Data Structures and Algorithms

S.PRABHAVATHI

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SEIENCE & IT

JAMAL MOHAMED COLLEGE (A)

TRICHY – 620 020

Unit: I

Introduction and Overview: Basic Terminology – Data Structures – Data Structure Operations –Mathematical Notations and Functions – Control Structures – Algorithms: Time-space Trade-off –Complexity of Algorithms – Asymptotic Notations – Arrays – Introduction – Linear Array, Representation of Linear Array in Memory, Traversing Linear Arrays, Inserting and Deleting, Two Dimensional Arrays – Representation of Two Dimensional Array in Memory.

Data Structures:

A **data structure** is a specialized format for organizing, processing, retrieving and storing data.Data structures refer to data and representation of data objects within a program, that is, the implementation of structured relationships. A data structure is a collection of atomic and composite data types into a set with defined relationships.

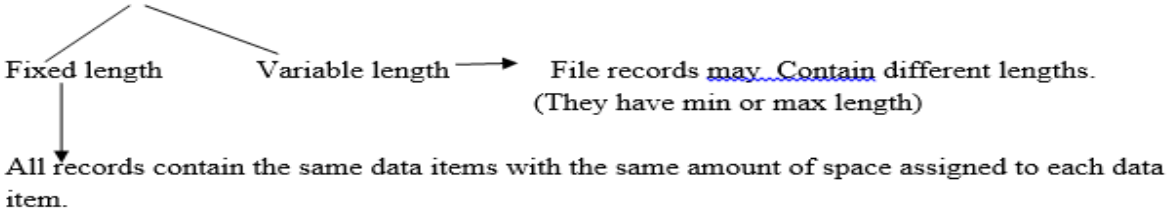
The term Data Structure refers to the organization of data elements and the interrelationships among them.

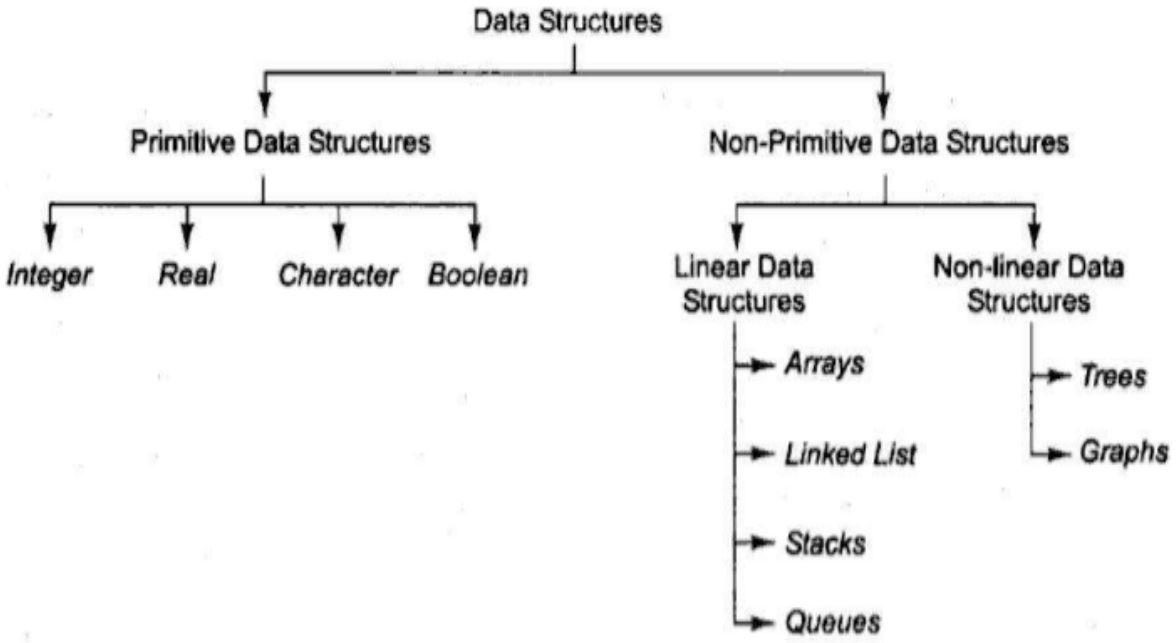
Data Structure is a way to store and organize data so that it can be used efficiently.

BASIC TERMINOLOGY, ELEMENTARY DATA ORGANIZATION:

1. DATA- raw fact or values or set of values
2. DATA ITEM - single unit of values
3. DATA ITEM are of 2 types.
   They are: Group item, Elementary item.
   Group items divided into sub items e.g.: Emp name: first name, middle initial, last name.
   Elementary items treated as single item, not divided e.g.: emp no.
4. ENTITY- Attributes or properties with assigned values. Values may be numeric / non numeric.
   E.g.: Attributes : Name Age Sex
   Values : ROHINI 34 F
5. ENTITY SET-Entities with similar attributes.
   E.g.: All employee in an organisation.
6. INFORMATION- Data with given attributes or processed data.
7. FIELD- Single elementary unit of information representing an attributes of an entity.
8. RECORD- Collection of field values of a given entity.
9. FILES- Collection of records of the entities in a given entity set.
10. PRIMARY KEY- Value in a certain field which uniquely determines the record in the file.
    File K is called primary key and values $k_1, k_2, \ldots$ called keys or key values.

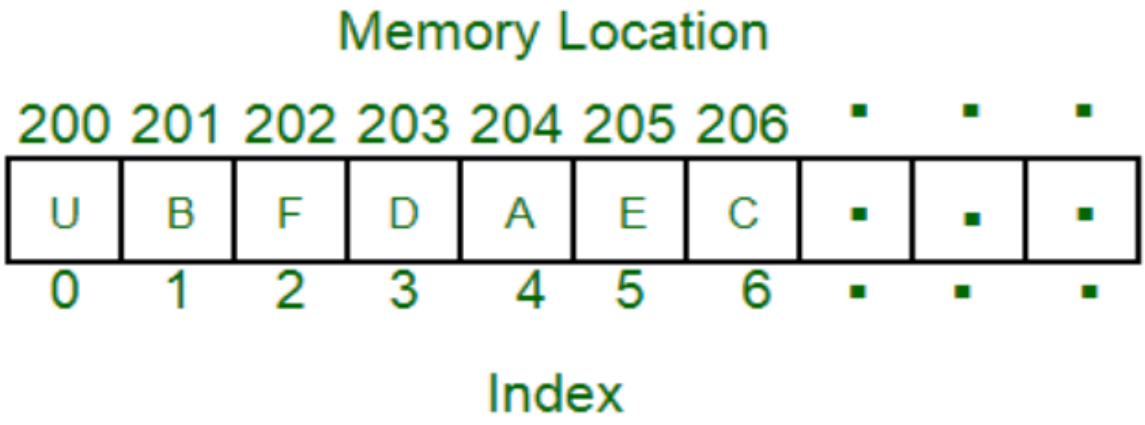
11. RECORDS





LINEAR Data Structure Array Data Structure:

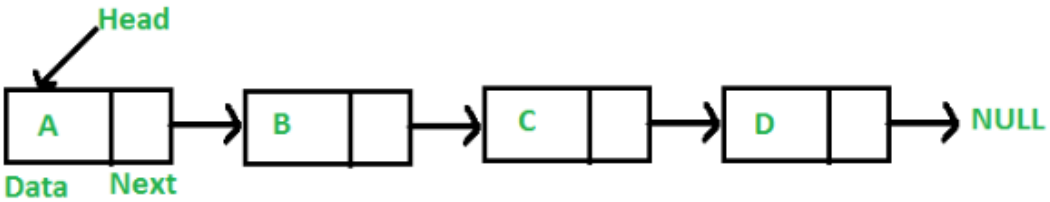
An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array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



The above image can be looked as a top-level view of a staircase where you are at the base of the staircase. Each element can be uniquely identified by their index in the array (in a similar way as you could identify your friends by the step on which they were on in the above example).

Linked List Data Structure

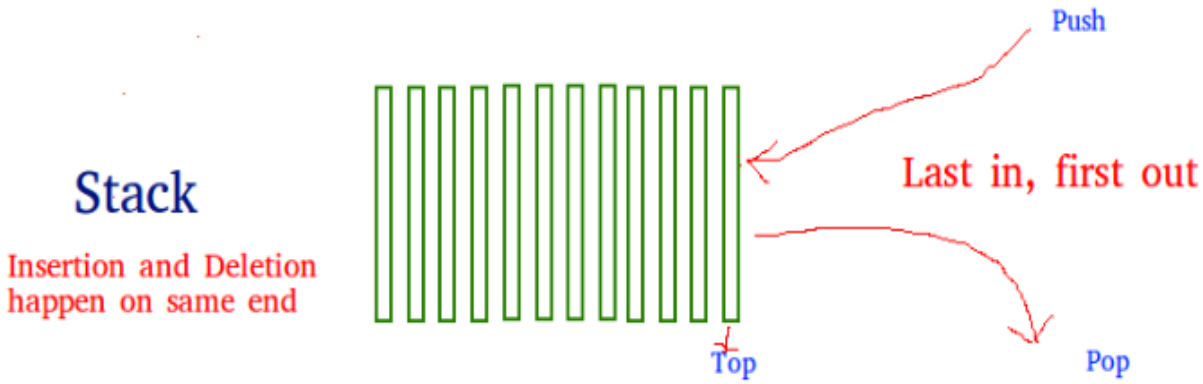
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Stack Data Structure

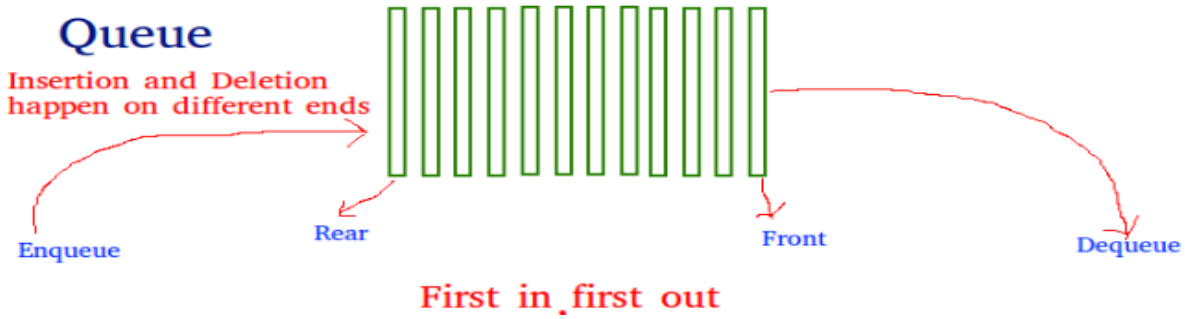
Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).



There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

Queue Data Structure

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

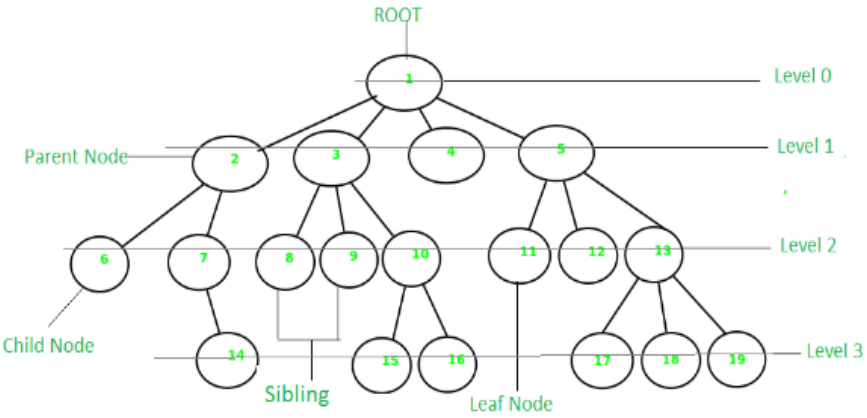


NON-LINEAR Data Structure

Tree Data Structure And Algorithms

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

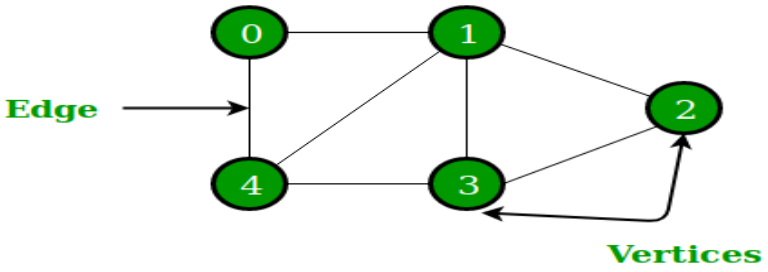
This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



Graph Data Structure And Algorithms

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. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of node



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}.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

DATA STRUCTURE OPERATIONS

FOUR MAJOR OPERATIONS

TRAVERSING/VISITING:

- Accessing each record exactly once so that certain items in the record may be processed.
- SEARCHING:

Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.

- INSERTING: Adding a new record to the structure.
- DELETION: Removing a record from the structure.
- OPERATIONS USED IN SPECIAL SITUATIONS
- SORTING: Arranging the records in some logical order. (Numerical/Alphabetical)
- MERGING:Combining records in two different sorted files into a single sorted file.
- OTHER OPERATIONS

Copying , concatenation , etc.

Mathematical notations and functions

1. Floor and Ceiling Functions:

If x is a real number, then it means that x lies between two integers which are called the floor and ceiling of x. i.e.

```
|_x_| is called the floor of x. It is the greatest integer that is not greater than x.
|  x  | is called the ceiling of x. It is the smallest integer that is not less than x
```

If x is itself an integer, then |_x_| = | x |, otherwise |_x_| + 1 = | x | E.g.

```
|_3.14_| = 3, |_-8.5_| = -9, |_7_| = 7

|  3.14  |= 4, |  -8.5  | = -8, |   7   |= 7
```

2. Remainder Function (Modular Arithmetic):

If k is any integer and M is a positive integer, then:

```
k (mod M)
```

gives the integer remainder when k is divided by M.
E.g.

```
25(mod 7) = 4
25(mod 5) = 0
```

3. Integer and Absolute Value Functions:

If x is a real number, then integer function INT(x) will convert x into integer and the fractional part is removed.
E.g.

```
INT (3.14) = 3
INT (-8.5) = -8
```

The absolute function ABS(x) or | x | gives the absolute value of x i.e. it gives the positive value of x even if x is negative.
E.g.

```
ABS(-15) = 15 or ABS | -15| = 15
ABS(7) = 7 or ABS | 7 | = 7
ABS(-3.33) = 3.33 or ABS | -3.33 | = 3.33
```

4. Summation Symbol (Sums):

The symbol which is used to denote summation is a Greek letter Sigma ?.
Let a1, a2, a3, ... , an be a sequence of numbers. Then the sum a1 + a2 + a3 + ... + an will be written as:

```
n
? aj
j=1
```

where j is called the dummy index or dummy variable.
E.g.

```
n
? j = 1 + 2 + 3 +…..+ n
j=1
```

5. Factorial Function:

n! denotes the product of the positive integers from 1 to n. n! is read as 'n factorial', i.e.

```
n! = 1 * 2 * 3 * ….. * (n-2) * (n-1) * n
```

E.g.

```
4! = 1 * 2 * 3 * 4 = 24
5! = 5 * 4! = 120
```

6. Permutations:

Let we have a set of n elements. A permutation of this set means the arrangement of the elements of the set in some order. E.g.

Suppose the set contains a, b and c. The various permutations of these elements can be: abc, acb, bac, bca, cab, cba.

If there are n elements in the set then there will be n! permutations of those elements. It means if the set has 3 elements then there will be 3! = 1 * 2 * 3 = 6 permutations of the elements.

7. Exponents and Logarithms:

Exponent means how many times a number is multiplied by itself. If m is a positive integer, then:

```
am = a * a * a * ….. * a (m times)
and
a-m = 1 / am
E.g.
24 = 2 * 2 * 2 * 2 = 16
2-4 = 1 / 24 = 1 / 16
```

The concept of logarithms is related to exponents. If b is a positive number, then the logarithm of any positive number x to the base b is written as logbx. It represents the exponent to which b should be raised to get x i.e. y = logbx and by = x
E.g.

```
  log28 = 3, since 23=8

log100.001 = - 3, since 10-3
= 0.001
logb1 = 0, since b0
= 1
logbb = 1, since b1
= b
```

2.4 CONTROL STRUCTURES

Algorithms and their equivalent computer programs are more easily understood if they mainly use self-contained modules and three types of logic, or flow of control, called

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

These three types of logic are discussed below, and in each case we show the equivalent flowchart.

Sequence Logic (Sequential Flow)

Sequence logic has already been discussed. Unless instructions are given to the contrary, the modules are executed in the obvious sequence. The sequence may be presented explicitly, by means of numbered steps, or implicitly, by the order in which the modules are written. (See Fig. 2.3.) Most processing, even of complex problems, will generally follow this elementary flow pattern.

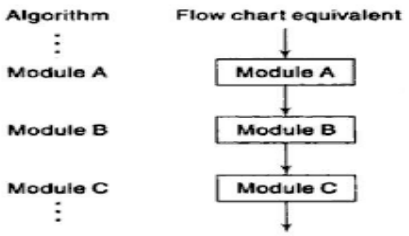


Fig. 2.3 Sequence Logic

Selection Logic (Conditional Flow)

Selection logic employs a number of conditions which lead to a selection of one out of several alternative modules. The structures which implement this logic are called conditional structures or If structures. For clarity, we will frequently indicate the end of such a structure by the statement

[End of If structure.]

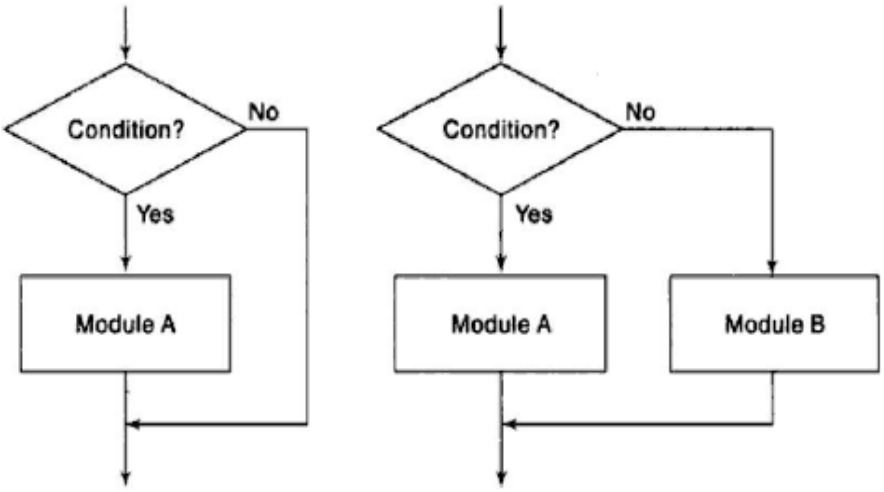
or some equivalent.

These conditional structures fall into three types, which are discussed separately.

1. **Single Alternative.** This structure has the form

If condition, then:
[Module A]
[End of If structure.]

The logic of this structure is pictured in Fig. 2.4(a). If the condition holds, then Module A, which may consist of one or more statements, is executed; otherwise Module A is skipped and control transfers to the next step of the algorithm.



(a) Single alternative. (b) Double alternative.

Fig. 2.4

2. **Double Alternative.** This structure has the form

```
If condition, then:
    [Module A]
Else:
    [Module B]
[End of If structure.]
```

The logic of this structure is pictured in Fig. 2.4(b). As indicated by the flow chart, if the condition holds, then Module A is executed; otherwise Module B is executed.

3. **Multiple Alternatives.** This structure has the form

```
If condition(1), then:
    [Module A1]
Else if condition(2), then:
    [Module A2]
        ⋮
Else if condition(M), then:
    [Module AM]
Else:
    [Module B]
[End of If structure.]
```

The logic of this structure allows only one of the modules to be executed. Specifically, either the module which follows the first condition which holds is executed, or the module which follows the final Else statement is executed. In practice, there will rarely be more than three alternatives.

Iteration Logic (Repetitive Flow)

The third kind of logic refers to either of two types of structures involving loops. Each type begins with a Repeat statement and is followed by a module, called the *body of the loop*. For clarity, we will indicate the end of the structure by the statement

[End of loop.]

or some equivalent.

Each type of loop structure is discussed separately.

