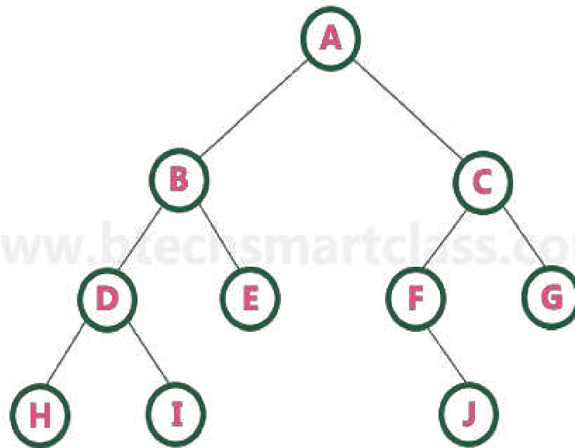
A Binary tree T is defined as a finite set of elements called nodes such that

(a) T is empty (called null tree or empty tree)

(b) T contains a distinguished node R called the root of T and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.



A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root. Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children.



Concepts of binary tree

- The binary tree T contains a root R. ‘A’ is the root T_1 and T_2 are called the left and right sub trees of R.
- If T_1 is non empty then its (T_1’s) root is called the left successor of R; so ‘B’ is the left successor of root A.
- Similarly If T_2 is non empty then its (T_2’s) root is called the right successor of R; so “C” is the right successor of root node A.
- The left sub tree of the root ‘A’ consists of the nodes B,D,E,H,I and the right sub tree of ‘A’ consists of the nodes C,F,G and J.
- The nodes with no successors are called **terminal or leaf nodes**. Nodes H, I, E, J and G are leaf nodes.
- Suppose N is a node in tree T with left successor S_1 and right successor S_2, and then N is the parent of S_1 and S_2. Further S_1 is called the left child of N and S_2 is the right child of N.
- S_1 and S_2 are said to be **siblings.** (B,C are siblings)
- Every node N in a binary tree T except the ROOT has a unique parent.
- The line drawn from a node N of tree T to a successor is called an **edge**.
- A sequence of consecutive edges is called a **path.**
- Each node in a binary tree T is assigned a **level number.** The root of the tree T is assigned the level number ‘0’ (zero). Every other node is assigned a level number which is ‘1’ more than the level number of its parent.

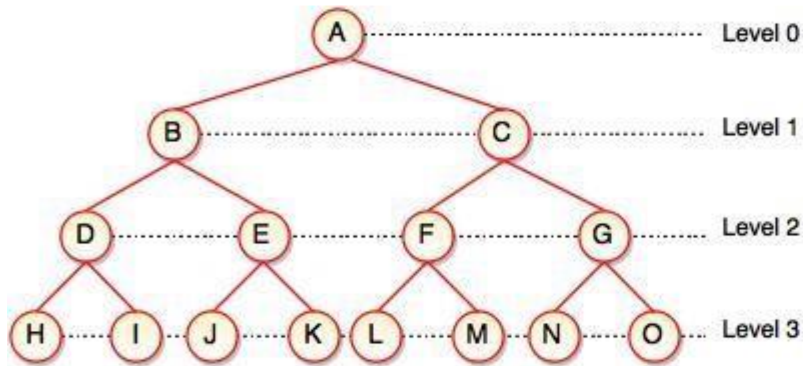


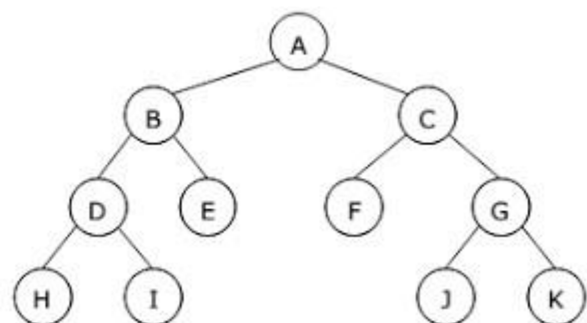
Fig. Complete Binary Tree

Her each node of can have at most two children. At level ‘r’ of T can have at most 2^r nodes

- Nodes with the same level number are said to belong to the same generation.
- The maximum **depth or height** is the number of nodes along the longest path from the root node down to the farthest leaf node. **It is one more than the largest level number of tree T**. It is the maximum number of **nodes in a branch** of tree T.

<u>Types of Binary Tree</u>

1. **<u>Full Binary Tree</u>** A Binary Tree is a full binary tree if every node has 0 or 2 children. The following is the example of a full binary tree.



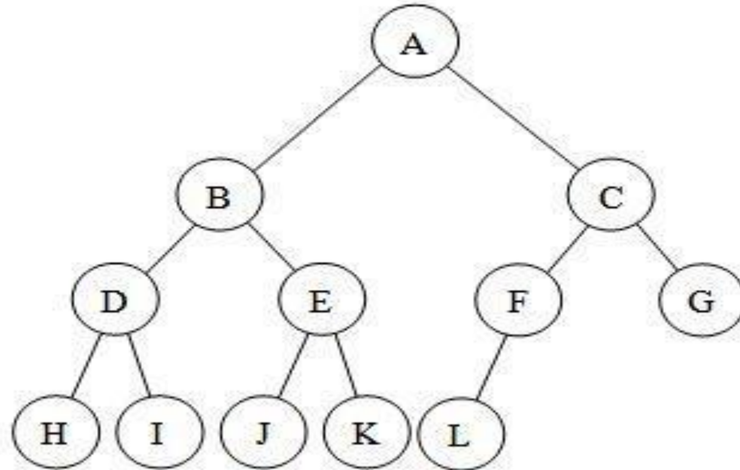
We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

In a Full Binary Tree, number of leaf nodes is the number of internal nodes plus 1

$L = I + 1$

Where L = Number of leaf nodes, I = Number of internal node.

2. **Complete Binary Tree:** A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level has all keys as left as possible.



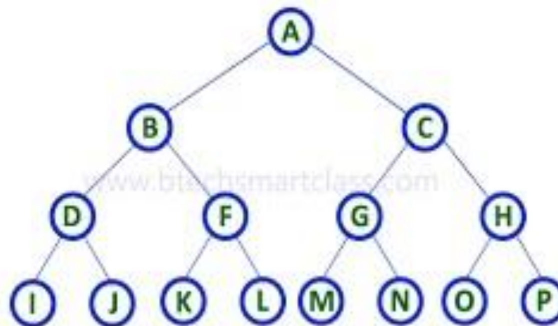
(complete Binary Tree)

The left and right child of node K are respectively 2*K and 2*K+1and the parent of K is the node [K/2].

3. **Perfect Binary Tree**

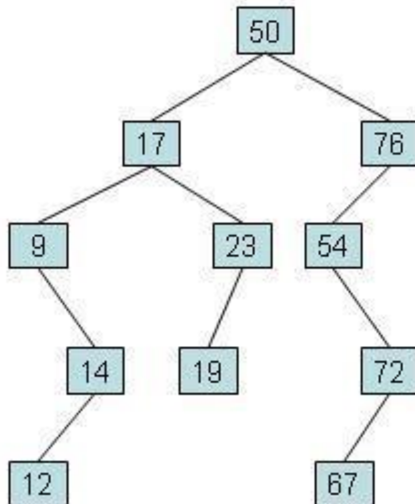
A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h – 1$ node.

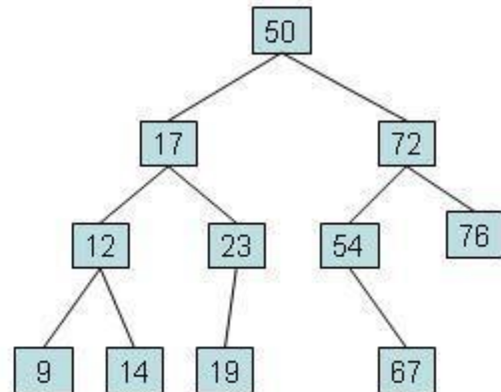


4. **Balanced Binary Tree**

A **balanced binary tree** is a **binary tree structure** in which the left and right sub trees of **every node** differ in height by no more than 1. One may also consider **binary trees** where no leaf is much farther away from the root than any other leaf. ... This means that the **tree** will behave like a linked list **data structure**.



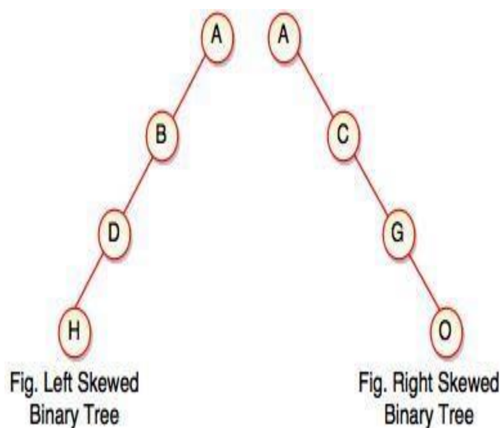
An unbalanced tree

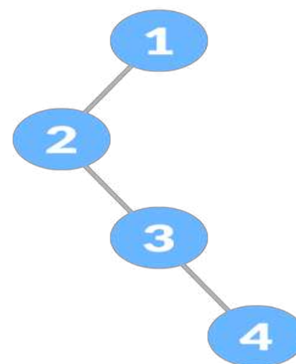


The same tree after being height-balanced

5. **A degenerate (or pathological) tree**

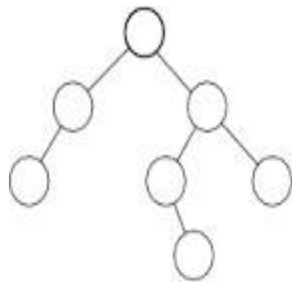
A Tree where every internal node has one child. A **degenerate tree** is a **tree** where for each parent node; there is only one associated child node. It is unbalanced and, in the worst case, performance degrades to that of a linked list. Such trees are performance-wise same as linked list.



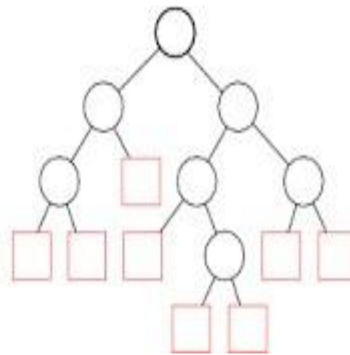


6. Extended Binary Trees:2-Trees

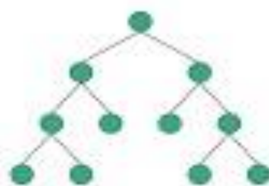
A binary tree 'T' is said to be a 2-tre or an extended binary tree if each node N has either 0 or 2 children. In such a case the nodes with 2 children are called **internal nodes**, and the nodes with '0' (zero) are called **external nodes.** A binary tree may be converted into a 2tree by replacing each empty sub tree by a new node. The new tree fig(b) is a 2-tree.Furthermore the nodes in the original tree fig(a) are now the internal nodes in the extended tree and the new nodes are the external nodes in the extended tree.

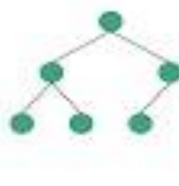


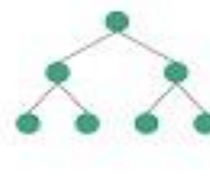
(a) Binary tree

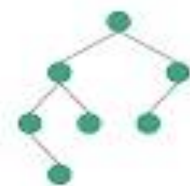


(b) Extended 2- tree









SECTION 6.2 REPRESNTING BINARY TREES IN MEMORY

Let 'T' be a binary tree. There are two ways of representing tree in memory

- Linked representation of binary tree
- Sequential of Array representation of binary tree

The main requirement of any representation of T is that one should have direct access to the root R of tree T and given any node N of T.