The *repeat-for loop* uses an index variable, such as K, to control the loop. The loop will usually have the form:

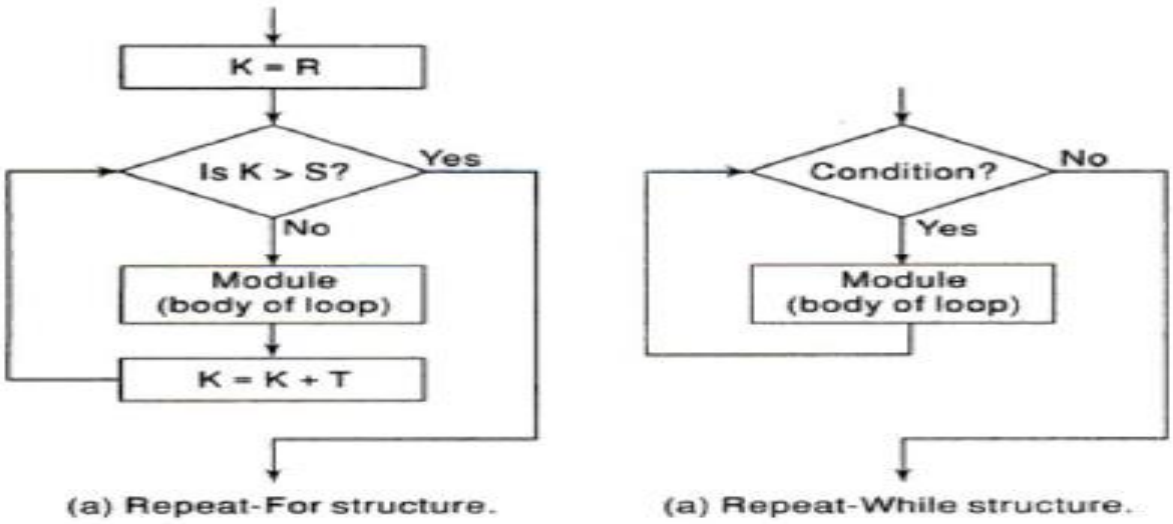
```
Repeat for K = R to S by T:
    [Module]
[End of loop.]
```

The logic of this structure is pictured in Fig. 2.5(a). Here R is called the *initial value*, S the *end value* or *test value*, and T the *increment*. Observe that the body of the loop is executed fitst with K = R, then with K = R + T, then with K = R + 2T, and so on. The cycling ends when K > S. The flow chart assumes that the increment T is positive; if T is negative, so that K decreases in value, then the cycling ends when K < S.

Algorithm 2.1 is rewritten using a repeat-while loop rather than a Go to statement:

Algorithm 2.3: (Largest Element in Array) Given a nonempty array DATA with N numerical values, this algorithm finds the location LOC and the value MAX of the largest element of DATA.

1. [Initialize.] Set K := 1, LOC := 1 and MAX := DATA[1].
2. Repeat Steps 3 and 4 while K ≤ N:
3. If MAX < DATA[K], then:
   Set LOC := K and MAX := DATA[K].
   [End of If structure.]
4. Set K := K + 1.
   [End of Step 2 loop.]
5. Write: LOC, MAX.
6. Exit.



(a) Repeat-For structure. (a) Repeat-While structure.

Fig. 2.5

The *repeat-while loop* uses a condition to control the loop. The loop will usually have the form

Repeat while condition:
[Module]
[End of loop.]

The logic of this structure is pictured in Fig. 2.5(b). Observe that the cycling continues until the condition is false. We emphasize that there must be a statement before the structure that initializes the condition controlling the loop, and in order that the looping may eventually cease, there must be a statement in the body of the loop that changes the condition.

Algorithm 2.3 indicates some other properties of our algorithms. Usually we will omit the word "Step." We will try to use repeat structures instead of Go to statements. The repeat statement may explicitly indicate the steps that form the body of the loop. The "End of loop" statement may explicitly indicate the step where the loop begins. The modules contained in our logic structures will normally be indented for easier reading. This conforms to the usual format in structured programming.

Any other new notation or convention either will be self-explanatory or will be explained when it occurs.

1.6 ALGORITHMS: COMPLEXITY, TIME-SPACE TRADEOFF

An algorithm is a well-defined list of steps for solving a particular problem. One major purpose of this text is to develop efficient algorithms for the processing of our data. The time and space it uses are two major measures of the efficiency of an algorithm. The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size. (The notion of complexity will be treated in Chapter 2.)

Each of our algorithms will involve a particular data structure. Accordingly, we may not always be able to use the most efficient algorithm, since the choice of data structure depends on many things, including the type of data and the frequency with which various data operations are applied. Sometimes the choice of data structure involves a time-space tradeoff: by increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data, or vice versa. We illustrate these ideas with two examples.

Searching Algorithms

Consider a membership file, as in Example 1.6, in which each record contains, among other data, the name and telephone number of its member. Suppose we are given the name of a member and we want to find his or her telephone number. One way to do this is to linearly search through the file, i.e., to apply the following algorithm:

Linear Search

Search each record of the file, one at a time, until finding the given Name and hence the corresponding telephone number.

First of all, it is clear that the time required to execute the algorithm is proportional to the number of comparisons. Also, assuming that each name in the file is equally likely to be picked, it is intuitively clear that the average number of comparisons for a file with n records is equal to $n/2$; that is, the complexity of the linear search algorithm is given by $C(n) = n/2$.

The above algorithm would be impossible in practice if we were searching through a list consisting of thousands of names, as in a telephone book. However, if the names are sorted alphabetically, as in telephone books, then we can use an efficient algorithm called binary search. This algorithm is discussed in detail in Chapter 4, but we briefly describe its general idea below.

Binary Search

Compare the given Name with the name in the middle of the list; this tells which half of the list contains Name. Then compare Name with the name in the middle of the correct half to determine which quarter of the list contains Name. Continue the process until finding Name in the list.

One can show that the complexity of the binary search algorithm is given by

$$C(n) = \log_2 n$$

Thus, for example, one will not require more than 15 comparisons to find a given Name in a list containing 25 000 names.

Although the binary search algorithm is a very efficient algorithm, it has some major drawbacks. Specifically, the algorithm assumes that one has direct access to the middle name in the list or a sublist. This means that the list must be stored in some type of array. Unfortunately, inserting an element in an array requires elements to be moved down the list, and deleting an element from an array requires element to be moved up the list.

The telephone company solves the above problem by printing a new directory every year while keeping a separate temporary file for new telephone customers. That is, the telephone company updates its files every year. On the other hand, a bank may want to insert a new customer in its file almost instantaneously. Accordingly, a linearly sorted list may not be the best data structure for a bank.

An Example of Time-Space Tradeoff

Suppose a file of records contains names, social security numbers and much additional information among its fields. Sorting the file alphabetically and running a binary search is a very efficient way to find the record for a given name. On the other hand, suppose we are given only the social security number of the person. Then we would have to do a linear search for the record, which is extremely time-consuming for a very large number of records. How can we solve such a problem?

Analysing Algorithm (or) Complexity of Algorithm: To select the best algorithms form the available algorithms we have to analyze the algorithms.

(a). Execution time (b).Memory space.

Space complexity: A space complexity of an algorithms is the amount of memory that an algorithm needs to run the program.

Eg: 1
```
Algorithm xyz(x,y,z)
{
 return(x+y+z*y+(x+y-z)) / ((x +y) +40);
}
```

Eg: 2
```
Algorithm(x,n)
{
total=0.0
for i=1 to n do
total=total+(x(i))
return total
}
```

Here a space complexity of an algorithm is add (x,n) because

(i) To store x(i) we need n locations (ii) to store i, total n we need 3 locations.

Time complexity: It is the amount of computer time for a program needs to run its completion.

Eg:1
```
Algorithm add (x,n)
{
Total =0.0;
For I =1to n do
Total= total + x(i);   Return total;}
```

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

There are mainly three asymptotic notations:

1. Big-O notation
2. Omega notation
3. Theta notation

1. Big oh notation (O):

It is define as upper bound and upper bound on an algorithm is the most amount of time required (the worst case performance).
Big oh notation is used to describe **asymptotic upper bound**.

Mathematically, if **f(n)** describes the running time of an algorithm; **f(n)** is **O(g(n))** if there exist positive constant **C** and **n0** such that,

$$0 <= f(n) <= Cg(n) \text{ for all } n >= n0$$

n = used to give upper bound an a function.
If a function is **O(n)**, it is automatically **O(n-square)** as well.

2. Big Omega notation (Ω) :

It is define as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).
Just like **O notation** provide an **asymptotic upper bound**, **Ω notation** provides **asymptotic lower bound**.

Let **f(n)** define running time of an algorithm;
f(n) is said to be **Ω(g (n))** if there exists positive constant **C** and **(n0)** such that

$$0 <= Cg(n) <= f(n) \text{ for all } n >= n0$$

n = used to given lower bound on a function
If a function is **Ω(n-square)** it is automatically **Ω(n)** as well.

3. Big Theta notation (Θ) :

It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.

Let **f(n)** define running time of an algorithm.
f(n) is said to be **Θ(g(n))** if **f(n)** is **O(g(n))** and **f(n)** is **Ω(g(n))**.

Mathematically,

0 <= f(n) <= C1g(n) for n >= n0
0 <= C2g(n) <= f(n) for n >= n0

Merging both the equation, we get :

0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0

The equation simply means there exist positive constants C1 and C2 such that f(n) is sandwich between C2 g(n) and C1g(n).

ARRAY:

Array:-

An array is defined as an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations in RAM. The elements of an array are of same data type and each item can be accessed using the same name.

Declaration of an array:- We know that all the variables are declared before they are used in the program. Similarly, an array must be declared before it is used. During declaration, the size of the array has to be specified. The size used during declaration of the array informs the compiler to allocate and reserve the specified memory locations.

INTRODUCTION:

- Data structures are classified as either linear or non – linear.
- The linear relationship between the elements represented by means of sequential memory locations. These linear structures are called arrays.
- The operations performed in an array are,
- Traversal – Processing each element in the list.
- Search – Finding the location of the element with a given value or the record with a given key.
- Insertion – Adding a new element to the list.
- Deletion – Removing an element from the list.
- Sorting – Arranging the elements in an order.
- Merging – Combining two lists into a single list.

LINEAR ARRAYS:

- A linear array is a list of a finite number n of homogenous data elements (data elements of same type).
- The elements of the array are referenced by an index set consisting of n consecutive numbers.
- The elements of the array are store respectively in successive memory locations.
- The number n of elements is called the length or size of the array.
- The index set consists of the integers 1, 2,…. ,n.
- 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

- UB – largest index called upper bound.
- LB – smallest index called lower bound.

  Length = UB when LB = 1

- The elements of an array A can be denoted by:
  - Subscript notation : A1,A2,A3,….An
  - Paranthesis notation : A(1),A(2),A(3)…..A(N)
  - Bracket notation : A[1],A[2],A[3]….A[N]
- Example: Let DATA be a 6 – element linear array of integers such that
  - DATA [1] = 47 DATA [2] = 56 DATA [3] = 29
  - DATA [4] = 35 DATA [5] = 87 DATA [6] = 15
- Array can be represented by

DATA

1	47
2	56
3	29
4	35
5	87
6	15

47	56	29	35	87	15
1	2	3	4	5	6

REPRESENTATION OF LINEAR ARRAYS IN MEMORY:

- Let LA be a linear array in memory of the computer

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

ALLOCATION OF ARRAY:

STATIC → Compile time during program execution.

DYNAMIC → read n, then declare array with n elements.

LENGTH OR SIZE OF THE ARRAY: Numbers of data elements of the array.

LENGTH=UB-LB+1

UB-upper bound LB- lower bound

E.g.: DATA

249	56	429	135	87
1	2	3	4	5

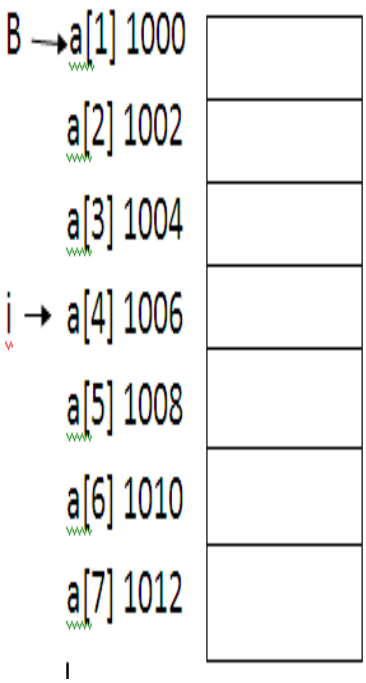
Length 5-1+1=5

LOCATION: Address of any element of a linear array LA.

$$a[i] = B + (i - LB) * s$$

B- Base address (i.e.) address of the first element of LA.

S- Size or number of words per memory cell for the array.



size s=2 bytes B=1000 i=4 LB=1

a[4] = 1000 + (4-1)*2

=1000+3*2

=1006

- The elements of LA are stored in memory cells.
- The computer keep track only of the address of the first element of LA , denoted by Base(LA) called the base address of LA.
- Using this the address is calculated. The formula is

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

- w = number of words per memory cell for the array LA.
- Example: Consider the array DATA , Base(DATA)=200 and w=4 words per memory cell for DATA.

  Then LOC(DATA[47])=200

  LOC(DATA[56]=204

  LOC(DATA[29]=208….

TRAVERSING LINEAR ARRAYS:

- Accessing and processing (visiting) each element of array exactly once.
- Algorithm using repeat while loop.
  - [Initialize counter] set K=LB
  - Repeat steps 3 and 4 while K UB
  - [visit element]apply PROCESS to LA[K].
  - [Increase counter] set K=KH

- [End of step 2 loop].
- Exit.
- Algorithm using repeat for loop
- Repeat for K=LB to UB
- Apply PROCESS to LA[K]
- [End of loop]
- Exit

E.g.: array AUTO records the number of automobiles sold each year from 1932 through 1984.

a) Find the number NUM of years during which more than 300 automobiles were sold.

1. [Initialization step] set NUM=0
2. Repeat for K=1932 to 1984
   If AUTO [k] > 300 then set NUM =NUM+1
   End of loop.
3. Return

b) Print each year and the number of automobiles sold in that year.

1. Repeat for K=1932 to 1984
   Write K , AUTO [K].
   [End of loop]
2. Return.

INSERTING AND DELETING:

INSERTING A NEW ELEMENT TO THE LIST

- Let A be a collection of data element in memory. Inserting refers to adding another element to the collection A.
- Inserting an element at the end can be done easily provided memory space allocated is larger enough to accommodate the additional element.
- Inserting an element in middle requires on a average half of the elements to be moved downward to new locations to accommodate the new element and keep the order of elements.

Algorithm INSERT (LA , N , K , ITEM)

1. [Initialize counter] set J = N.
2. Repeat steps 3 and 4 while J ≥ K.
3. [More J_{th} element downward] set LA[J+1]=LA[J]
4. [Decrease counter] set J=J-1
   [End of step 2 loop]
5. [Insert element] set LA[K]= ITEM
6. [Reset N] set N=N+ 1
7. Exit

E.g.:

1	Brown
2	Daris
3	John
4	Smith
5	Wagner
6	
7	
8	

Insert Ford

∴ $K = 3$

1	Brown
2	Daris
3	Ford
4	John
5	Smith
6	Wagner
7	
8	

Algorithm steps (1) J=5

(2)While J ≥ K (i.e. 5 ≥ 3) loop

6 ← 5 5 ← 4 4 ← 3

J=4 J=3 J=2 loop ends.

∴ 3rd position is now free for item to be inserted.

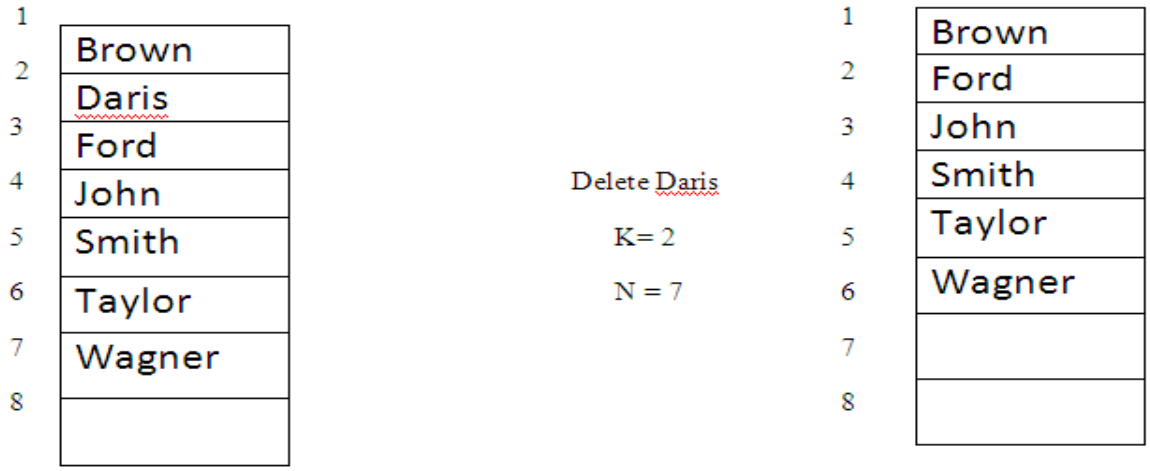
DELETING AN ITEM FROM THE LIST

- 'Deleting ' refers to the operation of removing one of the elements from the location.
- Deleting an element at the 'end' of an array is not difficult.
- Deleting an element in middle requires subsequent elements to be moved one location upward to 'fill up' the array.

Algorithm DELETE (LA,N,K,ITEM)

1. Set ITEM =LA(K)
2. Repeat for J=K to N-1.
   [more J+1st element upward] set LA(J)=LA(J+1)
   [End of loop]
3. [Reset the number N of elements in LA, set N=N-1.
4. Exit.

E.g.:



Algorithm steps

(1) LA(K) = 2

(2) J = 2

2 ← 3

3 ← 4

4 ← 5

5 ← 6

6 ← 7

(3) N = N-1 = 7-1 =6. Daris deleted.

Two - Dimension Array:

A two-dimension m x n array, A is it collection of m.n data elements such that each elements is specified by pair of integers (such as J,K) called subscripts, with the property that,

1<= J <= m , 1<= K = n

There is a standard way of drawing a two-dimensional m x n array A where, the elements of A form a rectangular array with m rows and n columns and where, the elements A[J.K] < appears in row J and column K,

Suppose, A is a two-dimensional m x n array. The first dimension of A contains the index set 1, ... m with lower bound 1 and upper bound m, and the second dimension of A contains the index set 1, 2, ... n with lower bound 1 and upper bound n. The length of a dimension is the n0 of integers in its index set. The pair of lengths **m x n is called size of the array**.

Programming language allows one to define multi-dimensional array in which lower bounds are not 1. However, the index set for each dimension is still consists of consecutive integers from the lower bound to the upper bound of the dimension. The length of a given dimension can be obtained from the formula,

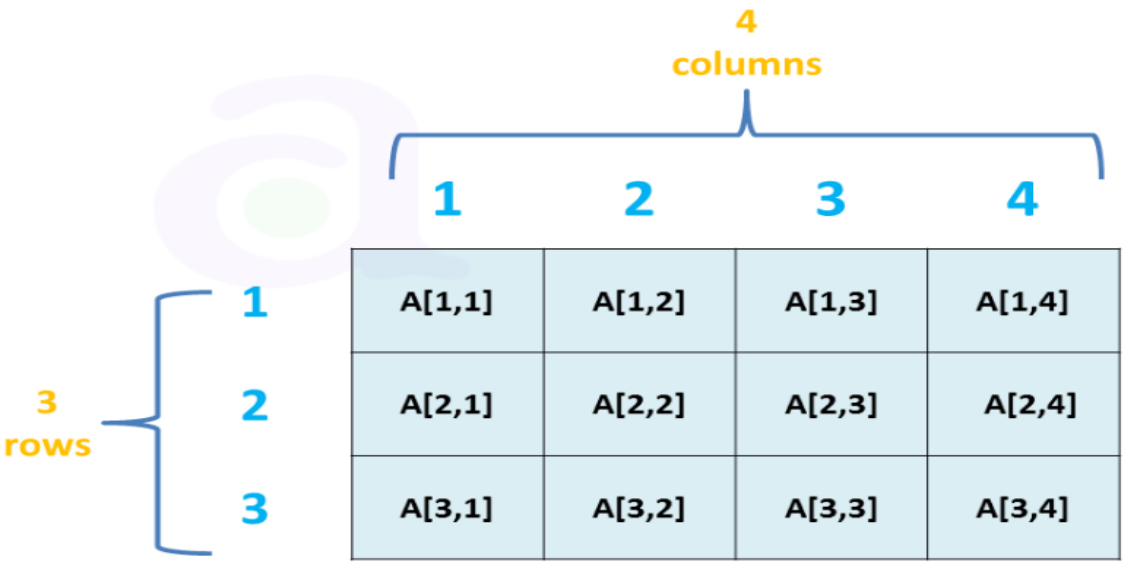


Fig: Two-dimension 3x4 array

Length = UB - LB + 1

For two-dimensional array we can find the length of the array by the following procedure,

First we will find the length of one-dimensional row array by the formula,

$L_1 = UB_1 - LB_1 + 1$

Do this again one-dimensional column array.

$L_2 = UB_2 - LB_2 + 1$

Length of array (number of elements in two-dimensional array)

$L = L_1 x\ L_2$

Example

Given array int A(2 : 5, - 3 : 1)

The lengeh of row one-dimensional array (5 - 2 +1 = 4)

The length of column one-dimensional array (1 - (-3) +1=5)

Length of given array = 4 x 5 = 20

So, there are 20 elements on the given array.

Representation of two-dimensional array in memory:

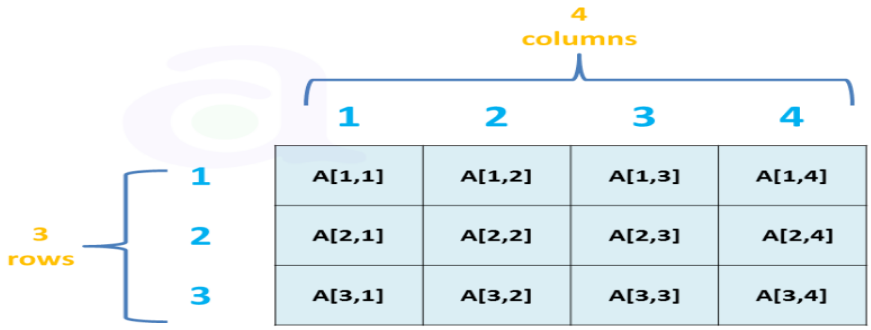


Fig: Two-dimension 3x4 array

- Let, A be a two-dimensional array m x n array. Although A is pictured as a rectangular array of elements with m rows and n columns, the array will be represented in memory by a block m.n sequential memory location. Specifically, the programming language will store the array A either,
- column by column is called **column major order** or
- Row by row, in **row major order**
- Given Fig. 3. shows these two ways when *A is a* two-dimensional 3 x 4 array. We emphasize that the particular representation used depends upon the programming language, not the user.
- Recall that, for a linear array, the computer does not keep track of the address LOC(LA[K]) of every element LA[K] of LA, but does keep track of Base (LA) the address of the first element of LA. The computer uses the formula.

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

For 3x4 array Column Major Order (CMO)

	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Concept:
Column Major Order (CMO)

ELEMENTS	ADDRESS	
A[1,1]	200	Column 1
A[2,1]	201	
A[3,1]	202	
A[1,2]	203	Column 2
A[2,2]	204	
A[3,2]	205	
A[1,3]	206	Column 3
A[2,3]	207	
A[3,3]	208	
A[1,4]	209	Column 4
A[2,4]	210	
A[3,4]	211	

For 3x4 array Row Major Order (RMO)

	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Concept:
Row Major Order (RMO)

ELEMENTS	ADDRESS	
A[1,1]	200	Row 1
A[1,2]	201	
A[1,3]	202	
A[1,4]	203	
A[2,1]	204	Row 2
A[2,2]	205	
A[2,3]	206	
A[2,4]	207	
A[3,1]	208	Row 3
A[3,2]	209	
A[3,3]	210	
A[3,4]	211	

To find the address of LA[K] in time independent of *K*. Here, w is the number of words per memory cell for the array LA, and 1 is the lower bound of the index set of LA.

A similar situation also holds for any two-dimensional *m* x *n* array *A*. That is, the computer keeps track of base *[A]*, the address of the first element A[1, 1] of *A* - and computes the address LOC *(A[J, KJ)* of *A[J, K]* using the formula,

(Column major order)

$$LOC(A[J,K]) = Base(A) + w(A(K-1) + (J-1))]$$

(Row major order)

$$LOC\ (A[J, K]) = Base\ (A) + w\ [N(J - 1) + (K - 1)]$$

Again, w denotes the number of words per memorY location for the array *A* . Note that the formula is linear in J and K and that one can find the address LOC *(A[J, K])* in time independent of *J* and *K*.

Unit: II

Stacks- Array Representation of Stacks – Operations on Stack – Arithmetic Expressions: Polish Notation– Reverse Polish Notation – Evaluation of a postfix expression – Transforming Infix Expression into Postfix – Recursion – Queues – Representation of Queues – Operations on Queues – Deques.

2.1 STACK DEFINITION

A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only. The insertion and 'deletion operations in the case of a stack are specially termed PUSH and POP, respectively, and the position of the stack where these operations are performed is known as the TOP of the stack. An element in a stack is termed an ITEM. The maximum number of elements that a stack can accommodate is termed SIZE. Figure shows a typical view of a stack data structure.

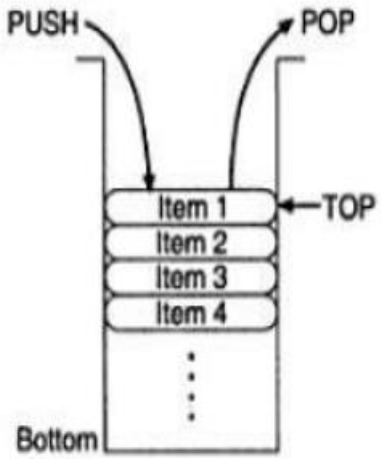


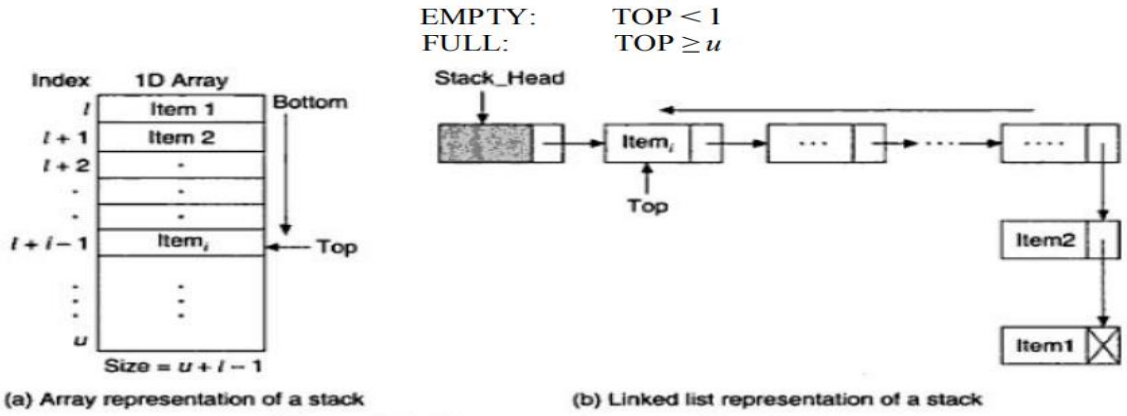
Figure 4.2 Schematic diagram of a stack.

2.2 ARRAY REPRESENTATION OF STACKS : A stack may be represented in the memory in various ways. There are two main ways: using a one-dimens iona l array and a single linked list.

Array Represe ntation of Stacks : First we have to allocate a memory block of suff icient s ize to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the items of the stack can be stored in a sequentia l fashion.

In Figure , Item i denotes the ith item in the stack; l and u denote the index range of the array in use; usually the values of these indices are 1 and SIZE respectively. TOP is a pointer to point the pos ition of the array up to which it is filled w ith the items of the stack. With this

Representation, the following two ways can be stated:



(a) Array representation of a stack (b) Linked list representation of a stack

Linke d List Represe ntation of Stacks : Although array representation of stacks is very easy and convenient but it a llows the representation of only fixed sized stacks. In several applications, the size of the stack may vary dur ing program execution. An obvious solution to this problem is to represent a stack using a linked list.

A single linked list structure is sufficient to represent any stack. Here, the DATA field is for the ITEM, and the LINK field is, as usua l, to point to the next' item. Above Figure b depicts such a stack using a single linked list.

In the linked list representation, the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom-most item. Thus, a PUSH operation w ill add a new node in the front and a POP operation will remove a node from the front of the list.

2.3 OPERATIONS ON STACK

The basic operations required to manipulate a stack are:

PUSH:To insert an item into a stack,

POP:To remove an item from a stack, STATUS: To know the present state of a stack

Algorithm Push_Array
Input: The new item ITEM to be pushed onto it.
Output: A stack with a newly pushed ITEM at the TOP position.
Data structure: An array A with TOP as the pointer.

Steps:

```
1. If TOP ≥ SIZE then
2.     Print "Stack is full"
3. Else
4.     TOP = TOP + 1
5.     A[TOP] = ITEM
6. EndIf
7. Stop
```

Here, we have assumed that the array index varies from 1 to SIZE and TOP points the location of the current top-most item in the stack. The following algorithm Pop_ Array defines the POP of an item from a stack which is represented using an array A.

Algorithm Pop_Array
Input: A stack with elements.
Output: Removes an ITEM from the top of the stack if it is not empty.
Data structure: An array *A* with TOP as the pointer.

Steps:

```
1. If TOP < 1 then
2.     Print "Stack is empty"
3. Else
4.     ITEM = A[TOP]
5.     TOP = TOP – 1
6. EndIf
7. Stop
```

Now let us see how the same operations can be defined for a stack represented with a single linked list.

Algorithm Push_LL
Input: ITEM is the item to be inserted.
Output: A single linked list with a newly inserted node with data content ITEM.
Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

```
1. new = GetNode(NODE)
   /* Insert at front */
2. new→DATA = ITEM
3. new→LINK = TOP
4. TOP = new
5. STACK_HEAD→LINK = TOP
6. Stop
```

Algorithm Pop_LL
Input: A stack with elements.
Output: The removed item is stored in ITEM.
Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

```
1. If TOP = NULL
2.     Print "Stack is empty"
3.     Exit
4. Else
5.     ptr = TOP→LINK
6.     ITEM = TOP→DATA
7.     STACK_HEAD→LINK = ptr
8.     TOP = ptr
9. EndIf
10. Stop
```

Arithmetic Expressions: An arithmetic expression consists of operands and operators. Operands are variables or constants and operators are of various types such as arithmetic unary and binary operators and Boolean operators. In addition to these, parentheses such as '(' and ')' are also used.

Thus, with the below rules of precedence and associativity of operators, the evaluation will take place for the above-mentioned expression in the sequence (sequence is according to the number 1, 2, 3, ... , etc.)

Table 4.1 Precedence and associativity of operators

Operators	*Precedence*	*Associativity*
– (unary), +(unary), NOT	6	–
^ (exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), – (subtraction)	4	Left to right
<, <=, +, < >, >=	3	Left to right
AND	2	Left to right
OR, XOR	1	Left to right

Example: Suppose we want to evaluate the following parenthesis free arithmetic expression 2 ↑3 + 5 * 2 ↑ 2 -12 /6
First we evaluate the exponentiation to obtain 8 + 5 * 4 -12/6
Then we evaluate the multiplication and division to obtain 8 + 20 -2. Last we valuate the addition and subtraction to obtain the final result, 26. Observe that the expression is traversed three times, each time corresponding to a level of precedence of the operations.

2.4 POLISH NOTATION

For most common arithmetic operations, the operator symbol is placed between its two operands. For example, A+B C-D E*F G/H. This is called infix notation.
Polish notation, named after the Polish mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before its two operands.

For example, +AB, -CD, *EF, /GH

We translate, step by step, the following infix expressions into polish notation using brackets to indicate a partial translation:

(A+B)*C = [+AB]*C=*+ABC
A+(B*C) = A+[*BC]=+A*BC
(A+B)/(C-D) = [+AB]/[-CD]=/+AB-CD

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

2.5 REVERSE POLISH NOTATION refers to the analogous notation in which the operator symbol is placed after its two operands: AB+ , CD-, EF*,GH/.

Again, one never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation.

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 -

The computer usually evaluates an arithmetic expression written in infix notation in two steps.
First, it converts the expression to postfix notation,
and then it evaluates the postfix expression.

In each step, the stack is the main tool that is used to accomplish the given task. We illustrate these applications of stack in reverse order. That is, first we show how stacks are used to evaluate postfix expressions and then we show how stacks are used to transform infix expressions into postfix expressions.

2.6 EVALUATION OF A POSTFIX EXPRESSION

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

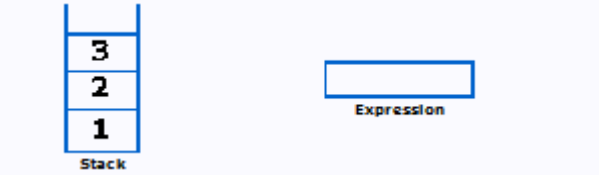
Algorithm:

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

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel]
2. ScanPfromlefttorightandrepeatsteps3and4foreachelementofP until the sentinel")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator 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 operator A
      Place the result of step

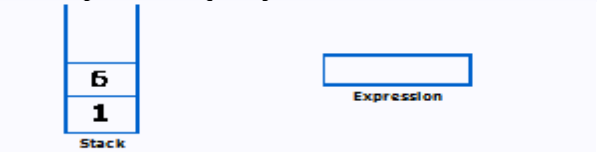
(b) back on STACK [End
of If structure]
[End of Step 2loop]
5.Set VALUE equal to the top
element on STACK. 6.Exit

Let us see how the above algorithm will be
implemented using an example. Postfix String :
123*+4-
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are
operands. Thus they will be pushed into the stack in that order.

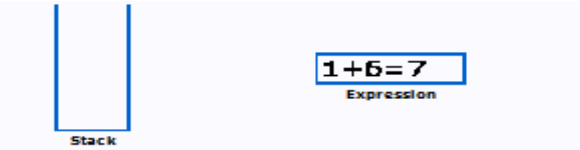


Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the"*"operation with the two operands. The second operand will be the first element that is popped.

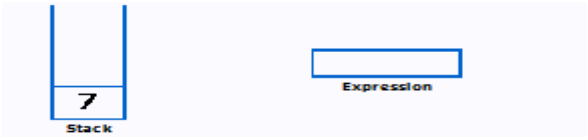
The value of the expression(2*3) that has been evaluated(6) is pushed into the stack.



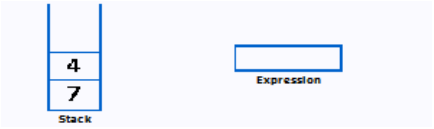
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



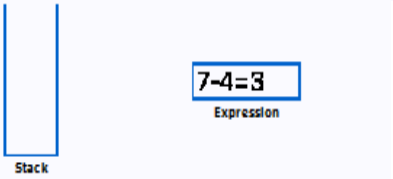
The value of the expression (1+6) that has been evaluated(7) is pushed into the stack.



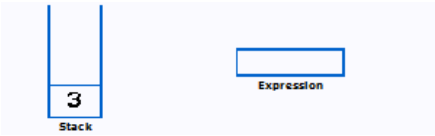
Next character scanned is "4", which is added to the stack.



Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression (7-4) that has been evaluated(3) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.
End result : Postfix String : 123*+4-
Result : 3

To illustrate the algorithm *EvaluatePostfix*, let us consider the following expression:
Infix: A + (B * C) / D
Postfix: A B C * D / +
Input: A B C * D / + # with A = 2, B = 3, C = 4, and D = 6

Read symbol	*Stack*	
A	2	PUSH(A = 2)
B	2 3	PUSH(B = 3)
C	2 3 4	PUSH(C = 4)
*	2 12	POP(4), POP(3), PUSH(T = 12)
D	2 12 6	PUSH(D = 6)
/	2 2	POP(6), POP(12), PUSH(T = 2)
+	4	POP(2), POP(2), PUSH(T = 4)
#		value = POP( )

2. 7 TRANSFORMING INFIX EXPRESSIONS INTO POSTFIX EXPRESSIONS

Let Q be an arithmetic expression written in infix notation. Besides operands and operators, Q may also contain left and right parentheses. We assume that the operators in Q consist only of exponentiations, multiplications, divisions, additions and subtractions and that they have the usual three levels of precedence as given above. We also assume that operators on the same level, including exponentiations, are performed from left to right unless otherwise indicated by parentheses.
The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression P will be constructed from left to right using the operands from Q and the operators which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is completed when STACK is empty.

Algorithm : POLISH(Q,P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK and add ")" to the end of Q.
2. Scan Q from left to right and repeat
   steps 3 to 6 for each element of Q
   until the STACK is empty.
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator is encountered, then
   a. Repeatedly pop from STACK and add to P each operator which has the same precedence as or higher precedence than the operator.
   b. Add operator to STACK. [End of if structure]
6. If a right parenthesis is encountered, then
   a. Repeatedly pop from STACK and add to P each operator until a left parenthesis is encountered.
   b. Remove the left parenthesis. [End of if structure]

[End of step 2 loop]
7. Exit

EXAMPLE: Let us illustrate the procedure *InfixToPostfix* with the following arithmetic expression:

Input: (A + B)^ C - (D * E) / F) (infix form)

Read symbol	*Stack*	*Output*
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+
6	(^	AB+
7	(^	AB + C
8	(–	AB + C ^
9	(– (	AB + C ^
10	(– (	AB + C ^ D
11	(– (*	AB + C ^ D
12	(– (*	AB + C ^ DE
13	(–	AB + C ^ DE *
14	(– /	AB + C ^ DE *
15	(– /	AB + C ^ DE * F
16		AB + C ^ DE * F / –

Output: A B + C ^ DE * F / – (postfix form)

2.8 RECURSION:

Recursion is an important concept in computer science. 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 recursive procedure. A recursive procedure must have the following two properties:

- Base Case: It is nothing more than the simplest instance of a problem, consisting of a condition that terminates the recursive function. This base case evaluates the result when a given condition is met.
- Recursive Step: It computes the result by making recursive calls to the same function, but with the inputs decreased in size or complexity.

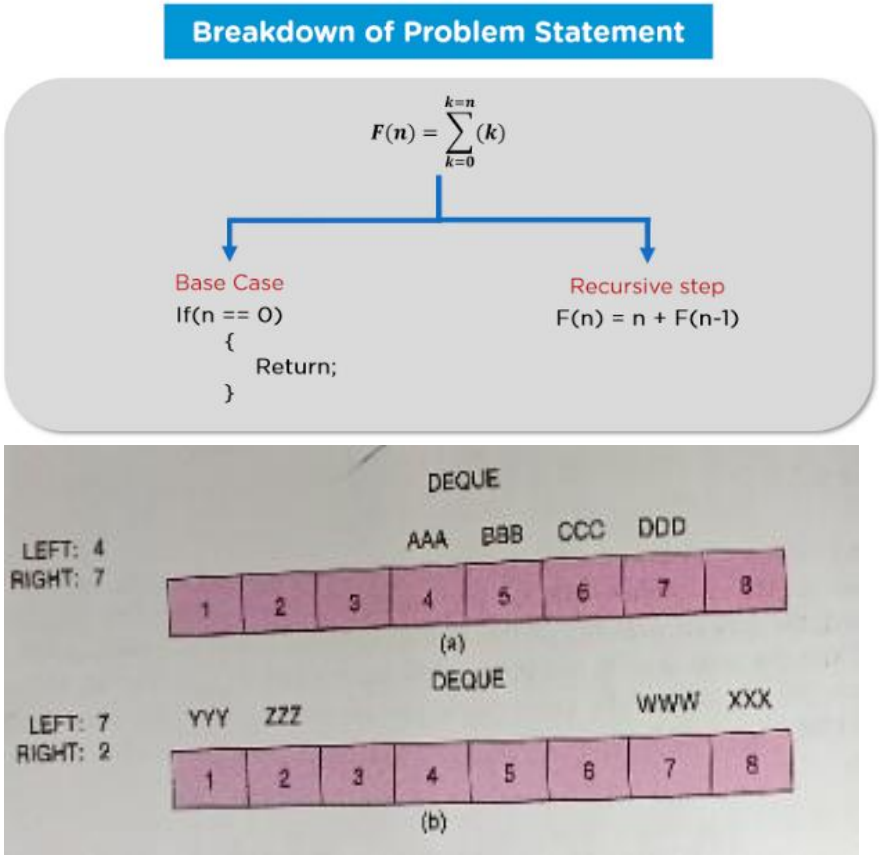
For example, consider this problem statement: Print sum of n natural numbers using recursion. This statement clarifies that we need to formulate a function that will calculate the summation of all natural numbers in the range 1 to n. Hence, mathematically you can represent the function as:

F(n) = 1 + 2 + 3 + 4 + ...+ (n-2) + (n-1) + n

It can further be simplified as:

$$F(n) = \sum_{k=1}^{k=n} (k)$$

You can breakdown this function into two parts as follows:



The following are two procedures that each calculate *n* factorial.

Procedure 6.9A: FACTORIAL(FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Set FACT := 1. [Initializes FACT for loop.]
3. Repeat for K = 1 to N.
   Set FACT := K*FACT.
   [End of loop.]
4. Return.