1.**Linked representation of binary tree**

Consider a binary tree T will be maintained in memory by means of a linked representation which uses three parallel arrays (a)INFO, (b) LEFT (c) RIGHT and a pointer variable ROOT such that

1. INFO[K] contains the data at the node N.
2. LEFT[K] contains the location of the left child of node N.
3. RIGHT[K] contains the location of the right child of node N.

Furthermore ROOT will contain the location of the root R of tree T.If any sub tree is empty, then the corresponding pointer will contain the null value, if the tree T itself is empty, and then ROOT will contain the null values.

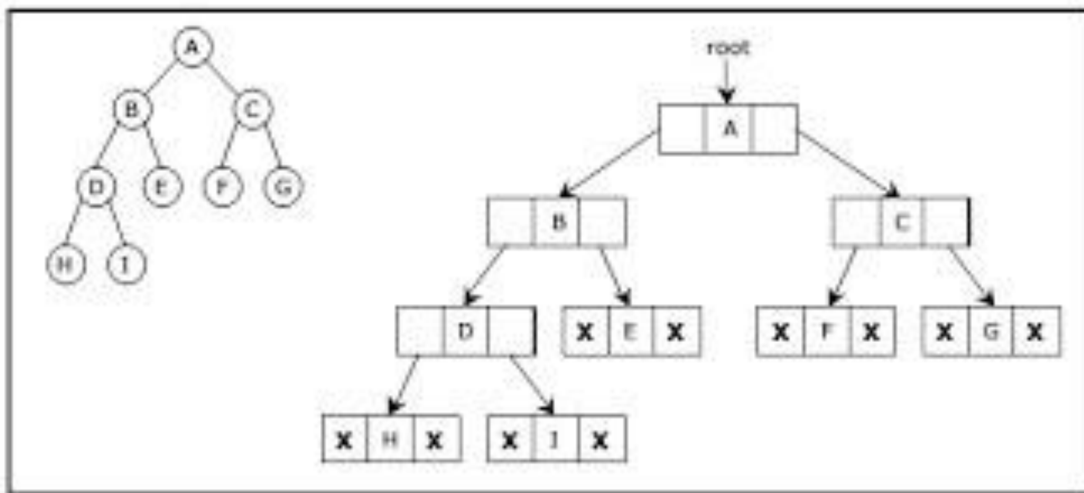


Figure 5.2.7. Linked representation for the binary tree

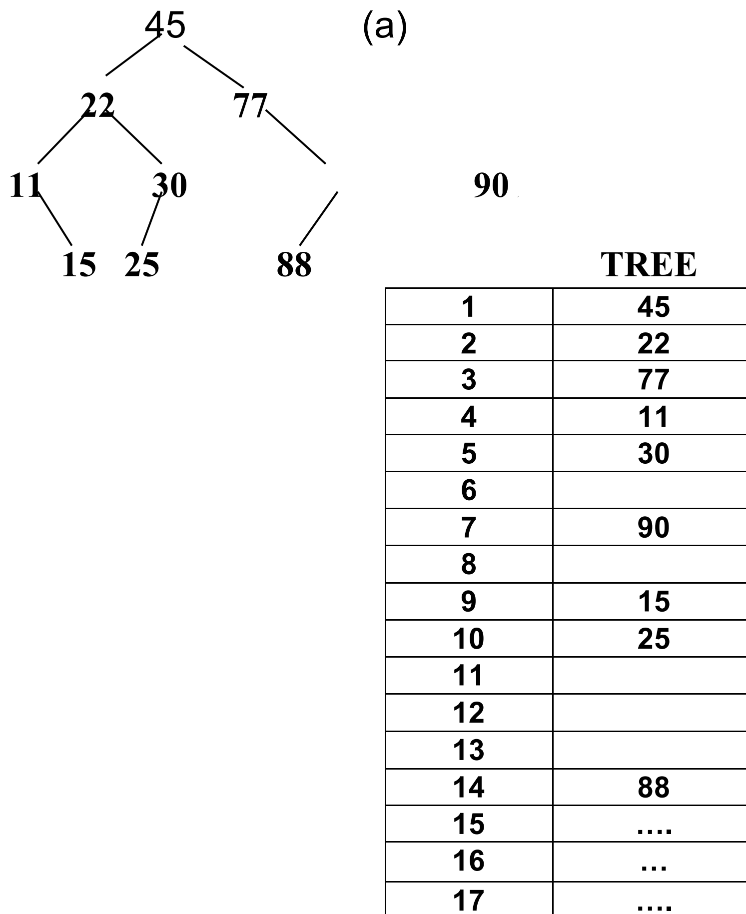
The linked representation of the tree may appear as shown below. ‘AVAIL’ list is maintained as a one way list using the array LEFT. The ROOT has initial value ‘5’ and ‘AVAIL’ list starts at 7.

		INFO	LEFT	RIGHT
	1	**F**	**0**	**0**
	2		**0**	
	3	**C**	**1**	**13**
	4	**I**	**0**	**0**
ROOT	5	**A**	**8**	**3**
	6	**D**	**12**	**4**
AVAIL	7		**9**	

	8	**B**	**6**	**10**
	9		**2**	
	10	**E**	**0**	**0**
	12	**H**	**0**	**0**
	13	**G**	**0**	**0**

2. Sequential of Array representation of binary tree

Suppose T is a complete binary tree. Then there is an efficient way of maintaining T in memory called sequential representation of T. This representation uses only a single linear array TREE as follows:



1	45
2	22
3	77
4	11
5	30
6	
7	90
8	
9	15
10	25
11	
12	
13	
14	88
15	….
16	…
17	….

(a) The root R of T is stored in TREE [1].

(b) If a node N occupies TREE[K], then its left child is stored in TRE[2*K] and its right child is stored in TRE[2*K+1]

NULL is used to indicate an empty subtree. A tree with depth ‘d’ will require an array with approximately 2^{d+1}.

TRAVERSING BINARY TREES

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root node. That is, we cannot randomly access a node in a tree. There are three standard ways of traversing a binary tree T' with root R

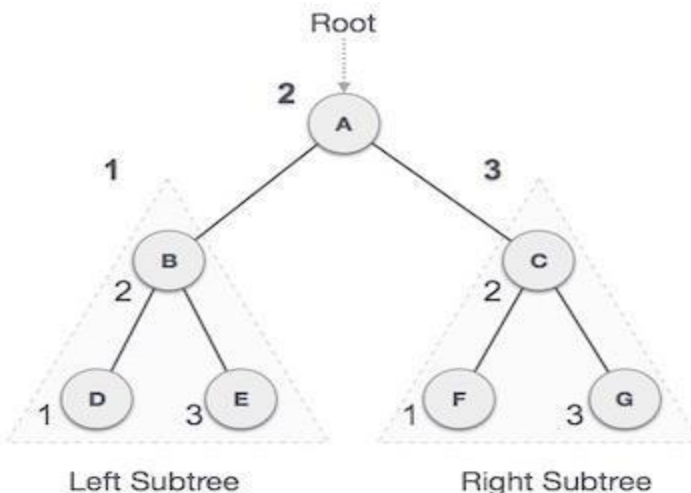
- In-order Traversal, L D R
- Pre-order Traversal, D L R
- Post-order Traversal, L R D

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

1.In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

Algorithm Procedure RINORDER(T)

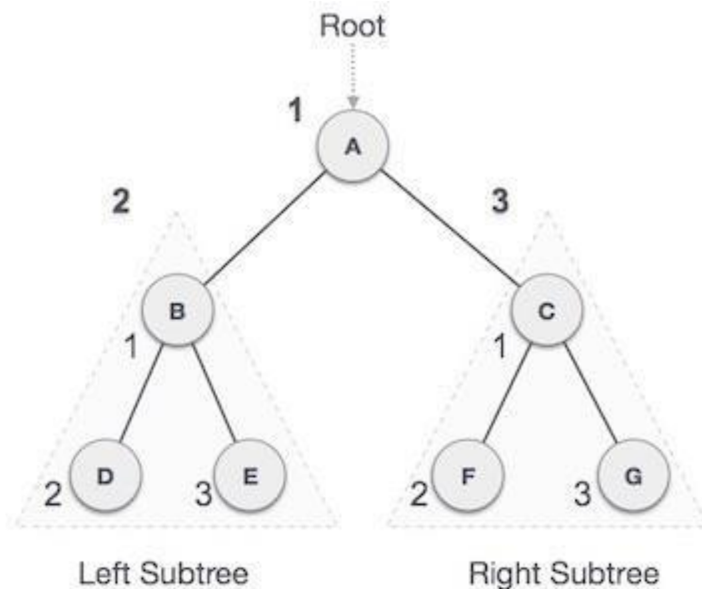
Given a binary tree whose root node address is given by a pointer variable T and this algorithm traverses the tree in Inorder in a **recursive** manner

Step 1 – [Check for empty tree]
If T =NULL
Then write ('Empty Tree')
Return
Step 2 – [Process left subtree]
If LPTR(T) ≠ NULL
Then Call RINORDER(LPTR(T))
Step 3 – [Process the root node]
Write (DATA(T))
Step 4 − [Process Right subtree]
If RPTR(T) ≠ NULL
Then Call RINORDER(RPTR(T))

Step 5 – [Finish]
Return

2.Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

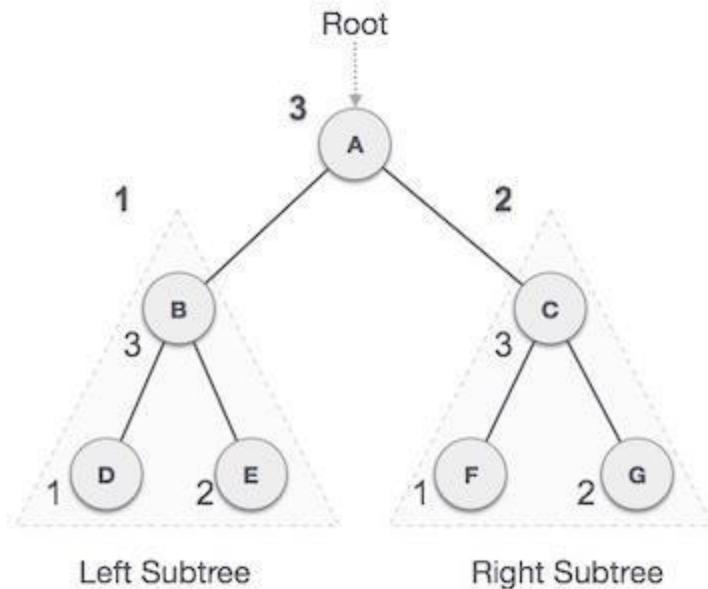
Algorithm Procedure RPREORDER(T)

Given a binary tree whose root node address is given by a pointer variable T and this algorithm traverses the tree in preorder in a recursive manner

Step 1 – [Process the root node]
If T ≠ NULL
Then Write (DATA(T))
Else write (‘Empty Tree’)
Return
Step 2 – [Process left subtree]
If LPTR(T) ≠ NULL
Then Call RPREORDER(LPTR(T))
Step 3 − [Process Right subtree]
If RPTR(T) ≠ NULL
Then Call RPREORDER(RPTR(T))
Step 4 – [Finish]
Return

3.Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

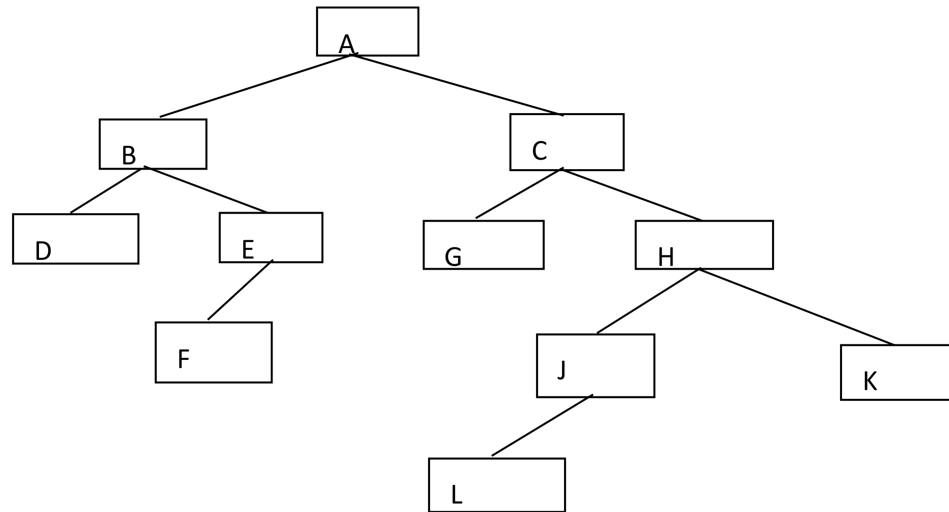
D → E → B → F → G → C → A

Algorithm Procedure RPOSTORDER(T)