2.9 QUEUE DEFINITION Like a stack, a queue is an ordered collection of homogeneous data elements; in contrast with the stack, here, insertion and deletion operations take place at two extreme ends. A queue is also a linear data structure like an array, a stack and a linked list where the ordering of elements is in a-linear fashion.

The only difference between a stack and a queue is that in the case of stack insertion and deletion (PUSH and POP) operations are at one end (TOP) only, but in a queue insertion (called ENQUEUE) and deletion (called DEQUEUE) operations take place at two ends called the REAR and FRONT of the queue, respectively. Figure represents a model of a queue structure. Queue is also termed first-in first-out (FIFO)

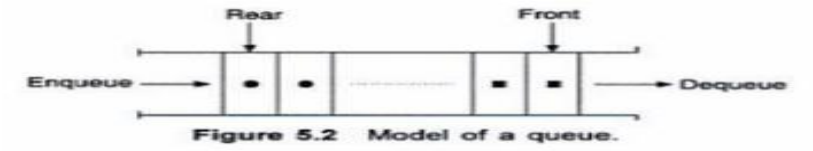


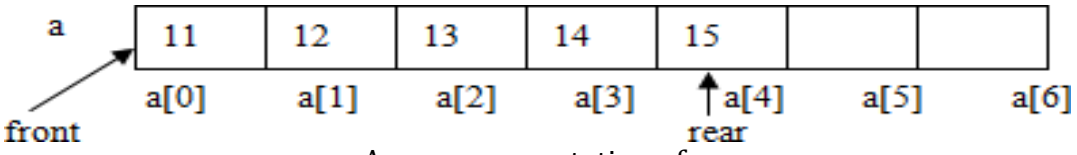
Figure 5.2 Model of a queue.

2.10 REPRESENTATION OF QUEUES :There are two ways to represent a queue in memory: Using an array & Using a linked list The first kind of representation uses a one-dimensional array and it is a better choice where a queue of fixed size is required. The other representation uses a double linked list and provides a queue whose size can vary during processing.

Representation of a Queue using an Array A one-dimensional array, say Q[l ... N], can be used to represent a queue. Figure shows an instance of such a queue. With this representation, two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue. For the insertion of the next element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.

Three states of a queue with this representation are given below:
Queue is empty
FRONT = 0
REAR = 0 (and/or)
Queue is full
REAR = N
FRONT = 1 (when full by compact)
Queue contains elements ≥ 1
FRONT ≤ REAR
Number of elements = REAR − FRONT + 1

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



Array representation of a queue

2.11 OPERATION ON QUEUE

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

This procedure inserts an element ITEM into a queue.

1. If FRONT=1 and REAR=N, or if FRONT=REAR+1, then Write OVERFLOW and return.
2. If
FRONT:=NU
LL then Set
FRONT:=1
and
REAR:=1
Else if
REAR=N
then
Set REAR :=1
Else
Set REAR :=REAR+1
3. Set QUEUE[REAR]:=ITEM
4. Return

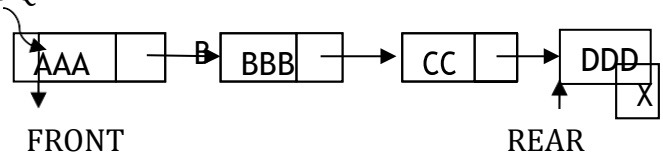
PROCEDURE: QDELETE(QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes and element from a queue and assigns it to the variable ITEM.

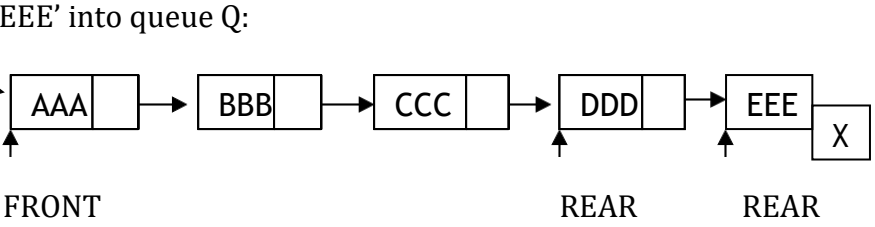
1. If FRONT:=NULL then write UNDERFLOW and return
2. Set ITEM:=QUEUE[FRONT]
3. If FRONT=REAR then
Set FRONT:=NULL and REAR:=NULL
Else if
FRON
T=N
then
Set
FRON
T:=1
Else
Set FRONT:=FRONT+1
4. Return

Linked Representation of Queue:

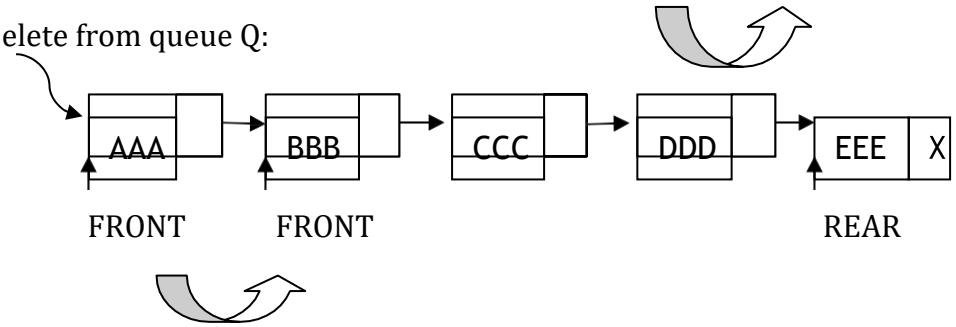
Queue Q:



Insert 'EEE' into queue Q:



Delete from queue Q:



LINK Q-INSERT (INFO, LINK, FRONT, REAR, AVAIL, ITEM)

1. [Available space?] IF AVAIL= NULL, then write OVERFLOW and EXIT.
2. [Remove first node from AVAIL first]
   Set NEW= AVAIL and AVAIL= LINK [AVAIL]
3. Set INFO[NEW]= ITEM and LINK
   [NEW]= NULL (Copies ITEM into
   new node)
4. IF(FROM= NULL) then
   FRONT=REAR=NEW (If Q is empty then
   ITEM is the first element in Q)
   Else set LINK [REAR] =NEW and REAR=NEW
   (REAR points to the new node appended to the end of the list)
5. EXIT.

LINK Q-DELETE (INFO, LINK, FRONT, REAR, AVAIL, ITEM)

This Procedure deletes the front element of the linked queue and stores it in item.

1. [Linked queue empty?]
   If (FRONT = NULL) then the Write: UNDERFLOW and EXIT.
2. Set TEMP = FRONT(If linked queue is nonempty, remember
   FRONT in a temporary variable TEMP)
3. ITEM =INFO(TEMP)
4. FRONT = LINK(TEMP)(Reset front to point to the next element in queue)
5. LINK(TEMP) = AVAIL and AVAIL = TEMP(return deleted node
   TEMP to AVAIL list)
6. EXIT.

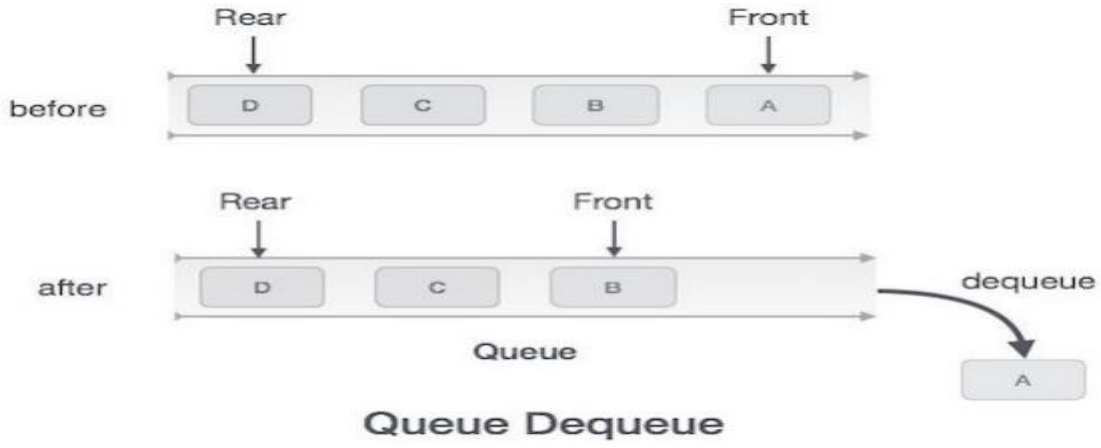
2.12 DEQUES

A deque is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name double ended queue. There are various ways of representing a deque in a computer.

Unless, it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array. The condition LEFT=NULL will be used to indicate that a deque is empty.

There are two variations of a deque- namely an input restricted deque and an output restricted deque—which are intermediate between a deque and a queue.

Specifically, an input restricted deque is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list; and an output restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.



Queue Dequeue

Algorithm for dequeue operation

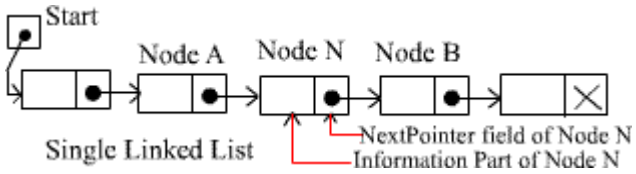
```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Unit: III

Linked List – Representation of Linked Lists in Memory – Traversing a Linked List – Insertion into a Linked List – Deletion from a Linked List – Two-way Linked Lists – Operations on Two-way Lists.

3.1 A linked list

A linked list or one way list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. That is, each node is divided into two parts: the first part contains the information of the element, and the second part, called the link field or next pointer field, contains the address of the next node in the list.



The null pointer, denoted by X in the diagram, signals the end of the list. The linked list also contains a list pointer variable—called START or NAME which contains the address of the first node in the list; hence there is an arrow drawn from START to the first node. Clearly, we need only this address in START to trace through the list. A special case is the list that has no nodes. Such a list is called the null list or empty list and is denoted by the null pointer in the variable START.

Example:

A hospital ward contains 12 beds, of which 9 are occupied. Suppose we want an alphabetical listing of the patients. This listing may be given by the pointer field, called Next. we use the variable START to point to the first patient. Hence START contains 5, since the first patient. Adams, occupies bed 5. Also, Adams's pointer is equal to 3, since Dean, the next patient, occupies bed 3; Dean's pointer is 11, since Fields, the next patient, occupies bed 11; and so on. The entry for the last patient (Samuels) contains the nullpointer,denotedby0.(Some8arrows have been drawn to indicate the listing of the first few patients}

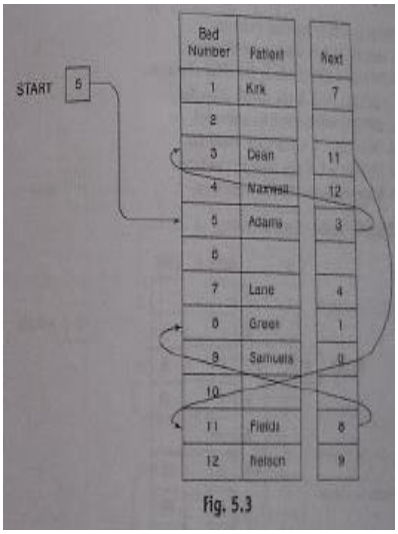
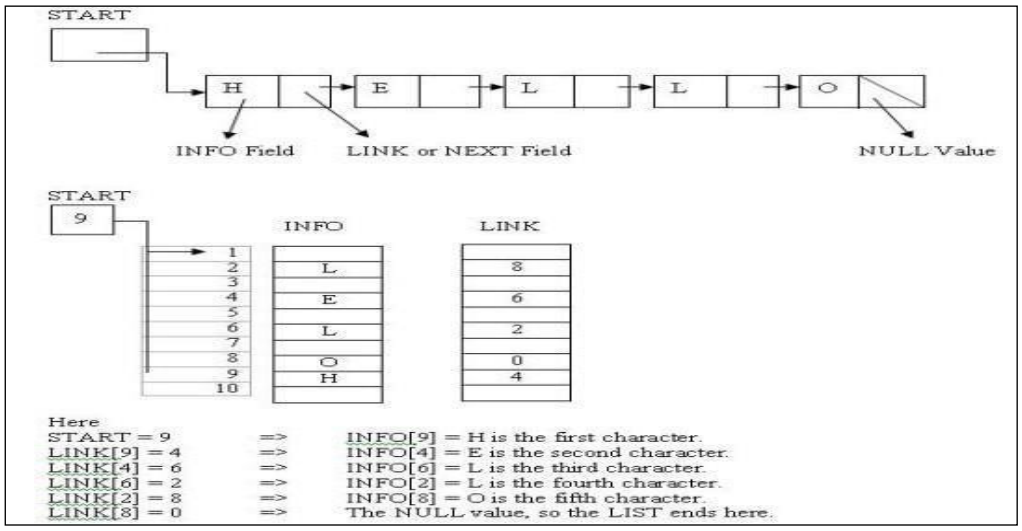


Fig. 5.3

3.2 Representation of Linked Lists in Memory

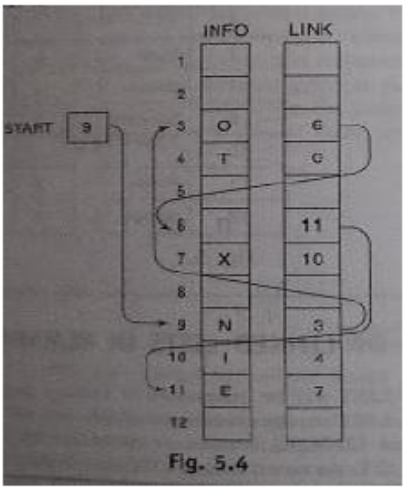
Let LIST be a linked list. Then LIST will be maintained in memory, unless otherwise specified or implied, as follows. First of all, LIST requires two linear arrays-we will call them here INFO and LINK – such that INFO[K] and LINK[K] contain, respectively, the information part and the next pointer filed of anode of LIST. As noted above, LIST also requires a variable name — such as START --- which contains the location of the beginning of the list, and a next pointer sentinel—denoted by NULL—which indicates the end of the list. Since the subscripts of the arrays INFO and LINK will usually be positive, we will choose NULL=0, unless otherwise stated.



➢ Example1:

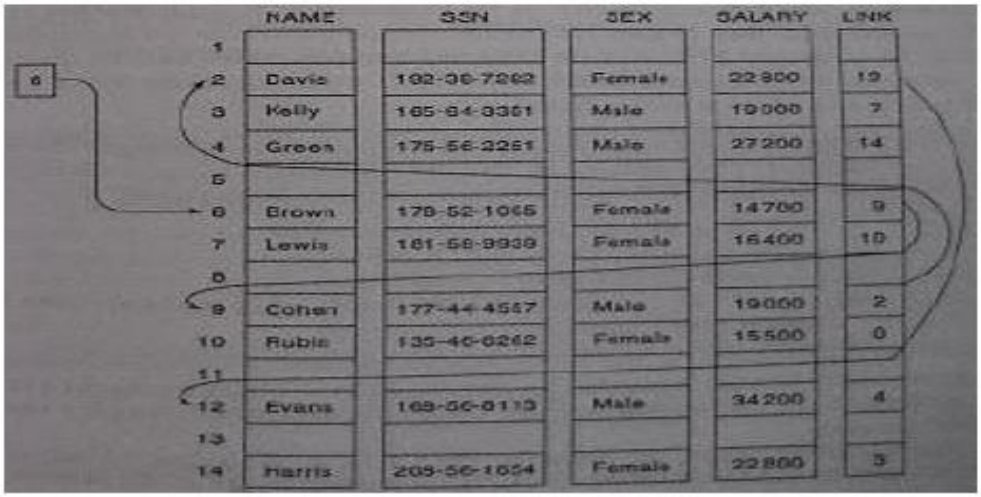
Pictures a linked list in a memory where each nodes of the list contains a single character.

START = 9, so INFO [9] = N is the first character.
LINK [9] = 3, so INFO [3] = 0 is the second character.
LINK [3] = 6, so INFO [6] =(blank) is 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.



Example2:

Suppose the personnel file of a small company contains the following data on its nine employees:
Name, Social Security Number, Sex, Monthly salary
Normally, four parallel arrays, say NAME, SSN, SEX, SALARY, are required to store the data.



3.3 Traversing a Linked list

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

Traversing algorithm uses 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 the assignment

PTR:=LINK[PTR]

Moves the pointer to the next node in the list.

The details of the algorithm are as follows. Initialize PTR or START. Then process INFO[PTR], the information at the first node. Update PTR by the assignment PTR:=LINK[PTR], so that PTR points to the second node. Then process INFO[PTR], the information at the second node. Again update PTR by the assignment PTR:=LINK[PTR], and then process INFO[PTR], the information at the third node. And so on. Continue until PTR=NULL, which signals the end of the list.

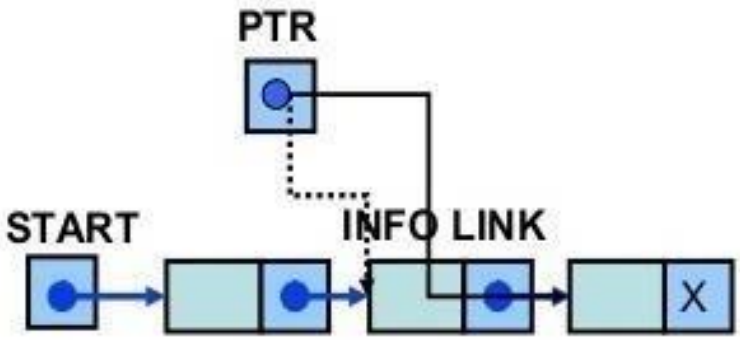


Fig : PTR : = LINK[PTR]

Algorithm: Traversing a Linked List

Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set PTR := START.
2. Repeat Steps 3 and 4 while PTR ≠ NULL
3. Write : INFO[PTR].
4. Set PTR:=LINK[PTR]
5. Return.

The following procedure finds the number NUM of elements in a linked list.

Procedure: COUNT(INFO,LINK,START,NUM)

1. Set NUM := 0.[initializes counter]
2. Set PTR := START.
3. Repeat Steps 4 and 5 while PTR ≠ NULL
4. Set NUM=NUM+1.
5. Set PTR:=LINK[PTR]
6. Return.

This procedure traverses the linked list in order to count the number of elements.

3.4 Insertion into a linked list

Insertion algorithms

Algorithms which insert nodes into linked lists come up in various situations.

- The first one inserts a node at the beginning of the list,
- Second one inserts a node after the node with a given location,
- Third one inserts anode into a sorted list.

All algorithms assume that the linked list is in memory in the form LIST(INFO,LINK,START,AVAIL) and that the variable ITEM contains the new information to be added to the list.

Since our insertion algorithms will use a node in the AVAIL list, all of the algorithms will include the following steps:

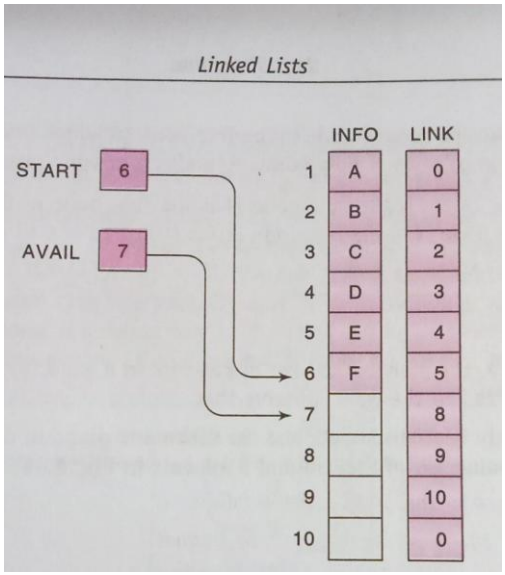
(a) Checking to see if space is available in the AVAIL list. If not, that is, if AVAIL=NULL, then the algorithm will print the message OVERFLOW.

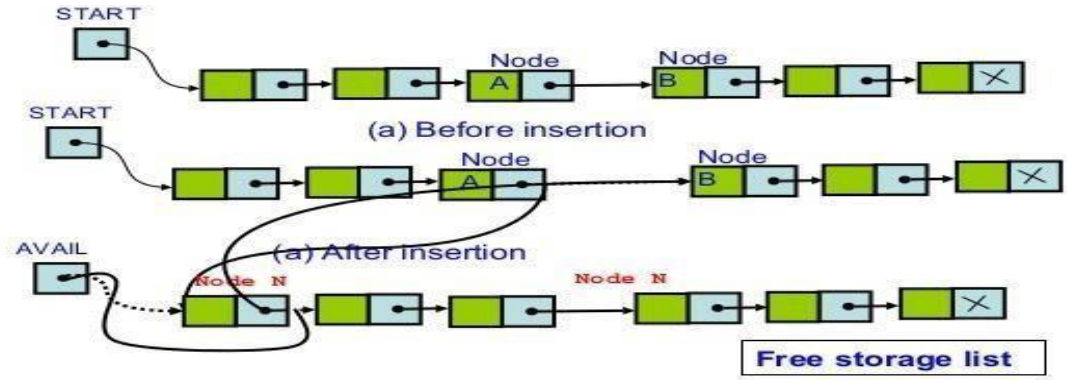
(b) Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node, this step can be implemented by the pair of assignments.

NEW:=AVAIL, AVAIL:=LINK[AVAIL]

(c) Copying new information into the new

node. In other words, INFO[NEW]:=ITEM





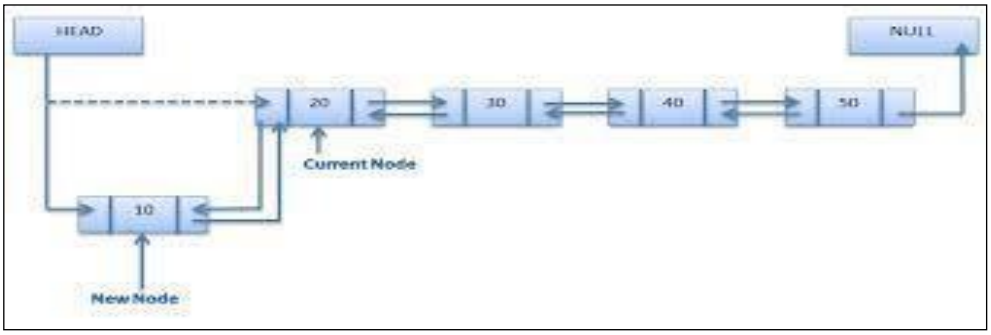
Inserting at the beginning of a list

Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list. Then the easiest place to insert the node is at the beginning of the list. An algorithm that does so follows.

Algorithm: INSFIRST(INFO,LINK,START,AVAIL,ITEM)

This algorithm inserts ITEM as the first node in the list.

1. If AVAIL=NULL, then write OVERFLOW, and exit.
2. Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
3. Set INFO[NEW]:=ITEM(copies new data into new node)
4. Set LINK[NEW]:=START(New node now points to original first node)
5. Set START:=NEW (changes START so it points to the new node)
6. Exit



Inserting after a given node

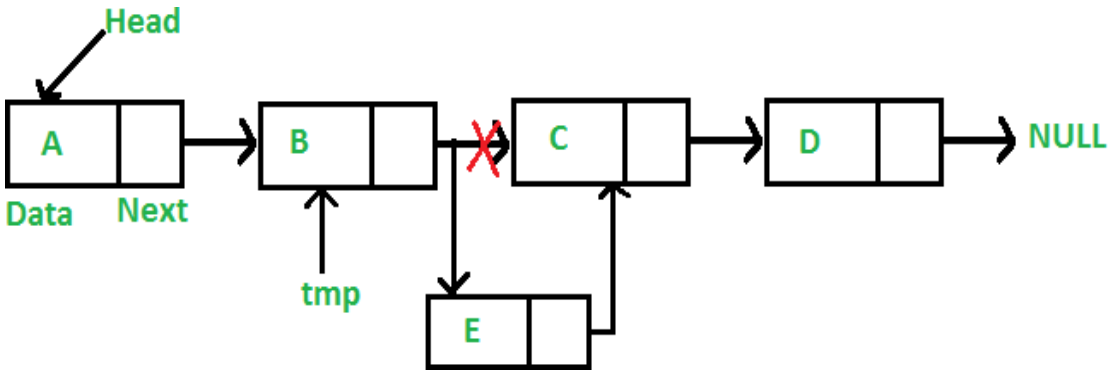
We are given a pointer to a node, and the new node is inserted after the given node. Follow the steps to add a node after a given node:

- Firstly, check if the given previous node is NULL or not.
- Then, allocate a new node and
- Assign the data to the new node
- And then make the next of new node as the next of previous node.
- Finally, move the next of the previous node as a new node.

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

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

1. If AVAIL=NULL, then write OVERFLOW and exit
2. Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
3. Set INFO[NEW]:=ITEM
4. If LOC=NULL, then
   Set LINK [NEW]:=START and START:=NEW
   Else
   Set LINK[NEW]:=LINK[LOC] and LINK[LOC]:=NEW
5. Exit



Inserting into a sorted linked list

Given a sorted list in increasing order and a single node, insert the node into the list's correct sorted position. The function should take an existing node and rearranges pointers to insert it into the list.

Suppose ITEM is to be inserted into assorted linked LIST. Then ITEM must be inserted between nodes A and B so that INFO(A) < ITEM ≤ INFO(B)

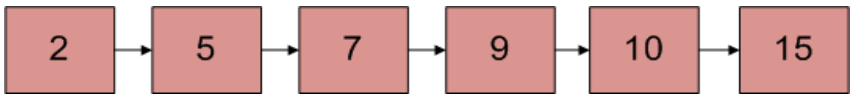
The following is a procedure which finds the location LOC of node A, that is, which finds the location LOC of the last node in LIST whose value is less than ITEM.

Procedure: FINDA(INFO,LINK,START,ITEM,LOC)
This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC]<ITEM, or sets LOC=NULL.
1. If START=NULL, then Set LOC:=NULL and return
2. If ITEM< INFO[START], then set LOC:=NULL and return
3. Set SAVE:=START and PTR:= LINK[START]
4. Repeat steps 5 and 6 while PTR ≠NULL
5. If ITEM< INFO[PTR] then
   Set LOC:=SAVE and return
6. Set SAVE:=PTR and PTR:=LINK[PTR]
7. Set LOC:=SAVE
8. Return

Now we have all the components to present an algorithm which inserts ITEM into a linked list. The simplicity of the algorithm comes from using the previous two procedures.

Algorithm : INSERT(INFO,LINK.START,AVAIL,ITEM)
This algorithm inserts ITEM into a sorted linked list.
1. Call FINDA(INFO,LINK,START,ITEM,LOC)
2. Call INSLOC(INFO,LINK,START,AVAIL,LOC,ITEM)
3. Exit

Linked List after insertion of 9



3.5 Deletion from a linked list

You can delete either from the beginning, end or from a particular position.
1. Delete from beginning
   - Point head to the second node, ie START will point to second node.

2. Delete from end
   - Traverse to second last element
   - Change its next pointer to NULL.

3. Delete from middle
   - Traverse to element before the element to be deleted
   - Change next pointers to exclude the node from the chain

Deleting the node following a given node

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node.

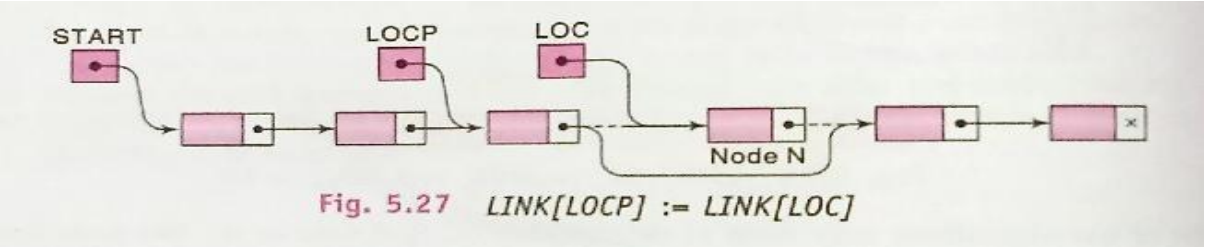


Fig. 5.27 LINK[LOCP] := LINK[LOC]

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

ThisalgorithmdeletesthenodeNwithlocationLOC.LOCPisthelocationofthe nodewhich precedes N or, when N is the first node,LOCP=NULL.

1. If LOCP=NULL,thenSet START:=LINK[START]
   Else
   Set LINK[LOCP]:=LINK[LOC].
2. Set LINK[LOC]:=AVAIL andAVAIL:=LOC
3. Exit.

Deleting the node with a given ITEM of information

Let LIST be a linked list in memory. Suppose we are given an ITEM of information and we want to delete from the LIST, the first node N which contains ITEM. Recall that before we can delete N from the list, we need to know the location of the node preceding N. Accordingly, first we give a procedure which finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N. If N is the first node, we set LOCP=NULL, and if ITEM does not appear in LIST, we setLOC=NULL.