Given a binary tree whose root node address is given by a pointer variable T and this algorithm traverses the tree in postorder in a recursive manner

Step 1 – [Check for empty tree]
If T =NULL
Then write (‘Empty Tree’)
Return
Step 2 – [Process left subtree]
If LPTR(T) ≠ NULL
Then Call RPOSTORDER(LPTR(T))
Step 3 − [Process Right subtree]
If RPTR(T) ≠ NULL
Then Call RPOSTORDER(RPTR(T))
Step 4 – [Process the root node]
Write (DATA(T))
Step 5 – [Finish]
Return

Example



(Binary Tree)

The tree traversals of the binary tree as follows:

Preorder traversal: Node –> Left -> Right

A B D E F C G H J L K

Inorder traversal: Left -> Node -> Right
D B F E A G C L J H K

Post order traversal: Left -> Right -> Node

D F E B G L J K H C A

BINARY SEARCH TREES

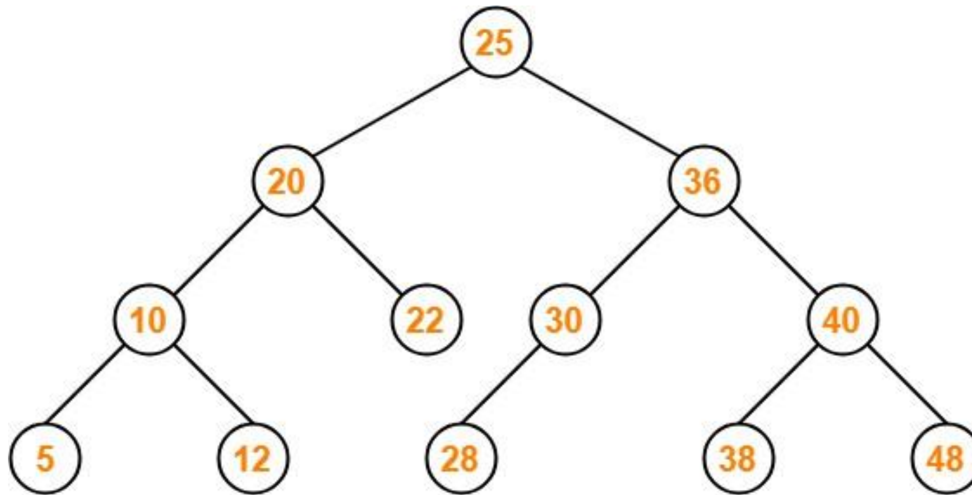
A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) < node (key) ≤ right_subtree (keys)

A binary tree is said to be a Binary Search Tree if each node N of T has the following property:
“The value at N is greater than every value in the left sub tree of N and is less than every value in the right sub tree of N”.



Binary Search Tree

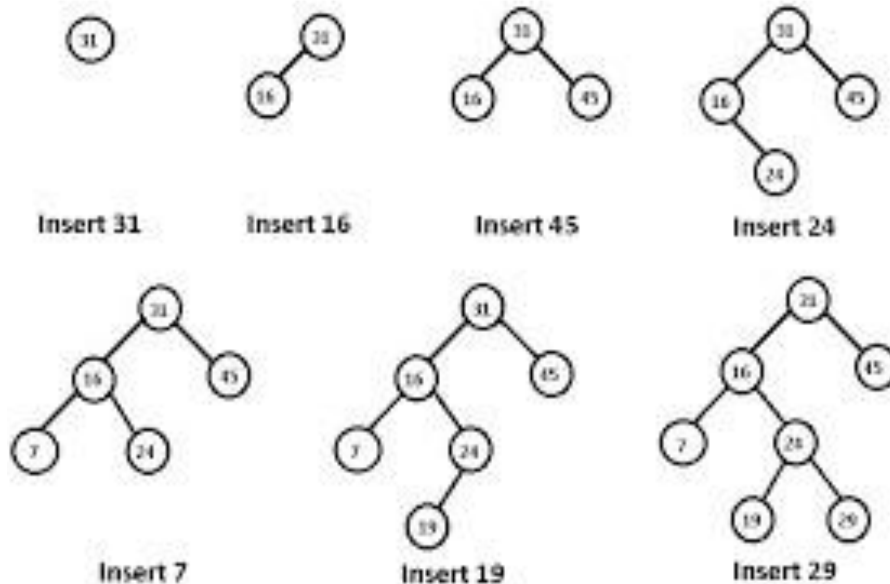
Basic Operations

Following are the basic operations are performed in a binary search tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Delete** – Delete an element from a tree
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

(a) Searching and Inserting in a Binary Search Tree

In binary search tree searching and inserting will be given by a single search and insertion algorithm. The elements are 31, 16, 45, 24, 7, 19 and 29.The elements are inserted on after the other and the Binary search tree is created.



Suppose ITM is the data given. It is required to find the location of ITEM in a binary search tree Tor inserts ITEM as a new node in appropriate place.

(a) Compare ITEM with the root node N
(i) If ITEM < N, proceed to the left child of N (ii) If ITEM > N, proceed to the right child of N (b) Repeat step (a) until one of the following occurs:

(i) We meet nod N such that ITEM = N(successful search)
(j) We meet an empty sub tree, indicate that search is unsuccessful and insert 'ITEM' in place of the empty sub tree.

(b) Deleting in a Binary Search Tree

Suppose T is a binary search tree and an ITEM of information is given which is to be deleted. The deletion algorithm find the location of the node N which contains ITEM and also the location of the parent node P(N). The way N is deleted from the tree depends primarily on the number of children of node N. There are there cases:

Case 1: N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer.

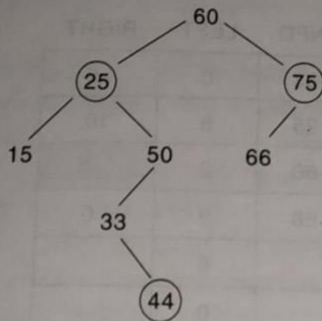
Case 2: N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

Case 3: N has two children. Let S(N) denote the inorder of N. Then N is deleted from T by first deleting S(N) from T and then replacing node N in T by the node S(N).

Example: all the three cases were considered as follows

Example 7.18

Consider the binary search tree in Fig. 7.25(a). Suppose T appears in memory as in Fig. 7.25(b).



(a) Before deletions

ROOT: 3

AVAIL: 5

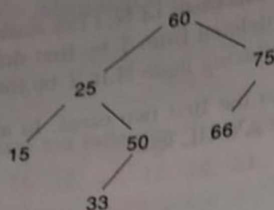
	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

(b) Linked representation

Fig. 7.25

(a) Suppose we delete node 44 from the tree T in Fig. 7.25. Note that node 44 has no children. Figure 7.26(a) pictures the tree after 44 is deleted, and Fig. 7.26(b) shows the linked representation in memory. The deletion is accomplished by simply assigning NULL to the parent node, 33. (The shading indicates the changes.)

(b) Suppose we delete node 75 from the tree T in Fig. 7.25 instead of node 44. Note that node 75 has only one child. Figure 7.27(a) pictures the tree after 75 is deleted, and Fig. 7.27(b) shows the linked representation. The deletion is accomplished by changing the right pointer of the parent node 60, which originally pointed to 75, so that it now points to node 66, the only child of 75. (The shading indicates the changes.)



(a) Node 44 is deleted

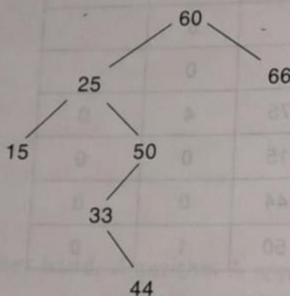
	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

ROOT 3

AVAIL 9

(b) Linked representation

Fig. 7.26



(a) Node 75 is deleted

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	4
4	66	0	0
5		6	
6		0	
7		5	
8	15	0	0
9	44	0	0
10	50	1	0

ROOT 3

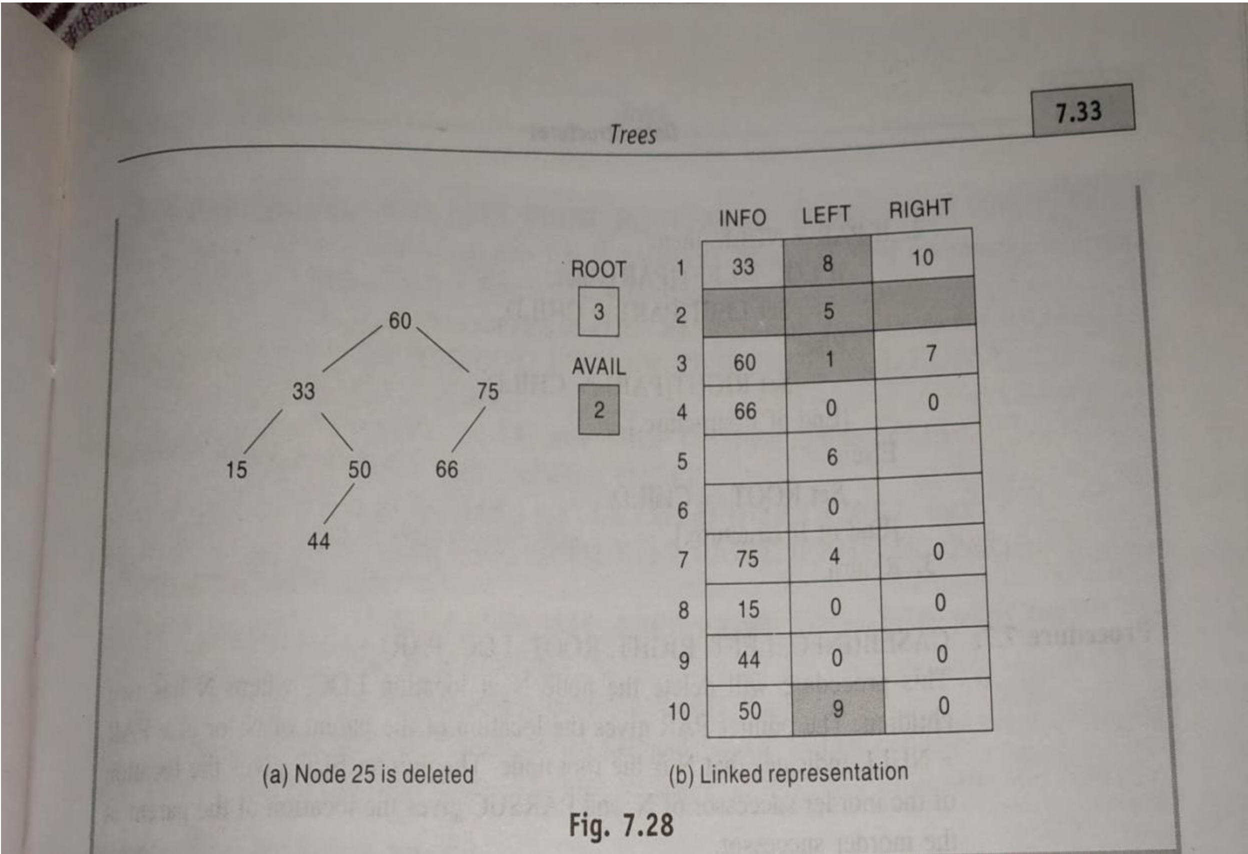
AVAIL 7

(b) Linked representation

Fig. 7.27

(c) Suppose we delete node 25 from the tree T in Fig. 7.25 instead of node 44 or node 75. Note that node 25 has two children. Also observe that node 33 is the inorder successor of node 25. Figure 7.28(a) pictures the tree after 25 is deleted, and Fig. 7.28(b) shows the linked representation. The deletion is accomplished by first deleting 33 from the tree and then replacing node 25 by node 33. We emphasize that the replacement of node 25 by node 33 is executed in memory only by changing pointers, not by moving the contents of a node from one location to another. Thus 33 is still the value of INFO[1].