Procedure: FINDB(INFO,START,ITEM,LOC,LOCP)

This procedure finds the location LOC of the first node N which contains ITEM and the locationLOCPofthenodepredecingN.IfITEMdoesnotappearinthelist,thent

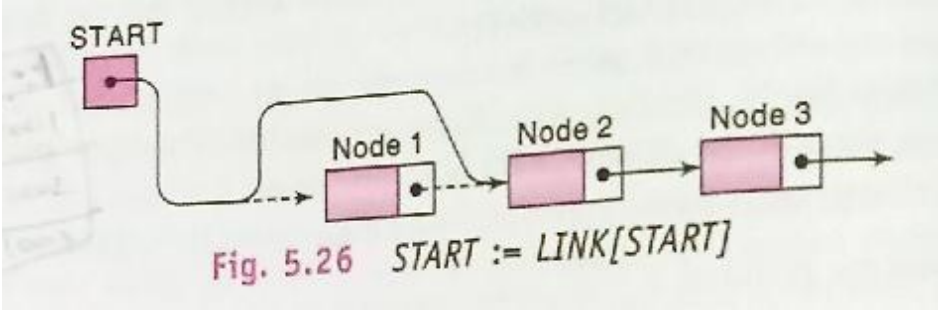


Fig. 5.26 START := LINK[START]

heprocedure sets LOC=NULL, and if ITEM appears in the first node, then it setsLOCP=NULL.

1. If START =NULLthen
Set LOC:=NULL and LOCP:=NULL and return.
2. If INFO[START]=ITEM,then
Set LOC:=START and LOCP=NULL and return.
3. Set SAVE:=START andPTR:=LINK[START]
4. Repeat Steps 5 and 6 while PTR≠NULL
5. If INFO[PTR]=ITEM then
   Set LOC:=PTR and LOCP:=SAVE and return.
6. Set SAVE:=PTR andPTR:=LINK[PTR]
7. SetLOC:=NULL
8. Return

Now we can easily present an algorithm to delete the first node N from a linked list which contains a given ITEM of information.

Algorithm: DELETE(INFO,LINK,START,AVAIL,ITEM)

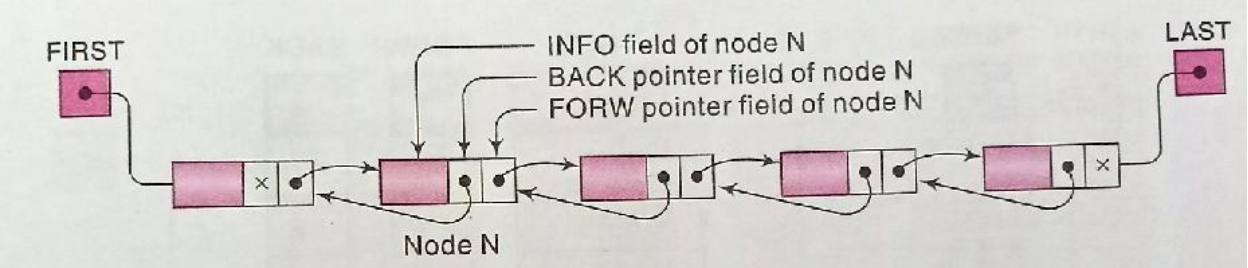
This algorithm deletes from a linked list the first node N which contains the given ITEM of information.

1. Call FINDB(INFO,LINK,START,ITEM,LOC,LOCP)
2. If LOC=NULL, then Write ITEM not in list, and exit.
3. If LOCP=NULL then
   Set START:=LINK[START]
   Else
   Set LINK[LOCP]:=LINK[LOC].
4. Set LINK[LOC]:=AVAIL and AVAIL:=LOC
5. Exit

3.6 Two way lists

A two way list is a linear collection of data elements, called nodes, where each node N is divided into three parts:

- An information field INFO which contains the data of N.
- A pointer field FORW which contains the location of the next node in the list.
- A pointer field BACK which contains the location of the preceding node in the list.



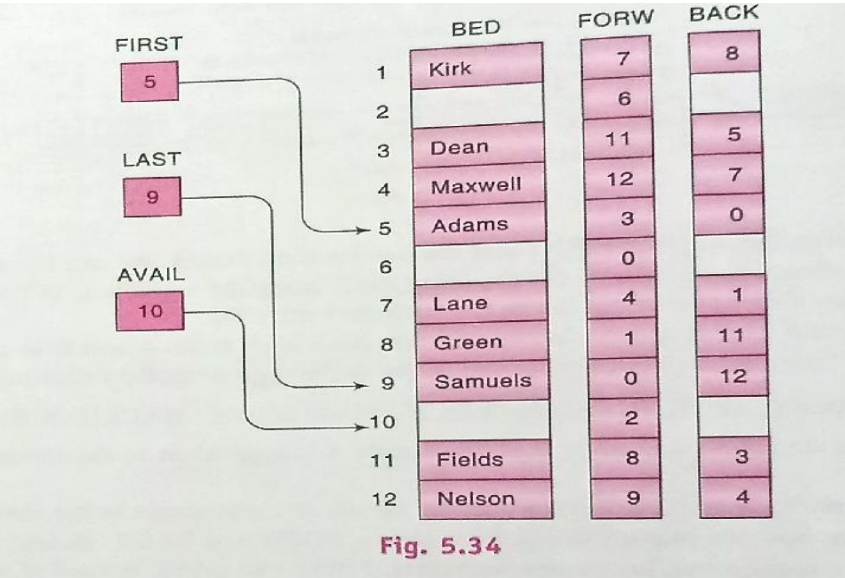


Fig. 5.34

The list also requires two list pointer variables: FIRST, which points to the first node in the list, and LAST, which points to the last node in the list.

Observe that the null pointer appears in the FORW field of the last node in the list and also in the BACK field of the first node in the list.

Observe that, using the variable FIRST and the pointer field FORW, we can traverse a two way list in the forward direction as before. On the other hand, using the variable LAST and the pointer field BACK, we can also traverse the list in the backward direction.

Suppose LOCA and LOCB are the locations, respectively, of nodes A and B in a two way list. Then the way that the pointers FORW and BACK are defined gives us the following:

Pointer property: FORW[LOCA]=LOCB if and only if BACK[LOCB]=LOCA
Two way lists may be maintained in memory by means of linear arrays in the same way as one way lists except that now we require two pointer arrays, FORW and BACK, instead of one pointer array LINK, and we require two list pointer variables, FIRST and LAST, instead of one list pointer variable START.

Two way Header Lists

The advantages of a two way list and a circular header list may be combined into a two way circular header list. The list is circular because the two end nodes point back to the header node. Observe that such a two way list requires only one list pointer variable START, which points to the header node. This is because the two pointers in the header node point to the two ends of the list.

3.7 Operations on Two way lists

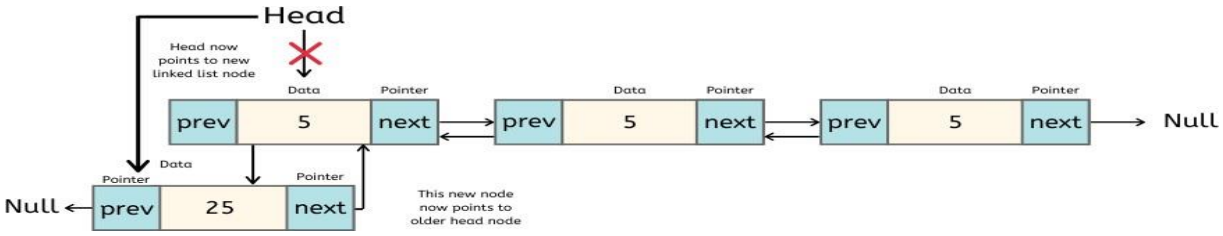
- **Traversing:** Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.
- **Searching**: Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
- **Inserting :**For each insertion operation, we need to consider the three cases. These three cases also need to be considered when removing data from the doubly linked list.

1. Insertion at the beginning
2. Insertion after nth Node
3. Insertion at last

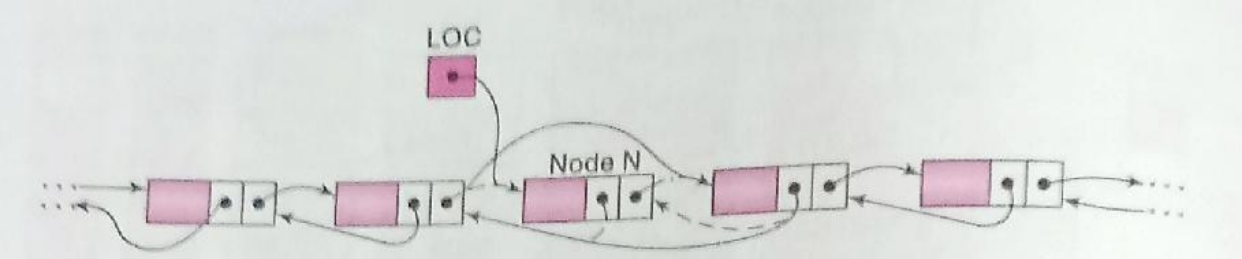
1.Insertion at the Beginning

In the doubly linked list, we would use the following steps to insert a new node at the beginning of the doubly linked list.

- Create a new node
- Assign its data value
- Assign newly created node's next ptr to current head reference. So, it points to the previous start node of the linked list address
- Change the head reference to the new node's address.
- Change the next node's previous pointer to new node's address (head reference)



- **Deleting :** We are given the location LOC of a node N in LISR , and we want to delete N from the list. BACK [LOCK] and FORW [LOC] are the locations. FORW[BACK[LOC]]=FORW[LOC] and BACK[FORW[LOC]]=BACK[LOC].



Algorithm
DELTWL (INFO, FORW, BACK, START, AVAIL, LOC)
1. [Delete node.]
Set FORW[BACK[LOC]]=FORW[LOC] and BACK[FORW[LOC]]=BACK[LOC].
2.[Return node to AVAIL list]
Set FORW[LOC]=AVAIL and AVAIL=LOC

3. Exit

Unit: IV

Trees - Binary Trees – Representing Binary Trees in Memory – Traversing Binary Tree – Threads –Binary Search Tree – Graph Theory – Terminology – Sequential Representation of Graph: Adjacency Matrix, Path Matrix – Traversing a Graph, Breadth First Search, Depth First Search.

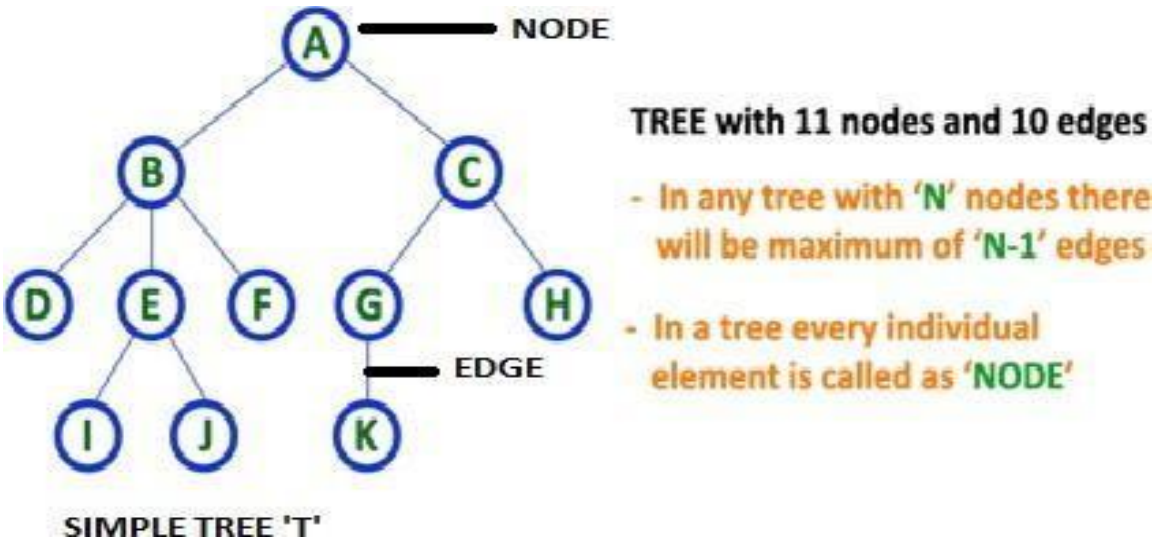
TREES AND BINARY TREES

INTRODUCTION

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

DEFINITION OF TREE:

Tree is collection of nodes (or) vertices and their edges (or) links. In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.



SIMPLE TREE 'T'

Note: 1. In a **Tree**, if we have **N** number of nodes then we can have a maximum of **N-1** number of links or edges.

2. **Tree** has no cycles.

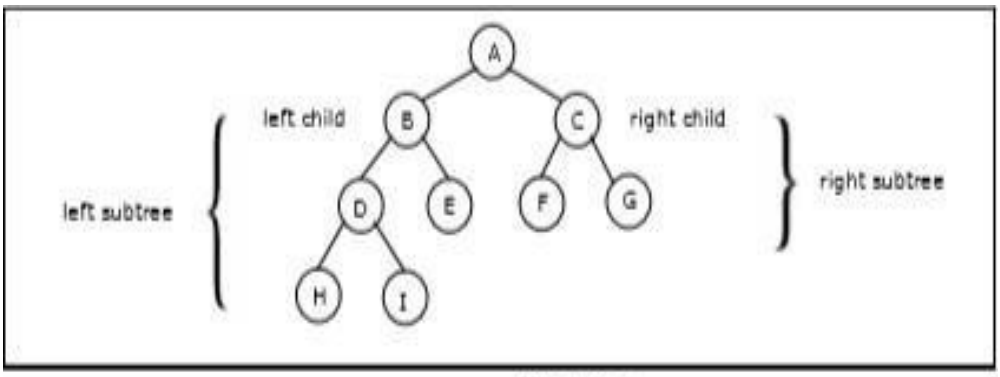
4.1 BINARY TREE:

In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as a left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called Binary Tree. In a binary tree, every node can have either 0 children or 1 child or 2 children butnot more than 2 children.

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either empty or consists of a node called the root together with two binary trees called the left subtree and the right subtree. A tree with no nodes is called as a null tree

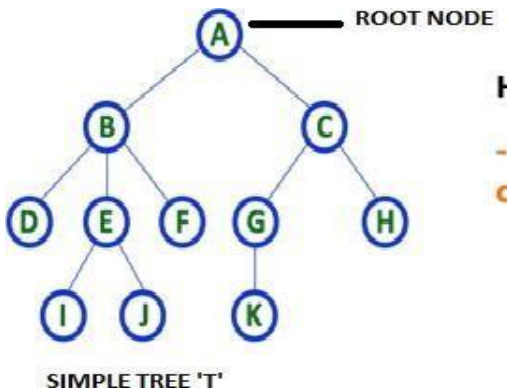
Example:



. Binary Tree

TREE TERMINOLOGIES:

1. Root Node: In a **Tree** data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

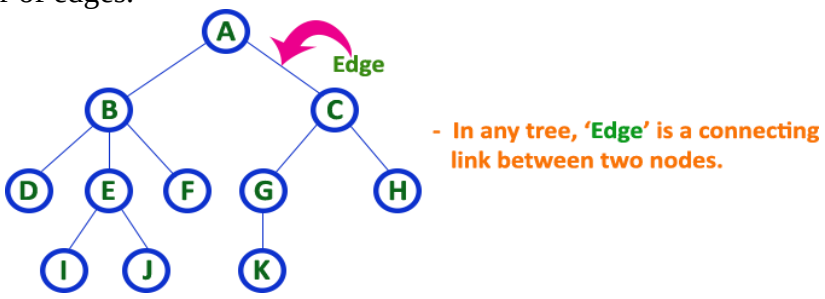


SIMPLE TREE 'T'

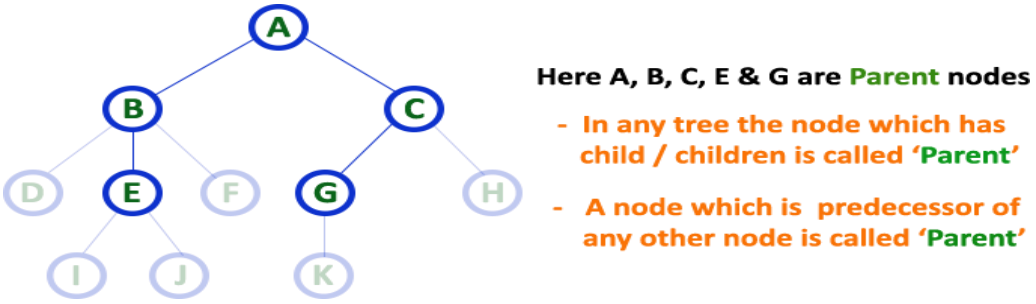
Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. **Edge:** In a **Tree**, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.

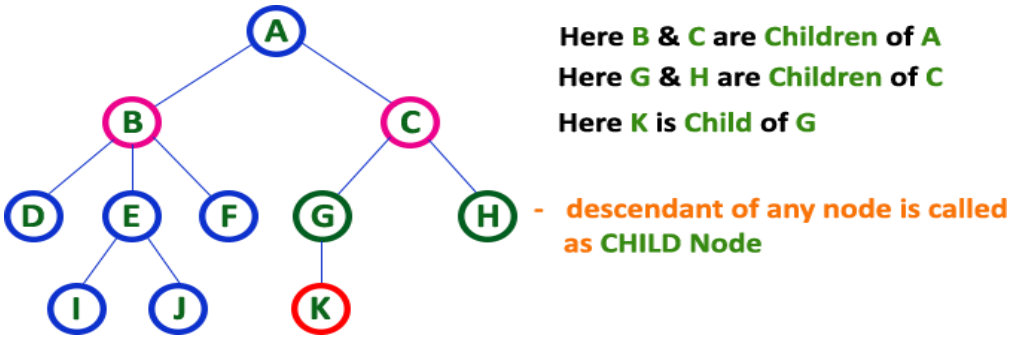


3. **Parent Node:** In a **Tree**, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**

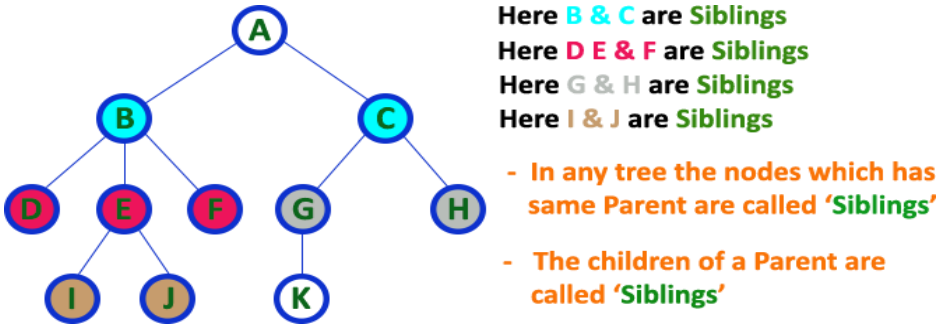


Here, A is parent of B&C. B is the parent of D,E&F and so on…

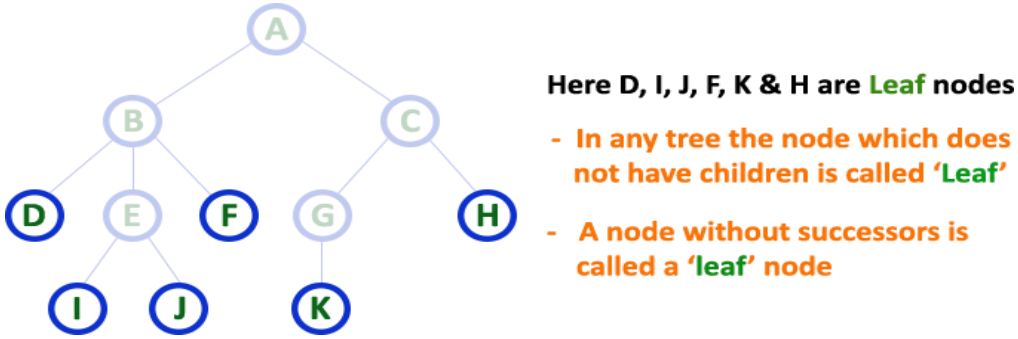
4. **Child Node:** In a **Tree** data structure, the no de which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



5. **Siblings:** In a **Tree** data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.

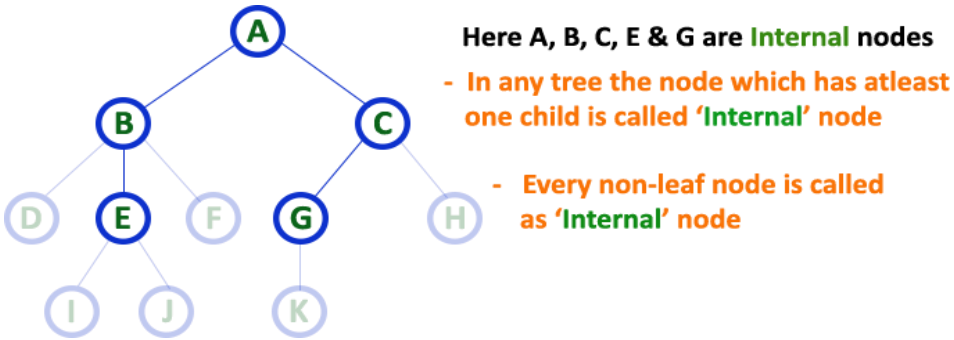


Leaf Node: In a **Tree** data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.

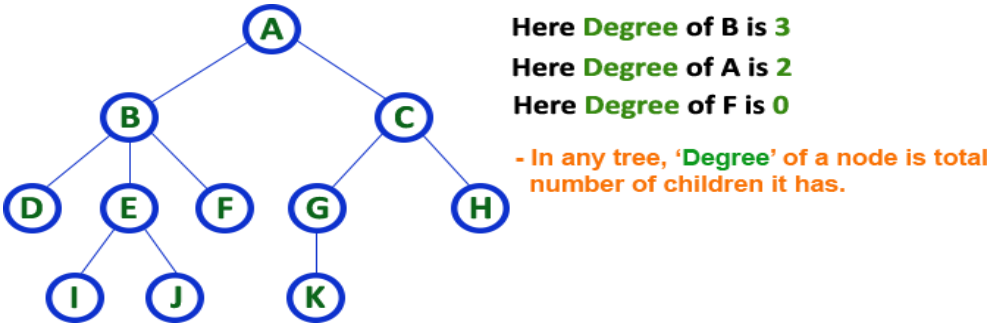


6. **Internal Nodes:** In a **Tree** data structure, the node which has at least one child is called as INTERNAL Node. In simple words, an internal node is a node with at least one child.

   In a **Tree** data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.

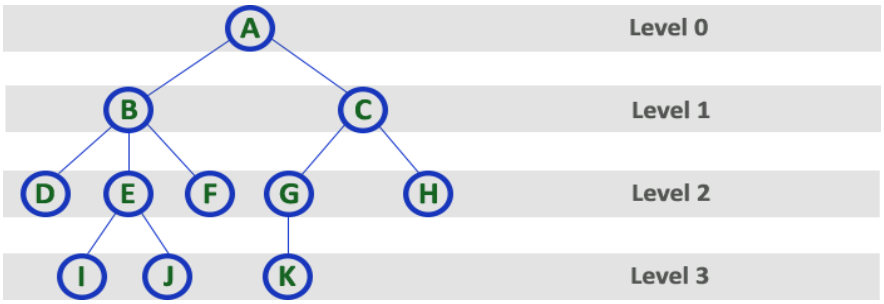


7. Degree: In a **Tree** data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'

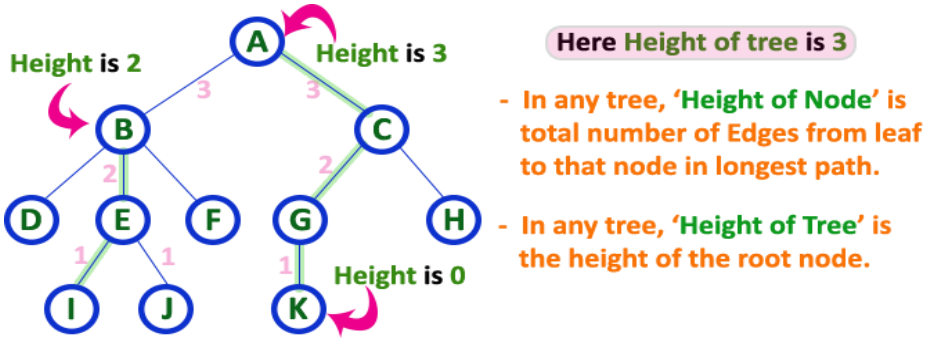


Degree of Tree is: 3

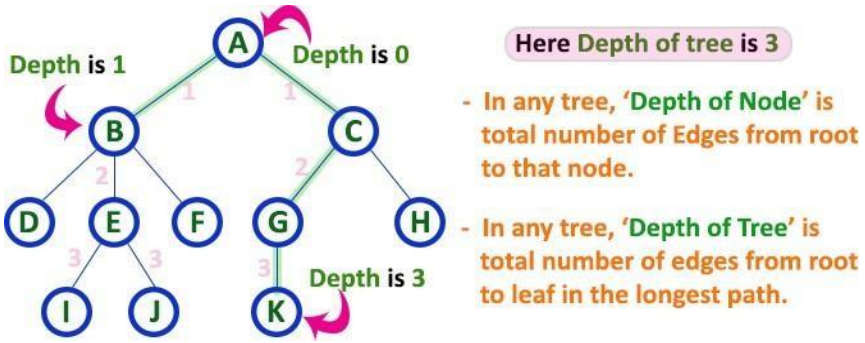
8.Level: In a **Tree** data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as aLevel andthe Level count starts with '0' and incremented by one at each level (Step).



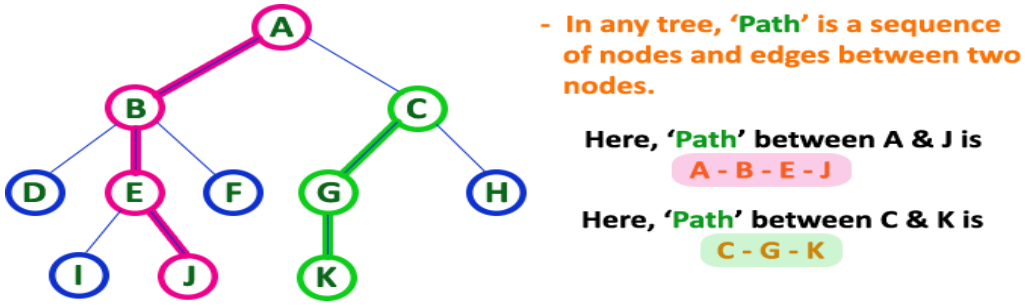
9.Height: In a **Tree** data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



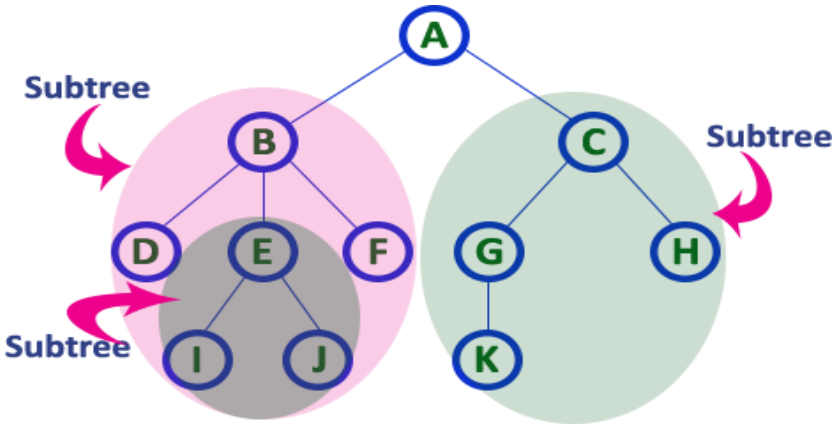
10. Depth: In a **Tree** data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**



11.Path: In a **Tree** data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - Jhas length 4**.



12.Sub Tree: In a **Tree** data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



TYPES OF BINARY TREE:

1. Complete Binary Tree:

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

Complete binary tree is also called as **Perfect Binary Tree.**

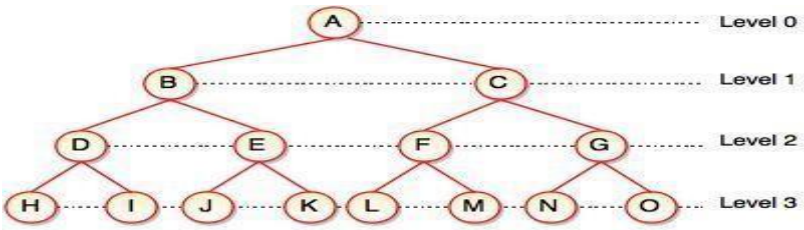
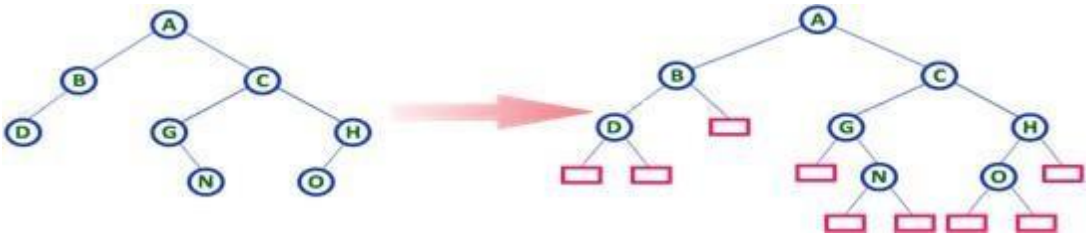


Fig. Complete Binary Tree

2.Extended Binary Tree:

A binary tree can be converted into Full Binary tree by adding dummy nodesto existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes.

Properties of binary trees:

Some of the important properties of a binary tree are as follows:

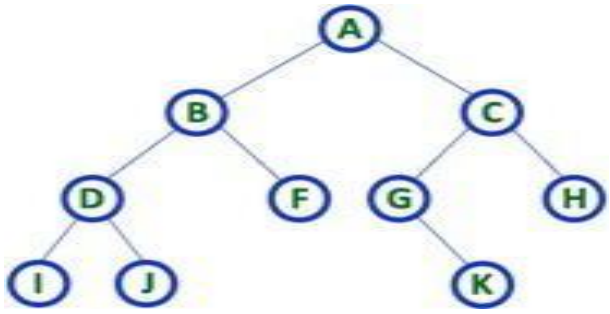
1. If h = height of a binary tree, then
   a. Maximum number of leaves = 2h
   b. Maximum number of nodes = 2h + 1 - 1
2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain atmost 2l node at level l.
4. The total number of edges in a full binary tree with n node is n –1.

4.2 BINARY TREE REPRESENTATIONS:

A binary tree data structure is represented using two methods. Thosemethods are asfollows...

1. **Array Representation**
2. **Linked List Representation**

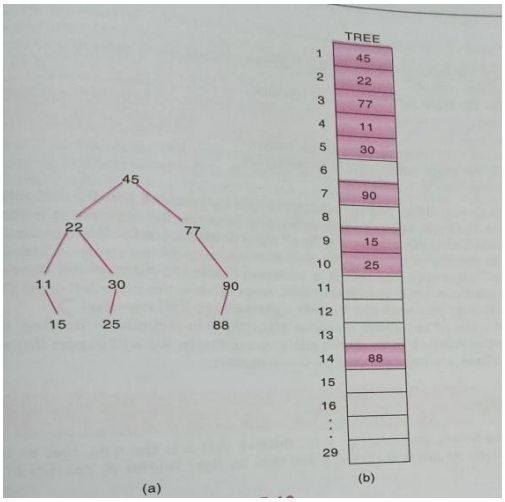
Consider the following binary tree.



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-DArray) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...



To represent a binary tree of depth '**n**' using array representation, we need one dimensional array with a maximum size of 2^{n+1}.

2.Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list,every node consists of **three fields**.

- First field for storing left child address.
- second for storing actual data and third for the right child address.
- In this linked list representation, a node has the following structure

The above example of the binary tree represented using Linked list representationis shown as follows

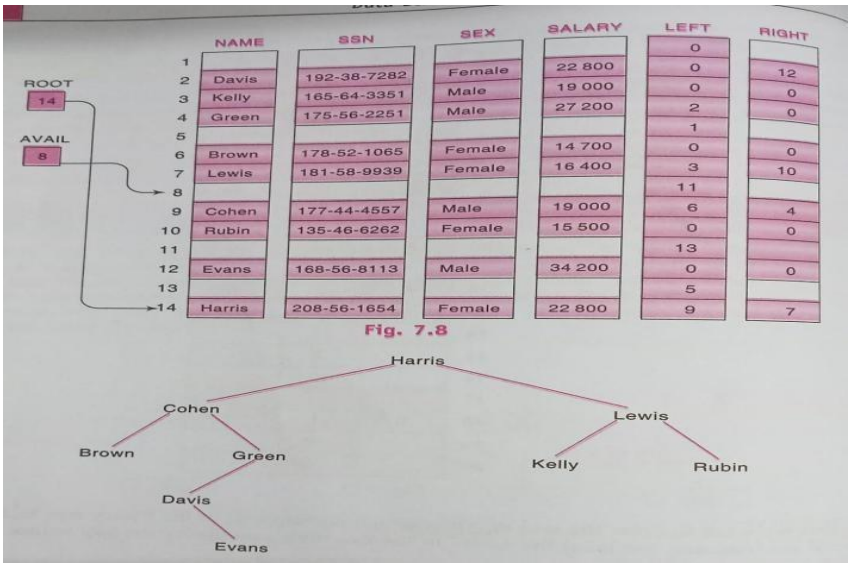


Fig. 7.8

4.3 BINARY TREE TRAVERSALS:

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, binary trees can be traversed in different ways. Following are the generally used ways for traversing binary trees.

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

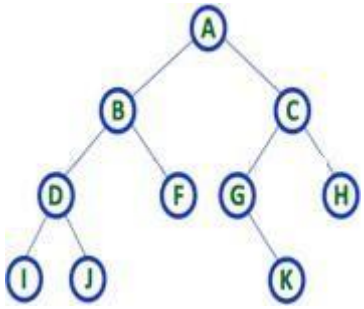
There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

1. In - Order Traversal (left Child - root - right Child):

In In-Order traversal, the root node is visited between the left child and right child.In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all sub trees in the tree. This is performed recursively for all nodes in the tree.

Step-1: Visit the left
subtree, using inorder.
Step-2: Visit the root.
Step-3: Visit the right subtree, using inorder.



In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**. With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A. Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for the right part of the node
A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit **'G'** and then visit G's right child K. With this we have completed the left part of node C. Then visit root node **'C'** and next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H**

using In- Order Traversal.

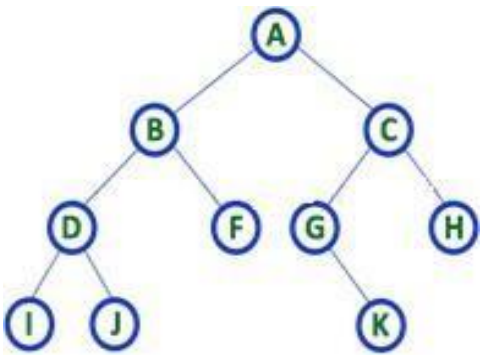
2.Pre - Order Traversal (root - leftChild - rightChild):

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtreesin the tree. Preorder search is also called backtracking

.

Algorithm:

Step-1: Visit the root.
Step-2: Visit the left subtree, usingpreorder.
Step-3: Visit the right subtree, using preorder



In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F. So we visit B's left child **'D'** and again D is a root for I and J. So we visit D's left child **'I'** which is the leftmost child. So next we go for visiting D's right child **'J'**. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**. With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this, we have completed node C's root and left parts. Next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H**

using Pre- Order Traversal.

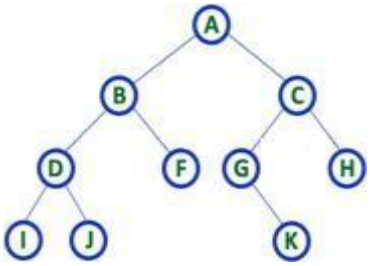
3.Post - Order Traversal (left Child – right Child - root):

In Post-Order traversal, the root node is visited after left child and right child.In this traversal, left child node is visited first, then its right child and then itsroot node. This is recursively performed until the right most nodes are visited.**Algorithm:**

Step-1: Visit the left subtree, using post order.

Step-2: Visit the right subtree, using post order

Step-3: Visitthe root.

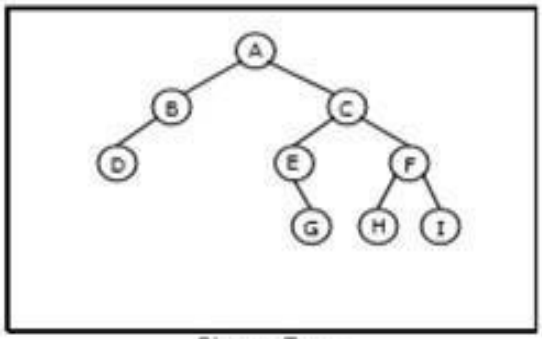


Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A**

using Post-OrderTraversal

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



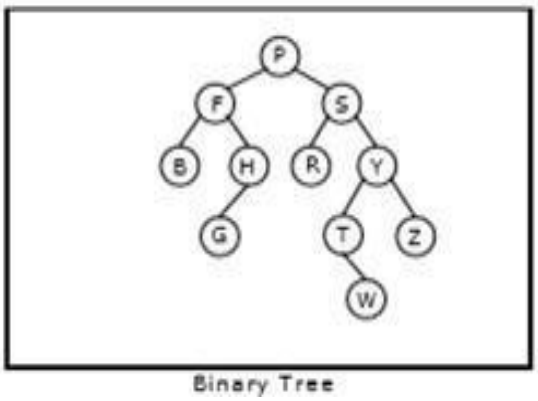
Binary Tree

- Preorder traversal yields: A, B, D, C, E, G, F, H, I
- Postorder traversal yields: D, B, G, E, H, I, F, C, A
- Inorder traversal yields: D, B, A, E, G, C, H, F, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z

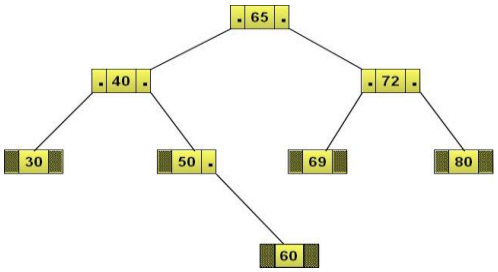
Pre, Post, Inorder and level order Traversing

4.4 HEADER NODES:THREADS

- In a binary search tree, there are many nodes that have an empty left childor empty right child or both.
- You can utilize these fields in such a way so that the empty left child of anode points to its in order predecessor and empty right child of the node points to its inorder successor.
- One way threading:- A thread will appear in a right field of a node andwill point to the next node in the inorder traversal.
- Two way threading:- A thread will also appear in the leftfield of a nodeand will point to the preceding node in the inorder traversal.

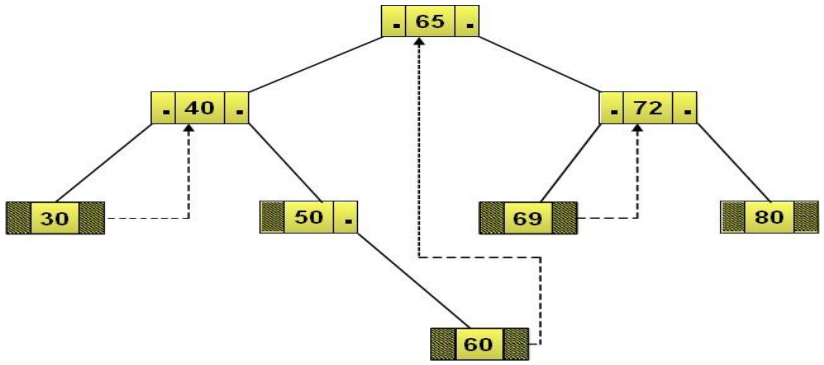
Defining Threaded Binary Trees

- Consider the following binary search tree.
- Most of the nodes in this tree hold a NULL value in their left or right child fields.
- In this case, it would be good if these NULL fields are utilized for some other useful purpose.



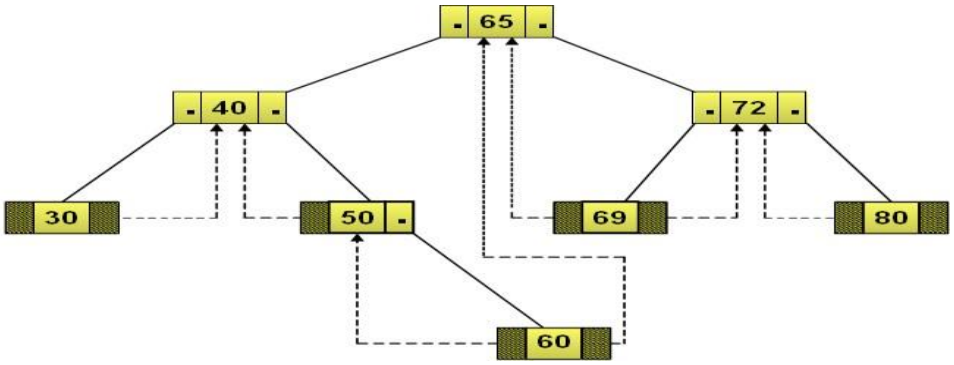
One Way Threaded Binary Trees

- The empty left child field of a node can be used to point to its in order predecessor.
- Similarly, the empty right child field of a node can be used to point to its in-order Such a type of binary tree is known as a one way threaded binary tree.
- A field that holds the address of its in-order successor is known as thread. In-order :- 30 40 50 60 65 69 72 80

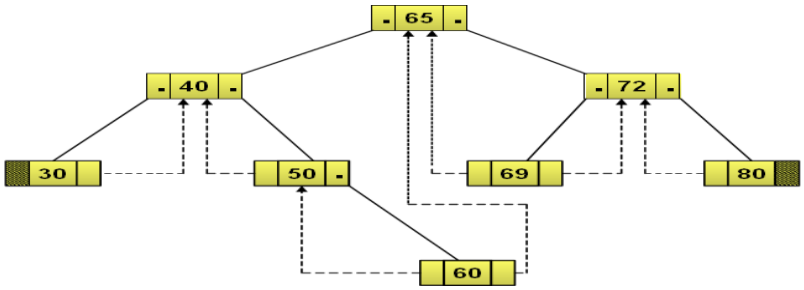


Two way Threaded Binary Trees

- Such a type of binary tree is known as a threaded binary tree.
- A field that holds the address of its inorder successor or predecessor is known as thread. The empty left child field of a node can be used to point to its inorder predecessor.
- Similarly, the empty right child field of a node can be used to point to its inorder successor. Inorder :- 30 40 50 60 65 69 72 80



Node 30 does not have an inorder predecessor because itis the first node to be traversed ininorder sequence

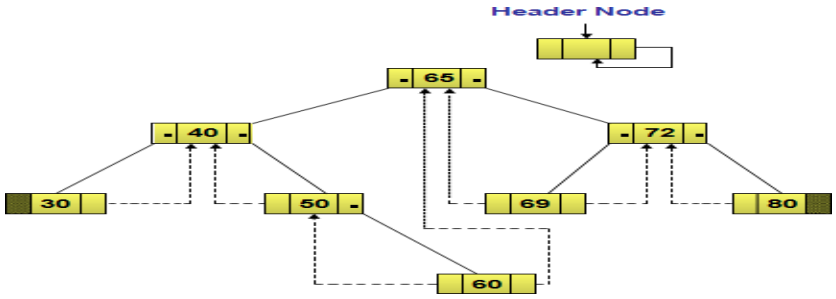


Similarly, node 80 does not have an inorder successor.

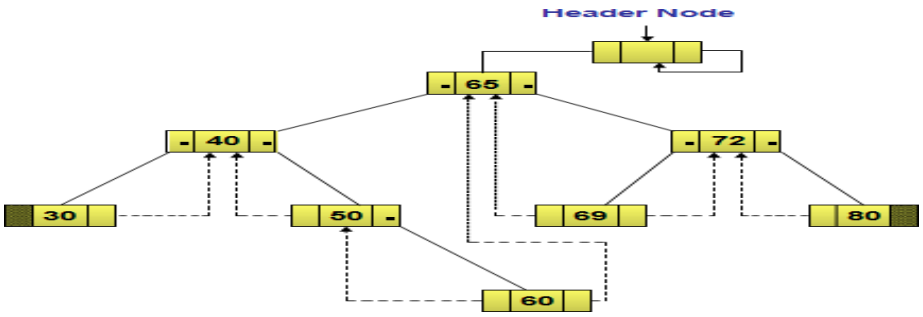
Two way Threaded Binary Trees with header Node

The right child of the header node always points to itself.

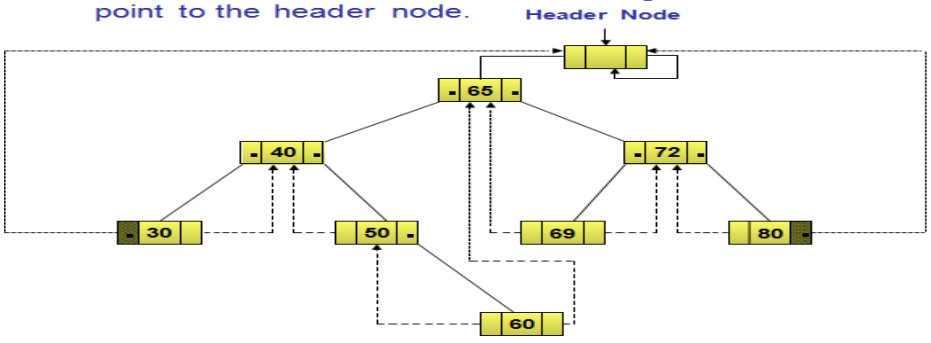
Therefore, you take a dummy node called the header node.



The threaded binary tree is represented as the left child of the header node.



- The left thread of node 30 and the right thread of node 80 point to the header node.



4.5 BINARY SEARCH TREES

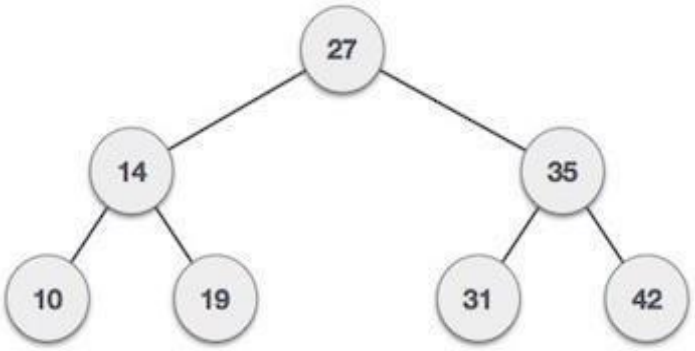
A binary search tree (BST) is a tree in which all nodes follows the below mentioned properties:

- The left sub-tree of a node has key less than or equal to its parent node's key.
- The right sub-tree of a node has key greater than or equal to its parent node's key.

Thus, a binary search tree (BST) divides all its sub-trees into two segments; *left* sub-tree and *right* sub-tree and can be defined as−

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

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has key and associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.An example of BST −



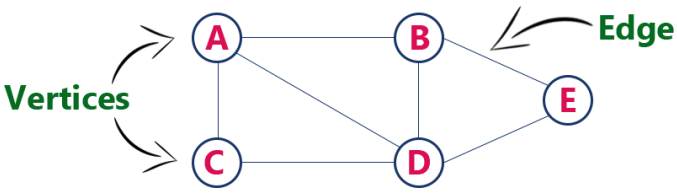
We observe that the root node key (27) has all less-valued keys on the left sub-tree and higher valued keys on the right sub-tree.

4.6 GRAPH

A graph is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G isrepresented as G = (V , E), where V is set of vertices and E is set of edges.

Example: graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}. This is a graph with 5 vertices and 6 edges.



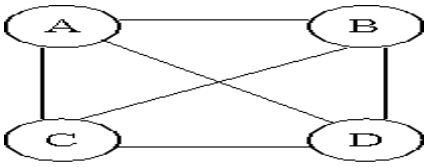
Graph Terminology

1. **Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.
2. **Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge isrepresented as(starting Vertex, ending Vertex). In above graph, the link between vertices A and B isrepresented as (A,B).

Types of Graphs

1.Undirected Graph

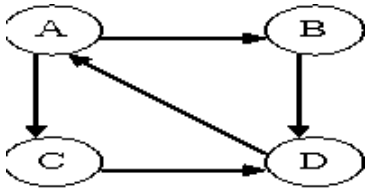
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

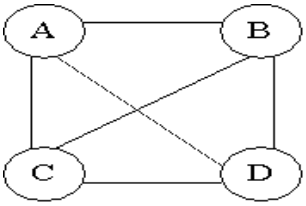
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

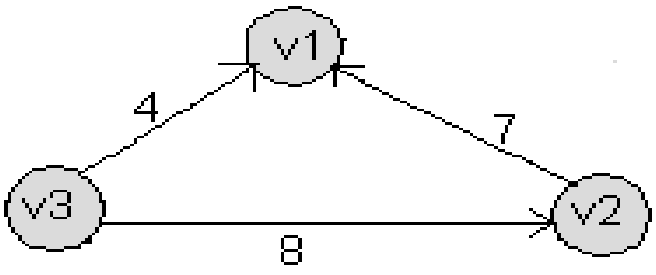
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

4.Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. Thevalue is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Adjacent nodes

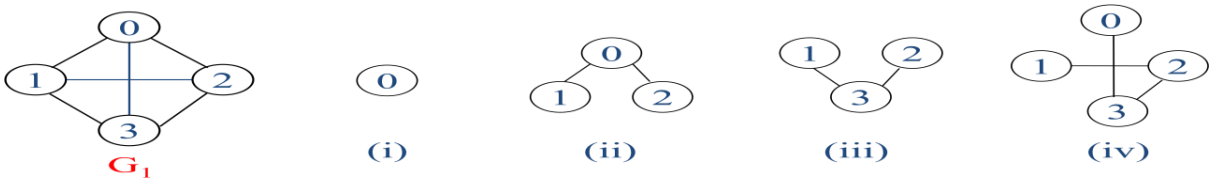
When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence

In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Sub Graph

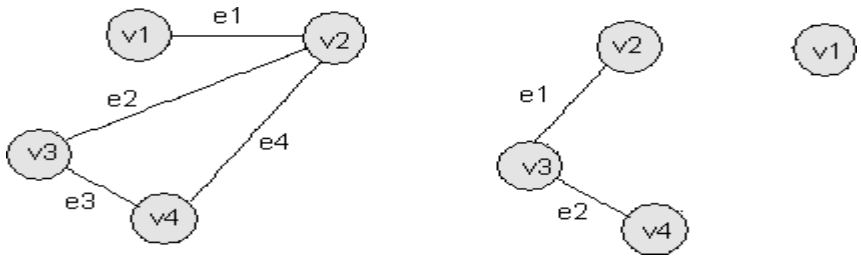
A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and eachedge ofS has the same end vertices in S as in G. A subgraph of G is a graph G' such that V(G') □ V(G) and E(G') □ E(G)



Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise, G is disconnected.

A connected graph G A disconnected graph G



This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

4.7SEQUENTIAL REPRESENTATION OF GRAPHS; ADJACENCY MARTIX;PATH MATRIX

1.Adjacency Matrix

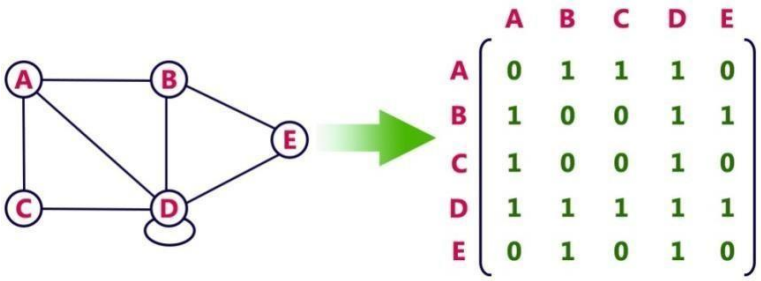
In this representation, graph can be represented using a matrix of size total number of vertices by totalnumber ofvertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

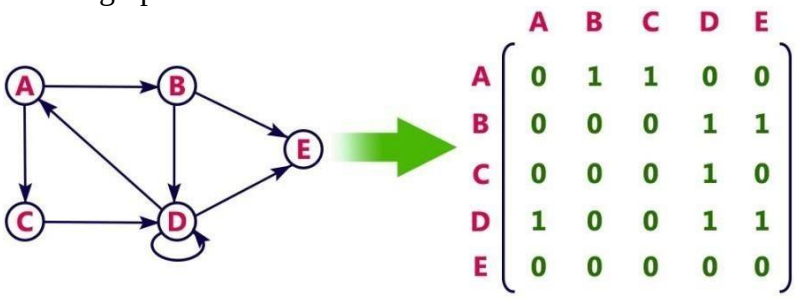
This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to columnvertex and 0represents there is no edge from row vertex to column vertex.

Adjacency Matrix : let G = (V, E) with n vertices, n □ 1. The adjacency matrix of G is a 2-dimensional n □ nmatrix, A, A(i, j) = 1 iff (v_i, v_j) □E(G) (□v_i, v_j□ for a diagraph), A(i, j)=0 otherwise.

example : for undirected graph



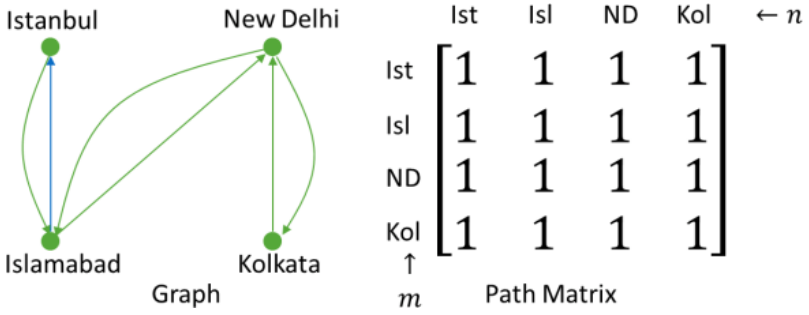
For a Directed graph



The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph neednot be symmetric.

2. Path Matrix in Graph Theory

Graph Theory is dependent on vertex (node) and edge (path) relationship. So, each and every processneeds path matrix in graph theory. To find a path between two vertex or node path matrix is the most easiest way. If you have a path matrix defined for a graph you can say whether a node can be traveledfrom another specific node.
Below is a real life Data Structure example of Path Matrix in Graph Theory.



This graph defines train routes among India, Pakistan and Turkey.
So, We can answer the following answer from the path matrix of the graph.
Is there a train route between New Delhi and Istanbul?Ans: Yes.
Is there a train route between Kolkata and Istanbul?Ans: Yes.
Is there a train route between Islamabad and Kolkata?Ans: Yes.
And that is how path matrix works.

4.8 TRAVERSING A GRAPH