REDMI NOTE 5 PRO
MI DUAL CAMERA



	INFO	LEFT	RIGHT
1	33	8	10
2		5	
3	60	1	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	9	0

(a) Node 25 is deleted (b) Linked representation

Fig. 7.28

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minimum effort

Chapter 7.0 GRAPH

Section 7.1 Graph terminology and its representation

A **Graph** is a non-linear **data structure** consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the **graph**. **Graphs** are used to represent networks.

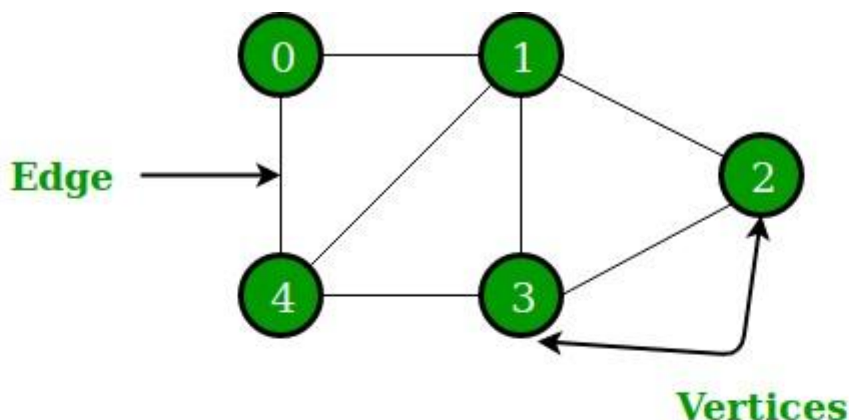
A graph is a pictorial **representation** of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the **vertices** are called edges.

Graphs are a powerful and versatile **data structure** that easily allow you to represent real life relationships between different types of **data** (nodes). ... The edges (connections) which connect the nodes i.e. the lines between the numbers in the image.

A graph G consists of two things

1. **A set V of elements called nodes (or points, or vertices)**
2. **A set E of edges such that each edge 'e' in E is identified with a unique unordered pair [u,v]of nodes in V, denoted by e = [u,v]**

Sometimes we indicate the parts of a graph by writing G = (V,E)



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Suppose **e = [u,v],** then the nodes u and v are called the **endpoints** of e and u and v are said to be **adjacent nodes or neighbors.**

Degree of a node u: it is the number of edges containing by u and denoted by deg(u).

If deg(u)=0 i.e u does not belong to any edge and u is called an isolated node.

Path: A path P of length n from a node u to a node v is defined as a sequence of n+1 nodes

P = $(v_0,v_1,v_2,\ldots\ldots v_n)$

Such that $u=v_0$; v_{i-1} is adjacent to v_i for i=1,2,…n; and v_n = v.

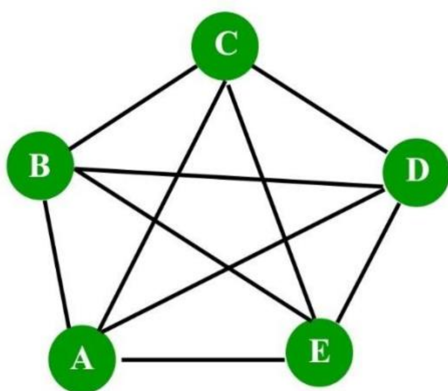
Closed path: A path is said to be closed if $v_0 = v_n$

Simple path: The path P is said to be simple if all the nodes are distinct except v_0 may equal v_n. P is simple if the nodes **$v_0,v_1,v_2,\ldots\ldots v_{n-1}$ are distinct and the nodes $v_1,v_2,\ldots\ldots v_n$ are distinct.**

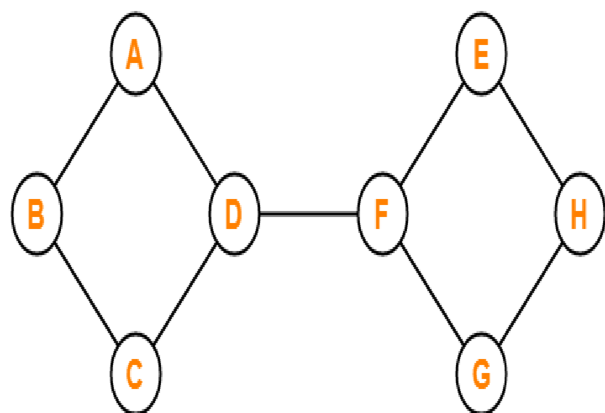
Cycle: A cycle is a closed simple path with length 3 or more. A cycle of length 'k' is called a k-cycle.

Connected graph: A graph G is said to be connected if there is a path between any two nodes.

Complete graph: A graph is said to be complete if every node 'u' in G is adjacent to every other node 'v' of G.A complete graph with 'n' nodes will have n(n-1)/2 edges.



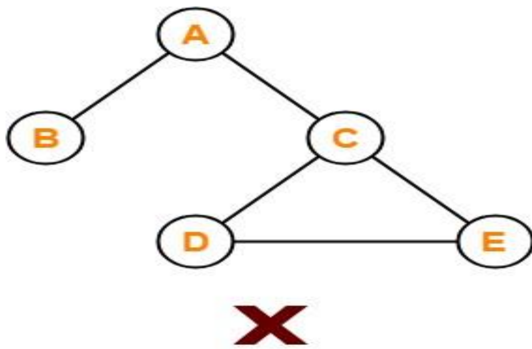
Complete Graph

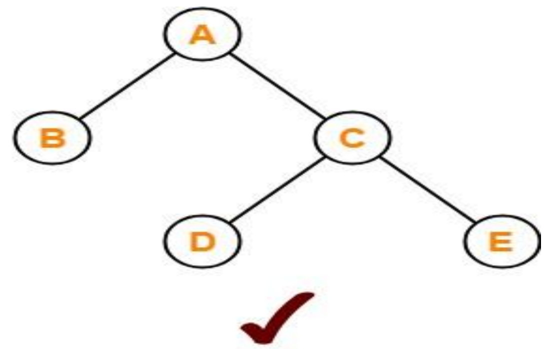


Example of Connected Graph

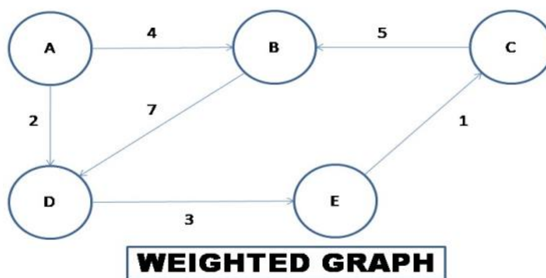
Tree graph or free graph: A connected graph T without any cycle is called a tree graph or free graph i.e there is a unique simple path P

between any two nodes u and v in T. If T is a finite tree with m nodes then T will have m-1 edges.





Weighted graph: A **weighted graph** is a **graph** in which each branch is given a numerical **weight**. A **weighted graph** is therefore a special type of labeled **graph** in which the labels are numbers (which are usually taken to be positive). Each edge 'e' in G is assigned a non-negative numerical value w(e) called the weight or length of 'e'. Here each path P in G is assigned a weight which is the sum of the weights of the edges along the path P.



Multigraph: A graph whose edges are unordered pairs of vertices, and the same pair of vertices can be connected by multiple edges. A **multigraph** is a graph that can have more than one edge between a pair of vertices.

1.**Multiple edges**: Distinct edges e and e' are called multiple edges if they connect the same end points i.e if e=[u,v] and e'= [u,v].

2.loops: An edge e called a loop if it has identical end points i.e if e=[u,u]

Example 8.1

(a) Figure 8.1(a) is a picture of a connected graph with 5 nodes—A, B, C, D and E—and 7 edges:

$$[A, B], \quad [B, C], \quad [C, D], \quad [D, E], \quad [A, E], \quad [C, E] \quad [A, C]$$

There are two simple paths of length 2 from B to E: (B, A, E) and (B, C, E). There is only one simple path of length 2 from B to D: (B, C, D). We note that (B, A, D) is not a path, since $[A, D)$ is not an edge. There are two 4-cycles in the graph:

$$[A, B, C, E, A] \quad \text{and} \quad [A, C, D, E, A].$$

Note that $\deg(A) = 3$, since A belongs to 3 edges. Similarly, $\deg(C) = 4$ and $\deg(D) = 2$.

(b) Figure 8.1(b) is not a graph but a multigraph. The reason is that it has multiple edges—$e_4 = [B, C]$ and $e_5 = [B, C]$—and it has a loop, $e_6 = [D, D)$. The definition of a graph usually does not allow either multiple edges or loops.

(c) Figure 8.1(c) is a tree graph with $m = 6$ nodes and, consequently, $m - 1 = 5$ edges. The reader can verify that there is a unique simple path between any two nodes of the tree graph.

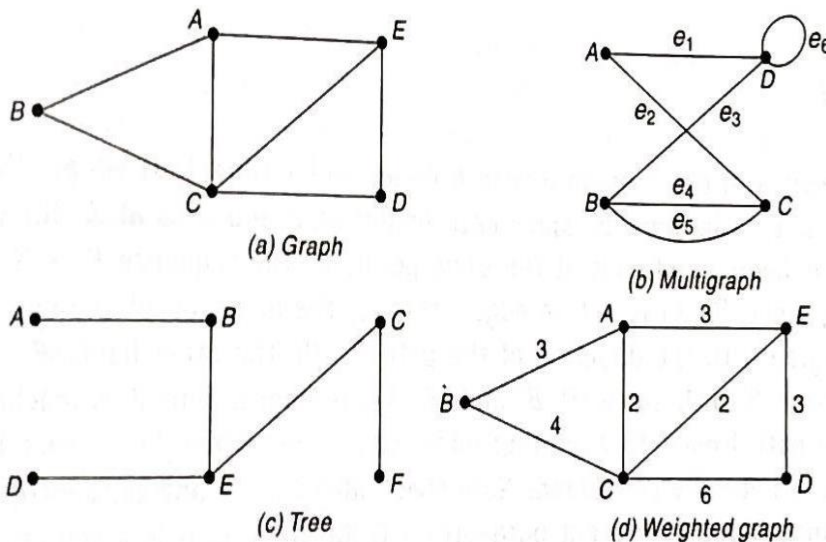


Fig. 8.1

(d) Figure 8.1(d) is the same graph as in Fig. 8.1(a), except that now the graph is weighted. Observe that $P_1 = (B, C, D)$ and $P_2 = (B, A, E, D)$ are both paths from node B to node D. Although P_2 contains more edges than P_1 the weight $w(P_2) = 9$ is less than the weight $w(P_1) = 10$.

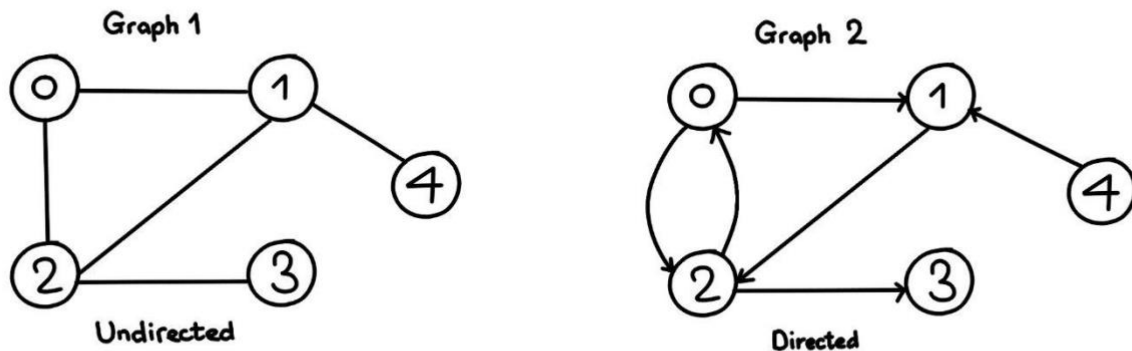
Types of graphs: There are two types of graphs

Undirected Graph:

1.Undirected graphs have edges that do not have a direction. The edges indicate a two-way relationship, in that each edge can be traversed in both directions. In an undirected graph, nodes are connected by edges that are all bidirectional. For example, if an edge connects node 1 and 2, we can traverse from node 1 to node 2, and from node 2 to 1.

2.Directed Graph

In a directed graph, nodes are connected by directed edges – they only go in one direction. For example, if an edge connects node 1 and 2, but the arrow head points towards 2, we can only traverse from node 1 to node 2 – not in the opposite direction.



The **main difference** between directed and undirected graph is that **a directed graph contains an ordered pair of vertices whereas an undirected graph contains an unordered pair of vertices.**

- An **edge** is (together with vertices) one of the two basic units out of which graphs are constructed. Each edge has two vertices to which it is attached, called its **endpoints**.
- Two vertices are called **adjacent** if they are endpoints of the same edge.
- **Outgoing edges** of a vertex are directed edges that the vertex is the origin.

- **Incoming edges** of a vertex are directed edges that the vertex is the destination.

- The **degree** of a vertex in a graph is the number of edges incident to it.

- In a directed graph, **outdegree** of a vertex is the number of outgoing edges from it and **indegree** is the number of incoming edges.

- A vertex with indegree zero is called a **source vertex**, while a vertex with outdegree zero is called sink vertex.

- An **isolated vertex** is a vertex with **degree zero**; that is, a vertex that is not an endpoint of any edge.
- A node ‘v’ is said to be reachable from a node ‘u’ if there is a path from ‘u’ to ‘v’.

- **Path** is a sequence of alternating vetches and edges such that each successive vertex is connected by the edge.

- **Cycle** is a path that starts and end at the same vertex.

- **Simple path** is a path with distinct vertices.

- A graph is **Strongly Connected** if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v.
- A graph G is said to **be unilaterally connected** if for every pair u,v of nodes in G there is a path from u to v or a path from v to u.

- A directed graph is called **Weakly Connected** if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. The vertices in a weakly connected graph have either outdegree or indegree of at least 1.

- **Connected component** is the maximal connected sub-graph of an unconnected graph.

- A **bridge** is an edge whose removal would disconnect the graph.

- **Forest** is a graph without cycles.

- **Tree** is a connected graph with no cycles. If we remove all the cycles from DAG(Directed acyclic graph) it becomes tree and if we remove any edge in a tree it becomes forest.

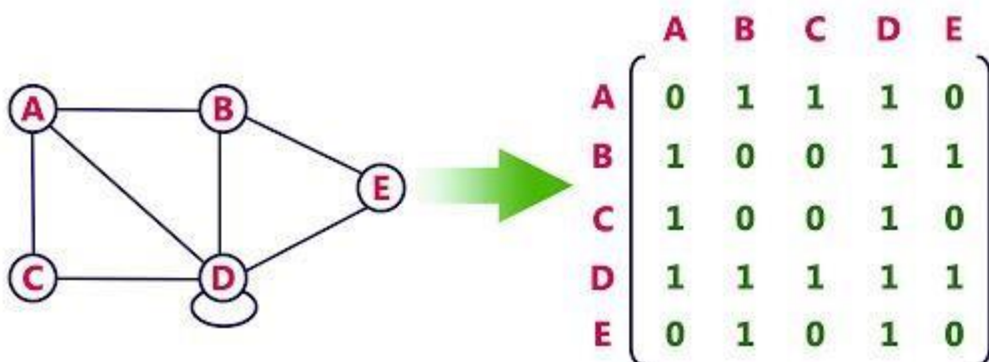
- **Spanning tree** of an undirected graph is a subgraph that is a tree which includes all of the vertices of the graph.
- **A** directed graph G is said to be **simple** if G has no parallel edges. A simple graph G may have loops, but it cannot have more than one loop at a given node.

REPRESENTATION OF GRAPHS IN MEMORY

There are two standard ways of maintaining a graph G in memory of a computer.

1. Sequential Representation
2. Linked Representation

   The Sequential Representation of graph G is made by means of its **adjacency matrix** 'A'.



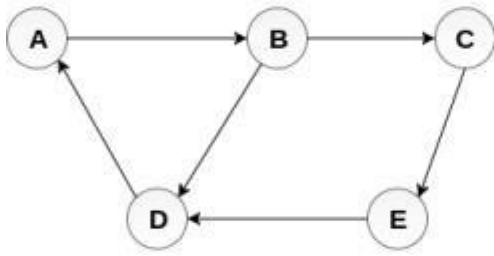
(Undirected graph G) (Adjacency matrix 'A')

Suppose G is a **simple directed graph** with 'm' nodes and the nodes of G have been ordered and are called $\mathbf{v_1, v_2, \ldots .. v_m}$. Then the adjacency matrix A = (a_{ij}) of a graph G is the m x n matrix denoted as follows:

a_{ij} = 1 if v_i is adjacent to v_j, that is if there is an edge (v_i , v_j) a_{ij}

= 0 otherwise

Such a matrix A which contains entries of 0 and 1 is called a bit matrix or Boolean matrix.



Directed Graph

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

Adjacency Matrix

If G is an undirected graph then the Adjacency matrix 'A' is a symmetric matrix that is $a_{ij} = a_{ji}$ for every I and j

Let

Example 8.3

Consider the graph G in Fig. 8.3. Suppose the nodes are stored in memory in a linear array DATA as follows:

DATA: X, Y, Z, W

Then we assume that the ordering of the nodes in G is as follows: $v_1 = X$, $v_2 = Y$, $v_3 = Z$ and $v_4 = W$. The adjacency matrix A of G is as follows:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Note that the number of 1's in A is equal to the number of edges in G.

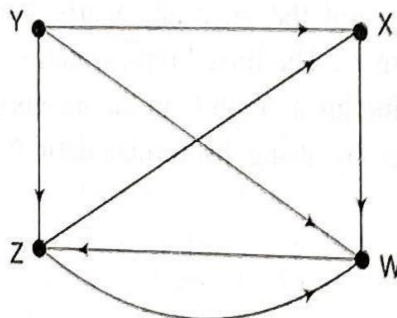


Fig. 8.3

Consider the powers A, A^2, A^3 … of the adjacency matrix A of graph G. Let us define

$a_k(i,j)$ = the ij entry in the matrix A^k

$a_1(i,j) = a_{ij}$ gives the number of paths of length 1 from node v_i to node v_j. Similarly $a_2(i,j) = a_{ij}$ gives the number of paths of length 2 from node v_i to node v_j.

Let A be the adjacency matrix of a graph G, then **$a_k(i,j)$** the ij entry in the matrix **A^k** gives the number of paths of length **K** from v_i to v_j.

From the above graph, we can find A^2, A^3, A^4 …. By multiplying A with itself and so on.

x y z w

x 0 0 1 0 here path length from X to Z, Y to X etc is

2 A^2 = y 1 0 1 2 Two paths from Y to W of path length 2

z 0 0 1 1 w 1 0 0 1

x y z w

x 1 0 0 1 here path length from X to X, Y to X etc is 3 A^3

= y 1 0 2 2 z 1 0 1 1 w 0 0 1 1 x

y z w x 0 0 1 1 A^4 = y 2 0 2 3 z 1 0 1 2

w 1 0 1 1

Now we can define the matrix B_r as follows:

$B_r = A + A^2 + A^3 + A^4 + \ldots\ldots + A^r$

Then the ij entry of the matrix B_r gives the number of paths of length 'r' or less from node v_i to v_j.

Path matrix

Let G be a simple directed graph with m nodes **$v_1, v_2, \ldots.., v_m$.** the path matrix or **reachability matrix** of G is the m-square matrix $P = (p_{ij})$ defined as follows:

p_{ij} = 1 if there is a path from v_i to v_j, p_{ij}

= 0 otherwise

Let A be the adjacency matrix and let P= (p_{ij}) be the path matrix of a directed graph, Then pij =1 if and only if there is a non-zero number in the ij entry of the matrix

$B_m = A + A^2 + A^3 + A^4 + \ldots\ldots + A^m$

Consider the graph with m = 4 nodes. Adding A, A^2, A^3 and A^4 we obtain the following matrix B_4 and by replacing the non-zero entries in B_4 by 1 we obtain the path matrix

x y z w

x 1 0 2 3

B_4 = y 5 0 6 8 and the path matrix P is defined

as z 3 0 3 5

w 2 0 3 3

x y z w x 1

0 1 1 P = y 1

0 1 1 z 1

0 1 1 w 1

0 1 1

Here node Y is not reachable from any node. As we know a graph is **Strongly Connected** if it contains a directed path from u to v and a **directed path** from v to u for every pair of vertices u, v. Accordingly G is strongly connected if and only if the path matrix P of G has no zero entries. The graph in this example is not strongly connected.

The Adjacency matrix and the Path matrix P of a graph G may be viewed as logical matrices where ‘0’ and’1’ represent false and true respectively. Thus the logical operations AND & OR may be applied to the entries of A and P.

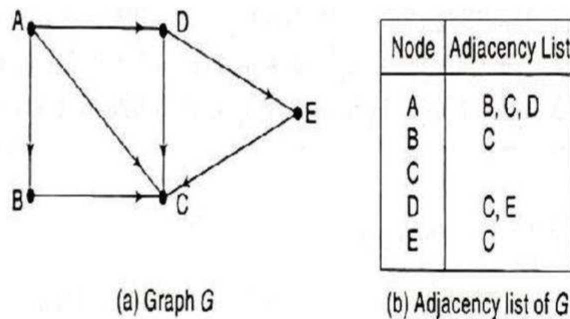
AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

8.5 LINKED REPRESENTATION OF A GRAPH

Let G be a directed graph with m nodes. The sequential representation of G in memory—i.e., the representation of G by its adjacency matrix A—has a number of major drawbacks. First of all, it may be difficult to insert and delete nodes in G. This is because the size of A may need to be changed and the nodes may need to be reordered, so there may be many, many changes in the matrix A. Furthermore, if the number of edges is $O(m)$ or $O(m \log_2 m)$, then the matrix A will be sparse (will contain many zeros); hence a great deal of space will be wasted. Accordingly, G is usually represented in memory by a linked *representation,* also called an *adjacency structure,* which is described in this section.

Consider the graph G in Fig. 8.7(a). The table in Fig. 8.7(b) shows each node in G followed by its *adjacency list,* which is its list of adjacent nodes, also called its *successors* or *neighbors.* Figure 8.8 shows a schematic diagram of a linked representation of G in memory. Specifically, the linked representation will contain two lists (or files), a node list NODE and an edge list EDGE, as follows.



(a) Graph G

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

(b) Adjacency list of G

Fig. 8.7

(a) *Node list.* Each element in the list NODE will correspond to a node in G, and it will be a record of the form:

NODE	NEXT	ADJ	

Here NODE will be the name or key value of the node, NEXT will be a pointer to the next node in the list NODE and ADJ will be a pointer to the first element in the adjacency list of the node, which is maintained in the list EDGE. The shaded area indicates that there may be

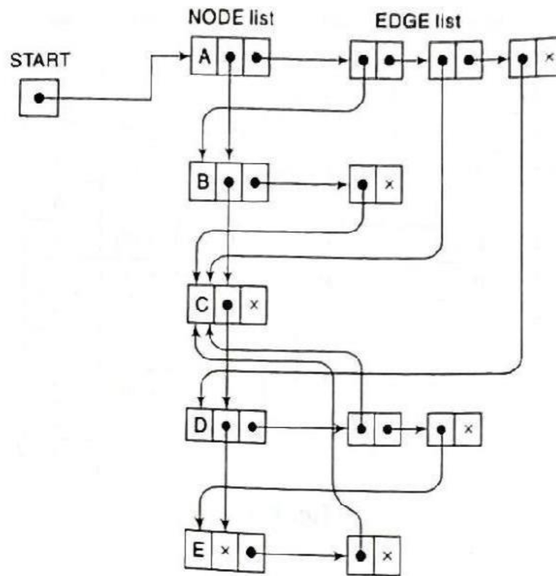


Fig. 8.8

other information in the record, such as the indegree INDEG of the node, the outdegree OUTDEG of the node, the STATUS of the node during the execution of an algorithm, and so on. (Alternatively, one may assume that NODE is an array of records containing fields such as NAME, INDEG, OUTDEG, STATUS, ...) The nodes themselves, as pictured in Fig. 8.7, will be organized as a linked list and hence will have a pointer variable START for the beginning of the list and a pointer variable AVAILN for the list of available space. Sometimes, depending on the application, the nodes may be organized as a sorted array or a binary search tree instead of a linked list.

(b) *Edge list.* Each element in the list EDGE will correspond to an edge of G and will be a record of the form:

DEST	LINK	

The field DEST will point to the location in the list NODE of the destination or terminal node of the edge. The field LINK will link together the edges with the same initial node, that is, the nodes in the same adjacency list. The shaded area indicates that there may be other information in the record corresponding to the edge, such as a field EDGE containing the labeled data of the edge when G is a labeled graph, a field WEIGHT containing the weight of the edge when G is a weighted graph, and so on. We also need a pointer variable AVAILE for the list of available space in the list EDGE.

Figure 8.9 shows how the graph G in Fig. 8.7(a) may appear in memory. The choice of 10 locations for the list NODE and 12 locations for the list EDGE is arbitrary.

The linked representation of a graph G that we have been discussing may be denoted by

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

The representation may also include an array WEIGHT when G is weighted or may include an array EDGE when G is a labeled graph.

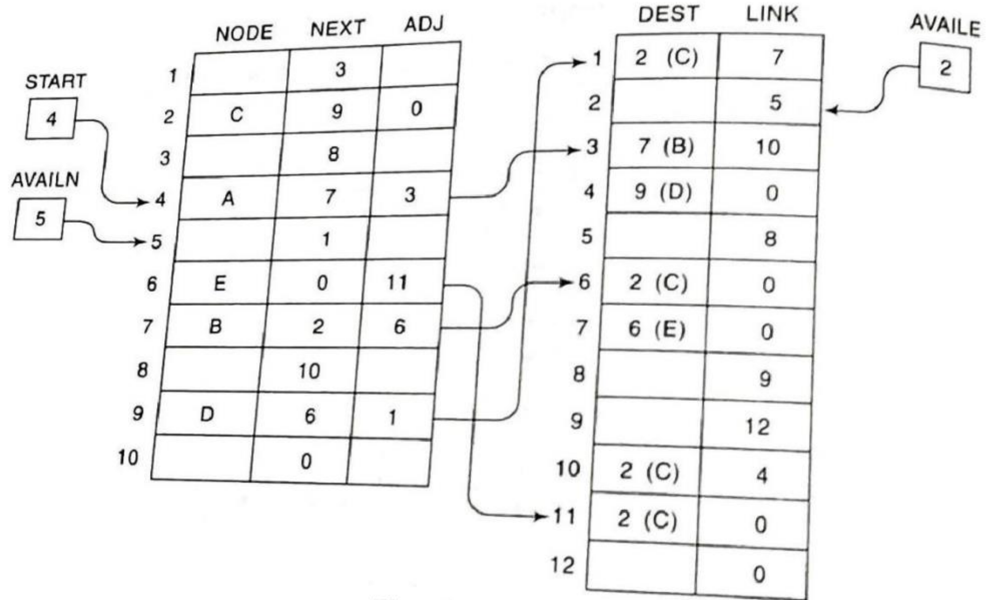


Fig. 8.9

CHAPTER 8 Sorting and Searching

Searching here refers to finding an item in the array that meets some specified criterion.

Searching Algorithms Techniques:

- Linear **Search**.
- Binary **Search**.

Sorting refers to rearranging all the items in the array into increasing or decreasing order. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

Following are some of the examples of sorting in real-life scenarios −

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

The different sorting techniques are

- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Bubble sort Radix sort

Three interesting issues to consider when thinking about different sorting algorithms are:

- Does an algorithm always take its worst-case time?
- What happens on an already-sorted array?
- How much space (other than the space for the array itself) is required?

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$ where **n** is the number of items.

BUBBLE SORT

Implementing Bubble Sort Algorithm

1. Starting with the first element (index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.

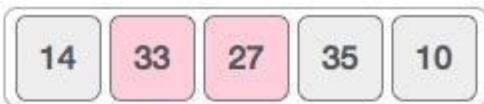
3. If the current element is less than the next element, move to the next element.
4. Repeat Step 1.

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

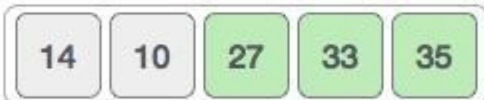
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

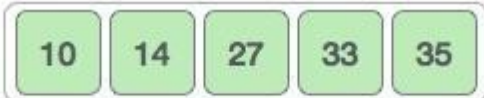
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Algorithm:

BUBBLE (DATA,N)

Here DATA is an array with N element. This algorithm sorts the element in DATA.

Step 1: [Loop]

Repeat step 2 and step 3 for K=1 to N-1

Step 2: [Initialize pass pointer PTR]

Set PTR :=1

Step 3: [Execute pass]

Repeat while PTR <= N-K

a. If DATA [PTR] > DATA [PTR+1]

Then interchange DATA [PTR] & DATA [PTR+1]

[End of if structure]

b. Set PTR =PTR+1

[End of Step 1 Loop]

Step 4: Exit

Complexity of the Bubble Sort Algorithm

The time for a sorting algorithm is measured in terms of the number of

comparisons. The number f(n) of comparisons in the bubble sort is easily computed. There are n-1 comparisons during the first pass, which places the largest element in the last position; there are n-2 comparisons in the second step, which places the second largest element in the next to last position and so on.

$F(n)=(n-1)+(n-2)+....+2+1=n(n-1)/2=n^2/2+O(n)$ **= $O(n^2)$**

The time required to execute the bubble sort algorithm is proportional to n^2, Where n is the number of input items.

QUICK_SORT OR PARTITION EXCHANGE SORT

We use two index variables I and j with initial values of 2 and 10 respectively. The two keys 42 and K_i are compared and if an exchange is required ($K_i < 42$) then i is incremented by 1 and the process is repeated. When $K_j<=42$ we proceed to compare K_j and 42. If an exchange is required then j is decremented by 1 and the process is repeated until $K_j<=42$. At this point the keys K_i and K_j (i.e 74 and 36) are interchanged. The entire process is then repeated with j fixed and I being incremented once again. When i>=j, the desired key is placed in its final position by interchanging keys 42 and K_j.

42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
42	23	36	11	65	58	94	74	99	87
42	23	36	11	65 i=5	58	94	74	99	87
42	23	36	11	65	58	94	74	99	87
42	23	36	11	65	58	94	74	99	87
42	23	36	11	65	58	94	74	99	87
42	23	36	11 j=4	65	58	94	74	99	87
11	23	36	42	65	58	94	74	99	87

QUICK_SORT(K,LB,UB)

Given a table K of N records. LB and UB denote the lower and upper bounds of the current sub table being processed. KEY the key value which is being placed in final position within the sorted sub table. **FLAG is a logical variable** which indicates

the end of the process. I and J are indices used to select certain keys during processing of sub tables.

```
Step 1 [Initialize]
        FLAG = true
Step 2  [Perform Sort]
         If LB<UB
         Then I = LB
               J = UB+1
               KEY = K[LB]
               Repeat while FLAG
                   I = I + 1
                   Repeat while K[I] < KEY     (scan the keys from left to right)
I  = I+1
J = J-1
                  Repeat while K[J] > KEY     (scan the keys from right to left)
                       J=J-1
                   If I <J
                   Then K[I] ←→   K[J]         ( Interchange records)
                   Else  FLAG = false
               K[LB] ←→   K[J]         ( Interchange records)
               Call QUICK_SORT(K,LB,J-1)       (Sort first subtable)
               Call QUICK_SORT(K,J+1,UB)       (Sort second subtable)
Step 3 [Finish]
        Exit
```

Complexity of the Quick Sort Algorithm

The time for a sorting algorithm is measured in terms of the number of comparisons. The best case analysis occurs when the table is always partitioned in half that is J = (LB+UB)/2

Best case complexities $O(N\log_2N)$ where N is the no. of records.

The average case analysis of procedure QUICK_SORT is also **$O(N\log_2 N)$.**

MERGING:

The operation of sorting is closely related to the process of merging. The merging of two order table which can be combined to produce a single sorted table. This process can be accomplished easily by successively selecting the record with the smallest key occurring by either of the table and placing this record in a new table.

Table1	**11 23 42 52**
Table 2	**9 25**
New table	**9**
Table1	11 23 42 52
Table 2	25
New table	**9 11**
Table 1	23 42 52
Table 2	25
New table	**9 11 23**
Table1	42 52
Table 2	25
New table	**9 11 23 25**
Table 1	42 52
Table 2	nil
New table	**9 11 23 25 42 52**

Values can be stored in a vector k

K	**11**	**23**	**42**	**52**	**9**	**25**
	FIRST				**SECOND**	**THIRD**

SIMPLE MERGE

SIMPLE MERGE [FIRST,SECOND,THIRD,K]

Given two orders in table sorted in a vector K with FIRST, SECOND, THIRD The variable I & J denotes the curser associated with the FIRST & SECOND table respectively. L is the index variable associated with the vector TEMP.

Algorithm

Step 1: [Initialize]

Set I = FIRST

Set J = SECOND

Set L = 0

Step 2: [Compare corresponding elements and output the smallest]

Repeat while I < SECOND & J < =THIRD

If K[I] <= K[J],

then L = L+1

TEMP [L]=K[I]

I=I+1

Else

L=L+1

TEMP[L]=K[J]

J=J+1

Step 3: [Copy remaining unprocessed element in output area]

If I>=SECOND

Then repeat while J<=THIRD

L=L+1

TEMP[L]=K[J]

J=J+1

Else

Repeat while I<SECOND

L=L+1

TEMP[L]= K[I]

I=I+1

Step 4: [Copy elements in temporary vector into original area]

Repeat for I = 1,2,……..L

K[FIRST-1+I] = TEMP[I]

Step 5: [Finished]

Return

COMPLEXITIES OF MERGE SORT

Complexity of an algorithm is a measure of the amount of time and /or space required by an algorithm for an input of given size (n).
Time **complexity** of **Merge Sort** is O(n*Log n) in all the 3 cases (worst, average and best) as **merge sort** always divides the array in two halves and takes linear time to **merge** two halves. It requires equal amount of additional space as the unsorted array. It is more efficient as it is in worst case also the runtime is O(nlogn) The space **complexity of Merge sort** is O(n).

SEARCHING:

Searching refers to finding the location i.e LOC of ITEM in an array. The search is said to be successful if ITEM appears the array & unsuccessful otherwise we have two types of searching techniques.

1. Linear Search
2. Binary Search

LINEAR SEARCH:

Suppose DATA is a linear array with n elements. No other information about DATA, the most intuitive way to search for a **given ITEM** in DATA is to compare ITEM with each element of DATA one by one. First we have to test whether DATA [1]=ITEM, nad then we test whether DATA[2] =ITEM , and so on. This method which traverses DATA sequentially to locate ITEM, is called *linear search* or *sequential search*.

Given a array, Search a given element in array.

Case 1:

Input: Search 20.

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Output: True (20 is present in array)

Case 2:

Input: Search 26.

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Output: False (26 is not present in array)

Algorithm Linear searching

LINEAR (DATA, N, ITEM, LOC)

Step 1: [Insert ITEM at the end of DATA]

Set DATA [N+1] = ITEM

Step 2: [Initialize counter]

Set LOC :=1

Step 3: [Search for ITEM]

Repeat while DATA [LOC]!= ITEM

Set LOC := LOC+1 [end of loop]

Step 4: [Successful?]

If LOC=N+1

Then Set LOC = 0

Step 5: Exit

Let us search for 88. Further we
search for 92

We will compare 88 with DATA[1] then DATA[2] and so on till we find a match. If a match is found then it is Successful search else Unsuccessful search

DATA[!]	DATA[2] --						DATA[N]	DATA[N+1]
45	56	33	29	88	....	42	93	92

Complexity of the Linear Search Algorithm

The complexity of search algorithm is measured by the number f(n) of comparisons required to find ITEM in DATA where DATA contains n elements. Two important cases to consider are the average case and the worst case.

The worst case occurs when one must search through the entire array DATA. In this case, the algorithm requires

F(n)= n+1

Thus, in the worst case, running time is proportional to n.

The running time of the average case uses the probabilistic notion of expectation. The probability that ITEM appears in DATA[K], and q is the probability that ITEM

does not appear in DATA . Since the algorithm uses k comparison when ITEM appears in DATA[K], the average number of comparison is given by

F(n) = 1 . p1 + 2 . p2 +…..+ n . pn + (n+1).q

F(n) = 1.1/n+2.1/n+…….+n.1/n +(n+1).0

=(1+2+3+4+…..+n).1/n =n(n+1)/2 .1/n = n+1/2

BINARY SEARCH

Binary search works on sorted arrays. Binary search begins by comparing an element in the middle of the array with the target value. If the target value matches the element, its position in the array is returned. If the target value is less than the element, the search continues in the lower half of the array. If the target value is greater than the element, the search continues in the upper half of the array. By doing this, the algorithm eliminates the half in which the target value cannot lie in each iteration.

Steps of Binary searching

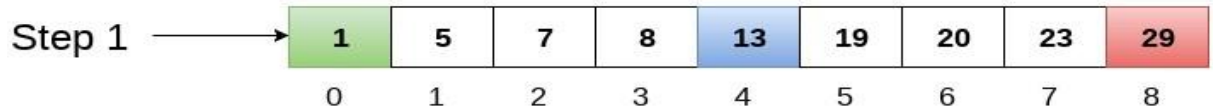
Binary search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list.
- **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.

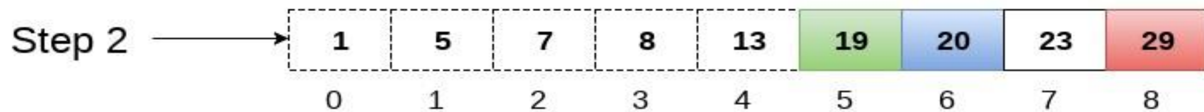
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.**

Example of Binary searching

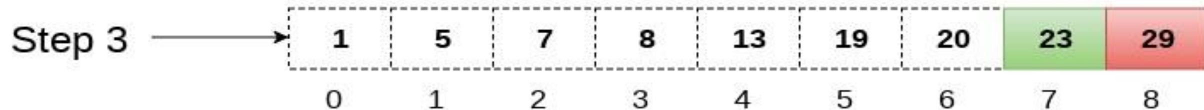
Item to be searched = 23



a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6



a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7



a [mid] = 23
23 = 23
loc = mid

Return location 7

Suppose DATA is an array which is sorted in increasing numerical order or equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called *binary search.*

Algorithm

(Binary search) BINARY (DATA, LB, UB, ITEM, LOC)

Step 1: [Initialize the segment variables]

Set BEG := LB, END := UB

and MID := INT ((BEG + END)/2)

Step 2: [Loop]
Repeat Step 3 and Step 4
while BEG <= END **and** DATA [MID] ≠ ITEM

Step 3: [Compare]

If ITEM < DATA [MID]
then set END := MID - 1
Else
Set BEG = MID + 1

Step 4: [Calculate MID]

Set MID := INT ((BEG + END)/2)

Step 5: [Successful search]

If DATA [MID] = ITEM
then set LOC := MID

Else

Set LOC := NULL

Step 6: Exit

Complexity of the Binary Search Algorithm

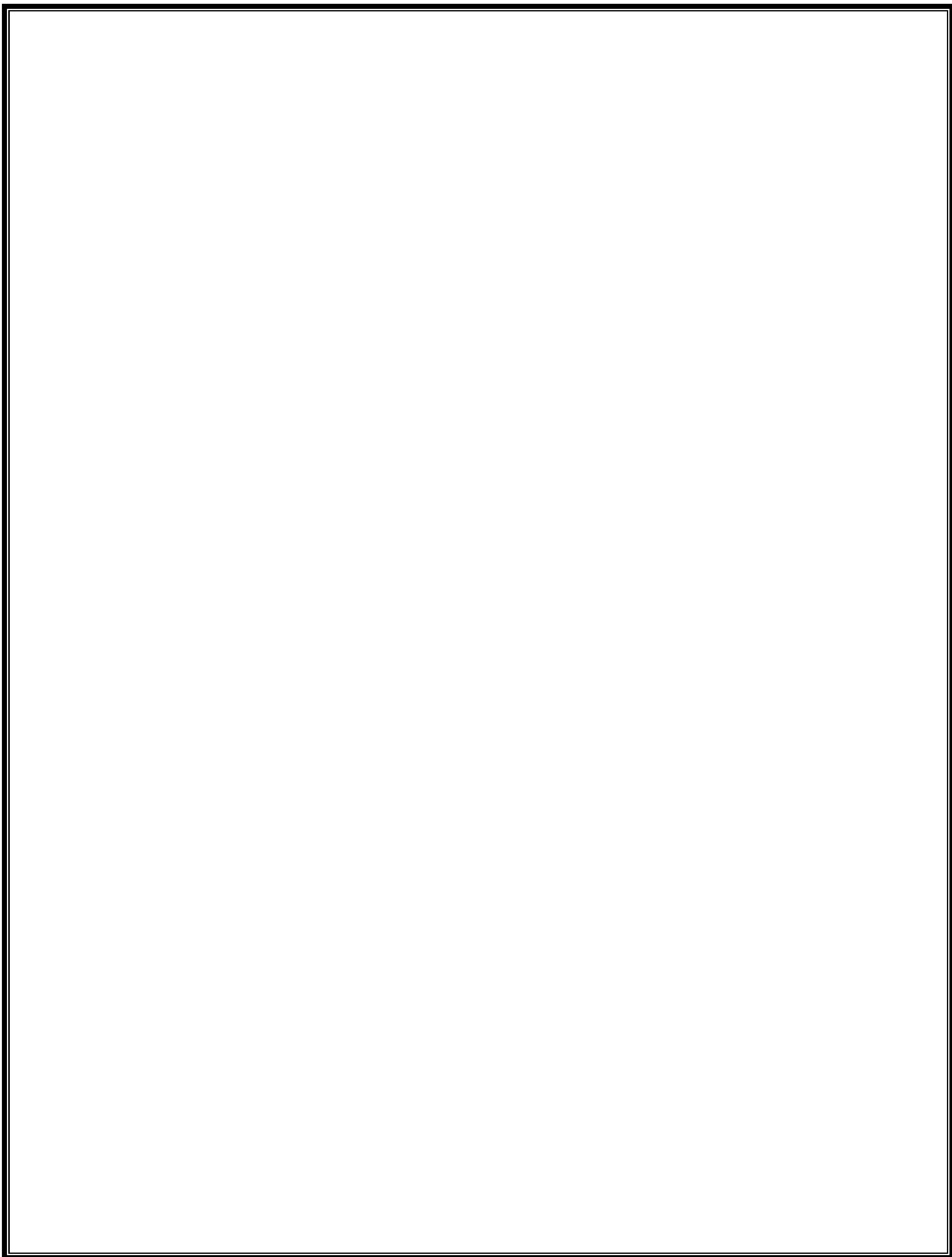
The time **complexity** of the **binary search** algorithm is O(log n). The best-case time **complexity** would be O(1) when the central index would directly match the desired value. **binary search** is far more faster-**searching** algorithm than linear **searching** if the array is sorted. And its **Big-O** run time is **O**(log n).The complexity is measured by the number f(n) of comparison to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most f(n) comparison to locate ITEM where 2f(n) > n Or equivalently $F(n) = [\log_2 n] + 1$

The running time for the worst case is approximately equal to log2 n and the average case is approximately equal to the running time for the worst case.

Time and Space Complexity Comparison Table :

SORTING ALGORITHM	TIME COMPLEXITY			SPACE COMPLEXITY
	BEST CASE	AVERAGE CASE	WORST CASE	WORST CASE
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(N \log N)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$
Binary Search	$O(1)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
Linear Search	$O(1)$	$O(N+1)/2$	$O(N+1)$	$O(N)$

SORTING ALGORITHM	TIME COMPLEXITY			SPACE COMPLEXITY
	BEST CASE	AVERAGE CASE	WORST CASE	WORST CASE



Chapter 9 File Organisation

File is a collection of records related to each other. The file size is limited by the size of memory and storage medium.

A record is a unit which data is usually stored in. Each record is a collection of related data items, where each item is formed of one or more bytes and corresponds to a particular field of the record. Records usually describe entities and their attributes. A collection of field (item) names and their corresponding data types constitutes a record type. In short, we may say that a record type corresponds to an entity type and a record of a specific type represents an instance of the corresponding entity type.

A file basically contains a sequence of records. **Usually all records in a file are of the same record type.**

In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made of **fixed-length records**. If different records in the file have different sizes, the file is said to be made up of **variable-length records**.

A file may have variable-length records for several reasons :

i) The file records are of the same record type, but one or more fields are of varying sizes (variable-length fields). ii) The file records are of the same record type but one or more fields may have multiple values for individual records. Such a field is called repeating field and a group of values for the field is often called a repeating group.

iii) The file records are of the same record type, but one or more fields are optional.

iv) The file has records of different record types and hence of varying size (mixed file). This would occur if related records of different types were clustered (placed together) on disk blocks.

File organization refers to the way **data** is stored in a **file**. File organization refers to physical layout or a structure of record occurrences in a file. File organization determines the way records are stored and accessed. **File organization** is very important because it determines the

1. Methods of access,
2. Efficiency,
3. Flexibility and
4. Storage devices to use

Types of File Organization

There are three types of organizing the file:

1. Sequential access file organization
2. Direct access file organization
3. Indexed sequential access file organization

1. Sequential access file organization

- Storing and sorting in contiguous block within files on tape or disk is called as **sequential access file organization**.
- In sequential access file organization, all records are stored in a sequential order. The records are arranged in the ascending or descending order of a key field.
- Sequential file search starts from the beginning of the file and the records can be added at the end of the file.
- In sequential file, it is not possible to add a record in the middle of the file without rewriting the file.

Advantages of sequential file

- It is simple to program and easy to design.
- Sequential file is best use if storage space.

Disadvantages of sequential file

- Sequential file is time consuming process.
- It has high data redundancy.
- Random searching is not possible.

2. **Direct access file organization**

- Direct access file is also known as **random access or relative file organization.**
- In direct access file, all records are stored in direct access storage device (DASD), such as hard disk. The records are randomly placed throughout the file.
- The records does not need to be in sequence because they are updated directly and rewritten back in the same location.
- This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.
- It is also called as **hashing**.

Advantages of direct access file organization

- Direct access file helps in online transaction processing system (OLTP) like online railway reservation system.
- In direct access file, sorting of the records are not required.
- It accesses the desired records immediately.
- It updates several files quickly.
- It has better control over record allocation.

Disadvantages of direct access file organization

- Direct access file does not provide back up facility.
- It is expensive.
- It has less storage space as compared to sequential file.

3**. Indexed sequential access file organization**

- Indexed sequential access file combines both sequential file and direct access file organization.
- In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.
- This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.
- The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.

Advantages of Indexed sequential access file organization

- In indexed sequential access file, sequential file and random file access is possible.
- It accesses the records very fast if the index table is properly organized.
- The records can be inserted in the middle of the file.
- It provides quick access for sequential and direct processing.
- It reduces the degree of the sequential search.

Disadvantages of Indexed sequential access file organization

- Indexed sequential access file requires unique keys and periodic reorganization.
- Indexed sequential access file takes longer time to search the index for the data access or retrieval.
- It requires more storage space.
- It is expensive because it requires special software.
- It is less efficient in the use of storage space as compared to other file organisation

ACCESS METHODS

A file organization refers to the organization of the data of a file into records, blocks and access structures; this includes the way the records and blocks are placed on the storage medium and interlinked. An access method on the other hand, provides a group of operations – such as **(i) find, (ii) read, (iii) modify, (iv) delete** etc., — that can be applied to a file. In general, it is possible to apply several access methods to a file organization. Some access methods, though, can be applied only to files organised in certain ways.

1. Sequential Access Method (SAM)

In sequential files, the records are stored in a predefined order. Records which occur in a sequential file are usually sorted on the primary key and physically arranged on the storage medium in order by primary key. If only sequential access is required (which is rarely the case), sequential media (magnetic tapes) are suitable and probably the most cost-effective way of processing such files.

Sequential access is fast and efficient while dealing with large volumes of data that need to be processed periodically. However, it is require that all new transactions be sorted into a proper sequence for sequential access processing. Also, most of the database or file may

have to be searched to locate, store, or modify even a small number of data records. Thus, this method is too slow to handle applications requiring immediate updating or responses. Sequential files are generally used for backup or transporting data to a different system. A sequential ASCII file is a popular export/import format that most database systems support.

2. Indexed Sequential Access Method (ISAM)

In indexed sequential files, record occurrences are sorted and stored in order by primary key on a direct access storage device. In addition, a separate table (or file) called an index is maintained on **primary key** values to give the physical address of each record occurrence. This approach gives (almost) direct access to record occurrences via the index table and sequential access via the way in which the records are laid out on the storage medium. The physical address of a record given by the index file is also called a **poin**ter. The pointer or address can take many forms depending on the operating system and the database one is using. Today systems use virtual addresses instead of physical addresses. A virtual address could be based on imaginary disk drive layout. The database refers to a base set of tracks and cylinders. The computer then maps these values into actual storage locations. This arrangement is the basis for an approach known as the **virtual sequential access method** (VSAM). Another common approach is to define a location in terms of its distance from the start of a file (**relative address**). Virtual or relative addresses are always better than the physical address because of their portability. In case a few records need to be processed quickly, the index is used to directly access the records needed. However, when large numbers of records must be processed periodically, the sequential organization provided by this method is used

3. Direct Access Method (DAM)

When using the direct access method, the record occurrences in a file do not have to be arranged in any particular sequence on storage media. However, the computer must keep track of the storage location of each record using a variety of direct organization methods so that data is retrieved when needed. New transactions data do not have to be sorted, and processing that requires immediate responses or updating is easily

handled. In the direct access method, an algorithm is used to compute the address of a record. The primary key value is the input to the algorithm and the block address of the record is the output. Address Data/Record Key Value Address/Pointer Index file Data file and Database Design To implement the approach, a portion of the storage space is reserved for the file. This space should be large enough to hold the file plus some allowance for growth. Then the algorithm that generates the appropriate address for a given primary key is devised. The algorithm is commonly called **hashing algorithm**. The process of converting primary key values into addresses is called **key-to-address transformation.** More than one logical record usually fits into a block, so we may think of the reserved storage area as being broken into record slots sequentially numbered from 1 to n. These sequential numbers are called relative pointers or relative addresses, because they indicate the position of the record relative to the beginning of the file. **The objective of the hashing algorithm is to generate relative addresses that disperse the records throughout the reserved storage space in a random but uniform manner.** The records can be retrieved very rapidly because the address is computed rather than found through table lookup via indexes stored on a disk file**. A collision is said to occur if more than one record maps to the same block.** Since one block usually holds several records, collisions are only a problem when the number of records mapping to a block exceeds the block's capacity. To account for this event, most direct access methods support an overflow area for collisions which is searched sequentially. The hashed key approach is extremely fast since the key's value is immediately converted into a storage location, and data can be retrieved in one pass to the disk

What is Hashing?

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.

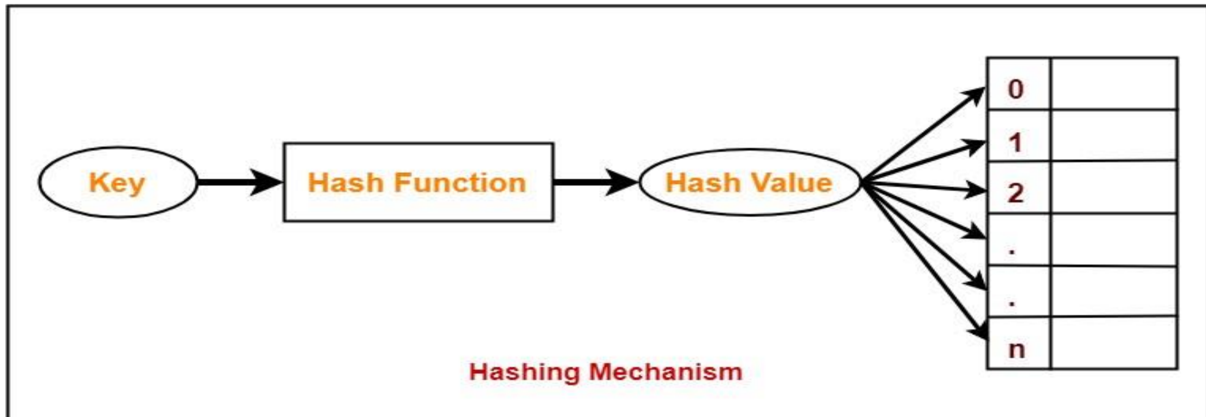
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.
- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.

What is Hash Function?

- A fixed process converts a key to a hash key is known as a **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

Hash Key Value

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be stored in the hash table.
- Hash key value is generated using a hash function.



What is Hash Table?

Hash Table is a data structure which stores data in an associative manner. In a **hash table**, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.

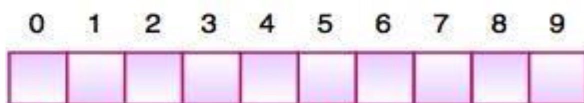


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.

- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.

Types of Hash Functions-

There are various types of hash functions available such as-

1. Mid Square Hash Function
2. Division Hash Function
3. Folding Hash Function
4. Digit Analysis

1. Mid Square method

Here the key K is squared and then the hash function H is defined by

H(K) = l where l is obtained by deleting from both ends of K^2

Example: K = 3205	2345	7148
K^2 = 10272025	5499025	51093904
H(K) = 72	99	93

The fourth and fifth digits, counting from right are chosen for the hash address.

2.Division Hash Function

Here choose a number m larger than the number n of the keys in K. The hash function H is defined by

H(K) = K(mod m) (Here range is from 0 to m-1) or H(K) = K(mod m) +1 (Here range is from 1 to m) Here K(mod m) denotes the remainder when K is divided by m.

Example: let m=97,then

H(3205) = 3205(mod 97) = 4 (remainder)

H(2345) = 2345(mod 97) = 17 (remainder)

H(7148) = 7148(mod 97) = 67 (remainder)

3.Folding Hash Function

Here the key K is partitioned into a number of parts $k_1,k_2,k_3....k_r$ where each part except possibly the last has the same number of digits as required address. Then the parts are added together ignoring the last carry. That is,

$H(K) = k_1+k_2+k_3+....+ k_r$

H(3205) = 32+05 = 37
H(2345) = 23+45 = 68
H(7148) = 71+48 = 19 (ignore carry)

4.Digit Analysis

It forms the address by selecting and shifting digits or bits of the original key. this hashing function is in a sense distributed dependent.

Example: let key K 7546123 is transformed to the address 2164 by selecting digits in positions 3 to 6 and reverse their order.

Characteristics of good hashing function

1. The hash function should generate different hash values for the similar string.
2. The hash function is easy to understand and simple to compute.
3. The hash function should produce the keys which will get distributed, uniformly over an array.
4. A number of collisions should be less while placing the data in the hash table.
5. The hash function is a perfect hash function when it uses all the input data.

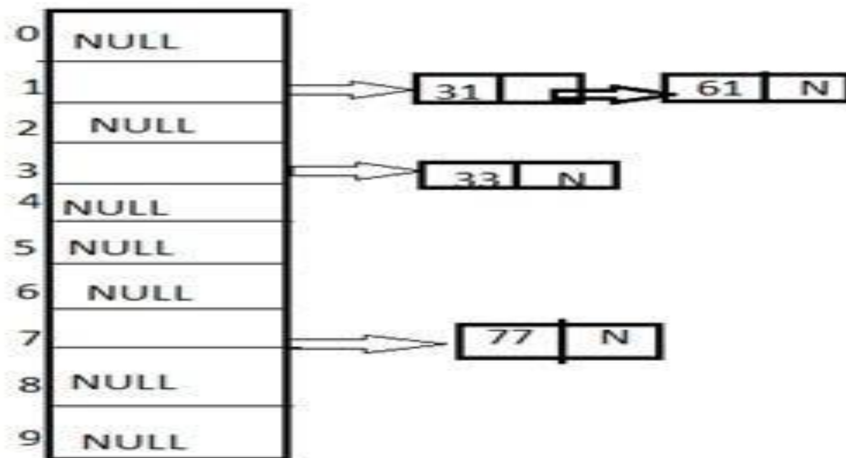
Collision: It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

Collision resolution technique

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below

1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

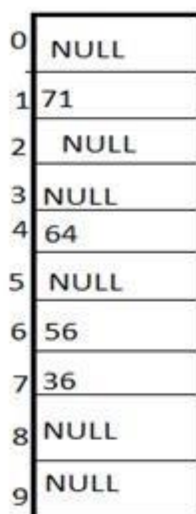


Example: Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are 31,33,77,61. In the above diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

2) Linear probing

It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.



In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

3) Quadratic probing

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the H(key)=(H(key)+x*x)%table size. Let us consider we have to insert following elements that are:-67, 90,55,17,49.

0	90
1	
2	
3	
4	
5	55
6	
7	67
8	17
9	49

In this we can see if we insert 67, 90, and 55 it can be inserted easily but at case of 17 hash function is used in such a manner that :-(17+0*0)%10=17 (when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so (17+1*1)%10=8.in this case bucket 8 is empty hence we will place 17 at index 8.

4) Double hashing

It is a technique in which two hash function are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

1. It must never evaluate to zero.
2. Must sure about the buckets, that they are probed.

The hash functions for this technique are:

```
H1(key)=key % table size
H2(key)=P-(key mod P)
```

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

Example: Let us consider we have to insert 67, 90,55,17,49.

Index	Value
0	90
1	17
2	
3	
4	
5	55
6	
7	67
8	
9	49

In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 again the bucket is full and in this case we have to use the second hash function which is H2(key)=P-(key mode P) here p is a prime number which should be taken smaller than the hash table so value of p will be the 7.

i.e. H2(17)=7-(17%7)=7-3=4 that means we have to take 4 jumps for placing the 17. Therefore 17 will be placed at index 1.