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used todecide the order of vertices is visited in the search process. A graph traversal finds the edges to be usedin the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...
1. BFS (Breadth First Search)
2. DFS (Depth First Search)

1. BFS (Breadth First Search)

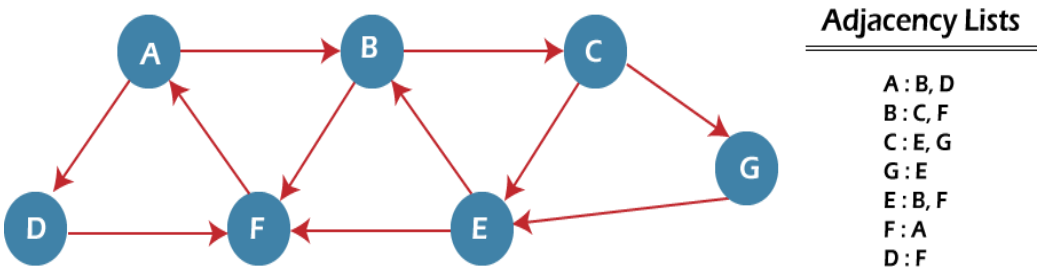
BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph withoutloops. We use Queue data structure with maximum size of total number of vertices in the graph toimplement BFS traversal.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.
Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue. Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insertthem into the Queue.
Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue thendelete that vertex.
Step 5 - Repeat steps 3 and 4 until queue becomes empty.
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges fromthe graph

Example of BFS algorithm

Now, let's understand the working of BFS algorithm by using an example. In the example given below,there is a directed graph having 7 vertices.



In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the

nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted fromQUEUE1.

Now, let's start examining the graph starting from Node A.

Step 1 - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A toqueue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B toqueue1.

1.QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1.The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will notinsert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

2.DFS algorithm

DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

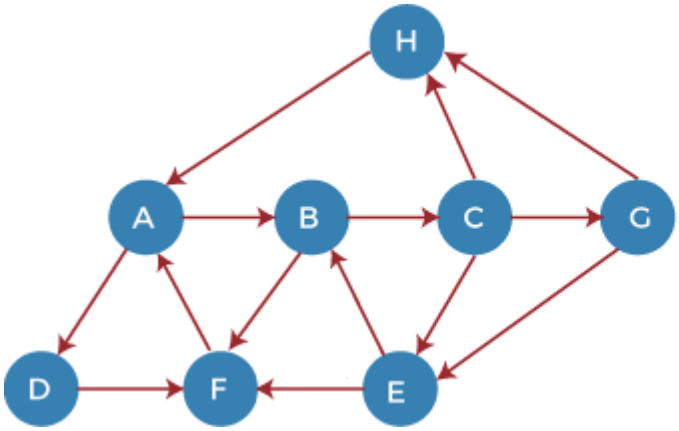
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and settheir STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.
1 STACK: H
Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto thestack that are in ready state.
1. Print: H
2. STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto thestack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto thestack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto thestack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto thestack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto thestack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that arein ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that arein ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Unit: V

Sorting and Searching : Sorting- Bubble sort, Insertion, Selection, Merge sort , Quick sort, Heap sort Searching: Liner search, Binary search.

Sorting

Sorting is the process of arranging the elements in the ascending or descending order. The default sorting is ascending order.

There are two types of sorting namely

1. External Sorting
2. Internal Sorting

External Sorting: It is used to handle massive amount of data for sorting. It is requied when the data is being sorted do not fit into the main memory of a computing device(usually main memory).

Merge sort (2 way merge sort is an example)

Internal Sorting: It is the type of sorting occur within a main memory. Internal sorting algorithm types are as follows

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merge sort
5. Quick sort
6. Heap sort

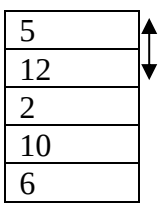
Bubble sort: In this the adjacent or adjoining values are compared and exchanged if they are not in the proper order. This process is repeated until the entire array is sorted.

Example: Consider the following numbers to be sort using bubble sort method

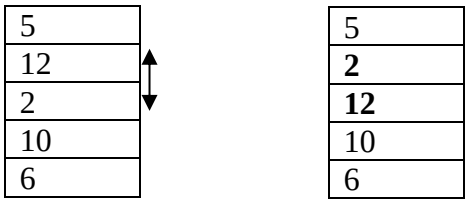
5 12 2 10 6

Pass 1

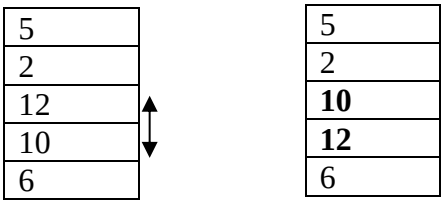
1. Compare 5 and 12 [**No change**]



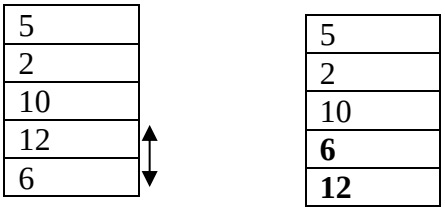
2. Compare 12 and 2 [Since 2 is smaller than 12 apply **swap**]



3. Compare 12 and 10 apply **swap**

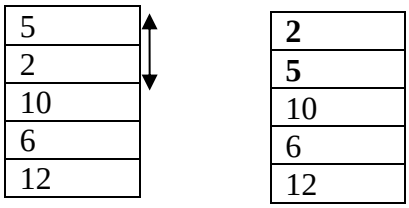


4. Compare 12 and 6 apply **swap**

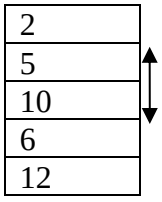


Pass 2

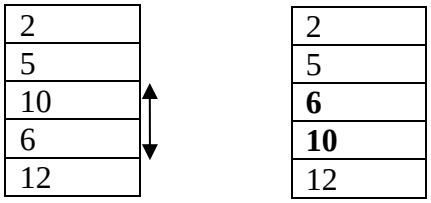
1. Compare 5 and 2 ,apply **swap**



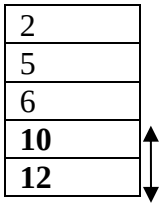
2. Compare 5 and 10 ,**No change**



3. Compare 10 and 6 apply **swap**



4. Compare 10 and 12, **No change**



Now we can find all the elements are sorted in the ascending order using bubble sort .

Bubble Sort Algorithm

1. Start
2. Read n [No .of elements]
3. Read data[] [No of elements]
4. Repeat steps 5,6 and 7 until 1 to n-1
5. Initially in a pass assume no interchange
   Interchange←FALSE
6. Repeat steps 7 8 9
7. If data[i]>data[i+1] then
   a. data[i]=data[i+1]
   b. interchange←TRUE

   // End of step 6
8. if(interchange==FALSE) then return
   //end of step 4
9. end

Insertion sort

In this each successive element is picked and inserted at an appropriate position in the previously sorted array.

Example: Consider the following numbers to be sort using insertion sort method

40 70 10 20 60 90

Process	Sorted Array	Unsorted Array
Pass1	----------	40 70 10 20 60 90
Pass2	40[Trivially sorted]	70 10 20 60 90
Pass3	40 70	10 20 60 90
Pass4	10 40 70	20 60 90
Pass5	10 20 40 70	60 90
Pass6	10 20 40 60 70	90
Pass7	10 20 40 60 70 90	-------

Sorted Array contains ascending order as follows

10	20	40	60	70	90

Insertion Sort Algorithm

1. Start
2. A[0]=minimum integer value /* Now start sorting the array*/
3. Repeat steps 4 through 9 for k=1,2,3….N-1
   {
4. Temp=A[k]
5. ptr=k-1
6. Repeat steps 7 to 8 while temp<A[ptr]
   {
7. A[ptr+1]=A[ptr] //Moves element forward
8. ptr=ptr-1
   }
9. A[ptr+1]=temp
   }
10. End

INSERTION SORT:

A[]

-∞	40	**70**	10	20	60	90
0	1	**2**	3	4	5	6

Pass-1

K=2, temp=70,ptr=1

70<40→false

Then A[2]=70

Pass-2

K=3, temp=10,ptr=2

10<70→true

Then swap→A[3]=70,ptr=1

10<40→true

swap→A[2]=40,ptr=0

10<-∞→false

Then A[1]=10

10	40	70	20	60	90

Pass-3

K=4, temp=20,ptr=3

20<70→true

Then swap→A[4]=70,ptr=2

20<40→true

Again swap→A[3]=40,ptr=1

20<10→false

Then A[2]=20

10	20	40	70	60	90

Pass-4

K=5, temp=60,ptr=4

60<70→true

Then swap→A[5]=70,ptr=3

60<40→false

Then A[4]=60

Pass-5

K=6, temp=90,ptr=5

90<70→false

Then A[6]=90

Final sorted values are

10	20	40	60	70	90

Selection Sort

In this the smallest or largest key from the remaining unsorted array us searched for and put in the sorted array.

Example: Consider the following numbers to be sort using insertion sort method

40 70 10 20 60 90

Process	Sorted Array	Unsorted Array
Pass1	----------	40 70 10 20 60 90
Pass2	10	40 70 20 60 90
Pass3	10 20	40 70 60 90
Pass4	10 20 40	70 60 90
Pass5	10 20 40 60	70 90
Pass6	10 20 40 60 90	90
Pass7	10 20 40 60 90	

In the above table in the unsorted array smallest element is picked and placed in the sorted array one by one for ascending order. For descending order largest element should be picked.

Selection Sort Algorithm

1.Repeat steps 2&3 for K=1,2,3,……N-1

2.Call MIN(A,K,N,LOC)

3.Temp=A[k],A[K]=A[LOC],A[LOC]=Temp

[End of loop]

4.Exit

MIN(A,K,NLOC)

1.MIN:=A[K],LOC=K

2.Repeat for J=k+1,k+2,……N

If MIN>A[J] then MIN=A[J] and LOC=J

[end loop]

3.Return

40	70	10	20	60	90
1	2	3	4	5	6

Pass-1

K=1,

MIN=40 LOC=1 J=2

40>70 false

J=3

40>10→ True

MIN=10 and LOC=3

J=4→10>20→false

J=5→10>60→false

J=6→10>90→false

10	40	70	20	60	90

Pass-2

K=2 MIN=40,LOC=2

J=3→40>70→false

MIN=40 LOC=3

J=4→40>20→True

MIN=20 LOC=4

J=5→20>60→false

J=6→20>90→false

10	20	40	70	60	90

Pass-3

K=3 MIN=40,LOC=3

J=4→40>70→false

J=5→40>60→false

J=6→40>90→false

0	20	40	70	60	90

Pass-4

K=4 MIN=70,LOC=4

J=5→70>60→true

MIN=60,LOC=5

J=6→60>90→false

10	20	40	60	70	90

Pass-5

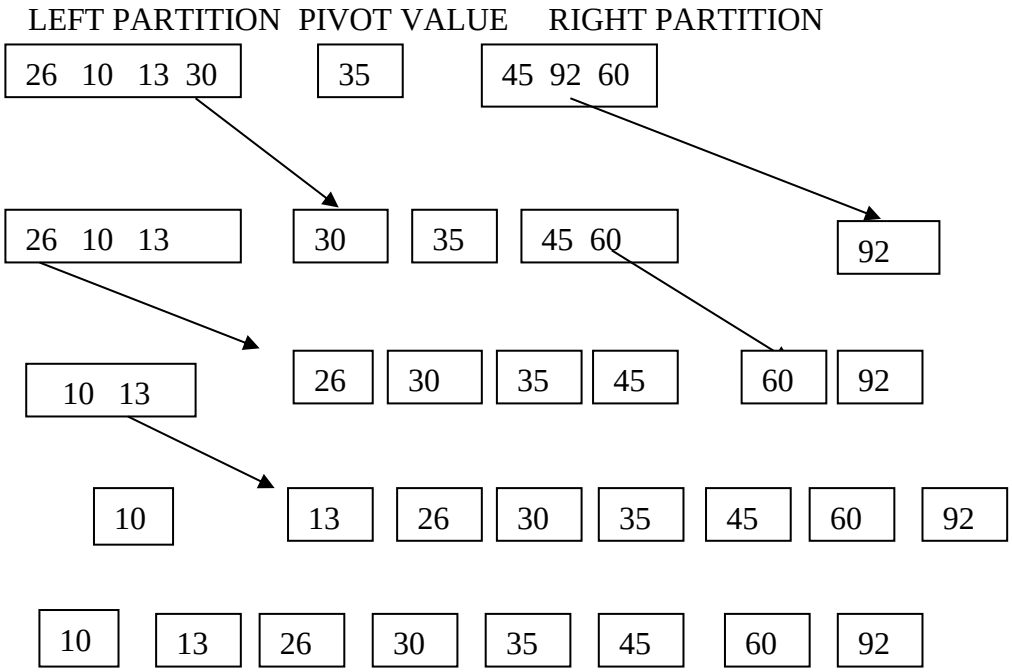
K=5 MIN=70 ,LOC=5

J=6→70>90→false

10	20	40	60	70	90

Quick Sort:- It is also called as partition exchange sort. In this we should place pivot element in the correct position [middle] and partition the remaining into two sets.
One set contains the number less than pivot element and other with greater.

Example: Consider the following list of numbers
35 26 10 13 45 92 30 60
Here 35 is the pivot element, so it should be placed at center



In the above diagram the following points should be observed

- Left partition contains the numbers less than pivot element. The partition may not be sorted.
- Right partition contains the numbers greater than pivot element. The partition may not be sorted.
- In the left partition the bigger number is removed and placed before the pivot element.
- In the right partition the bigger number is removed and placed at the last
- In each steps both numbers from left and right partition is taken and placed.
- Finally we will get the sorted elements.

Quick Sort Algorithm:- [QuickSort(data[],start,end)]

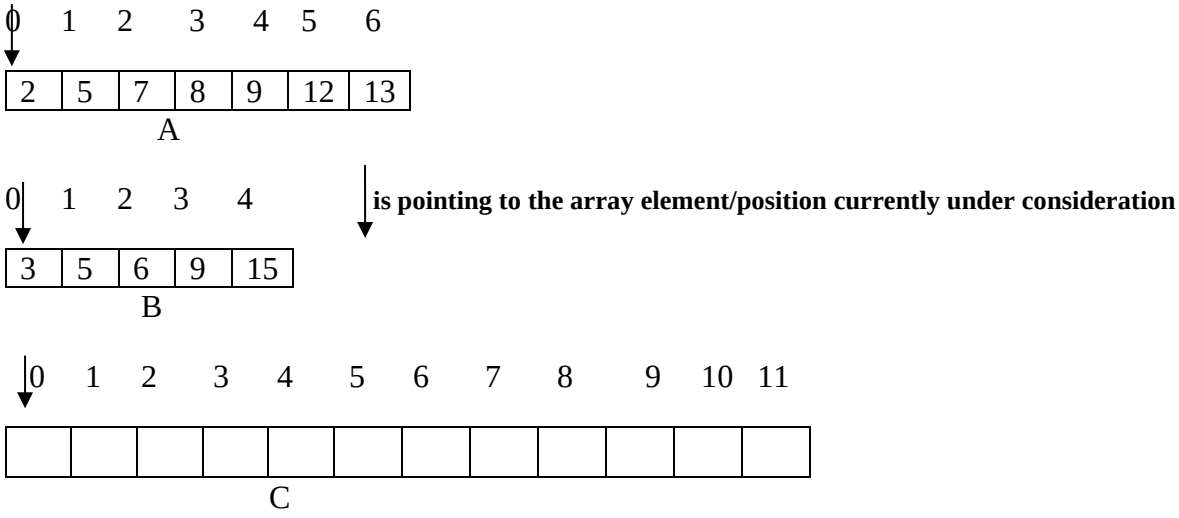
1. low←Start
2. high←End
3. elt←data[Low]
4. While(low<high)
    a. While(data[low]<=elt && (low<high)
       low←low+1
       While(data[high] >=elt &&(low<high)
    b. high←high+1
    c. if low<high then swap(data[low],data[high])
5. swap(data[start],data[high])
6. call QuickSort(data,start,high-1)
7. call QuickSort(data,high+1,end)

Merge Sort

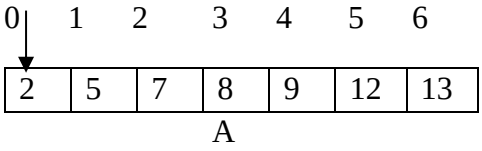
- ✓ Merging means combing elements of two arrays to form a new array.
- ✓ Simplest way of merging two arrays is first copy all the elements of one array into new array and then copy all the elements of other array into new array .Then sort the new array. Another popular technique to have a sorted array while merging. It is called merge sort.

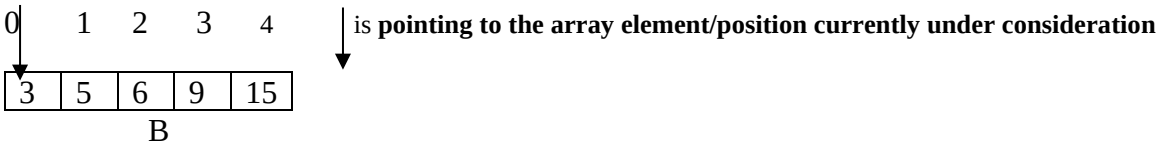
Example

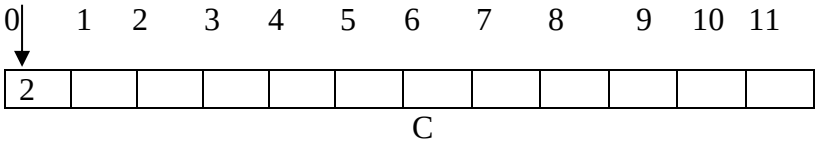
Step1:- Let us consider following two arrays A[7] and B[5] are to be merged to form a new array. The new array say C will have 7+5=12 elements.



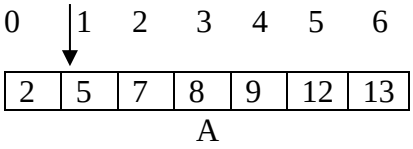
Step2:- Compare A[0] and B[0]; since A[0] <B[0], Move A[0] to C[0]. Move the pointers if array A and array C

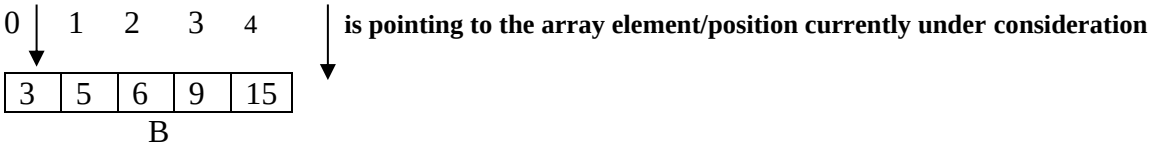


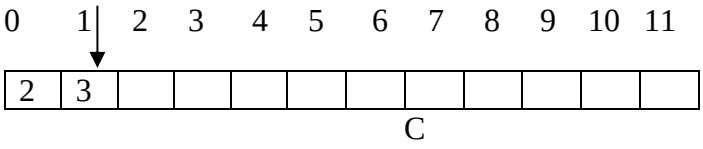




Step3:- Compare A[1] and B[0]; Now B[0]<A[1]; Move B[0] to C[1] and move the pointer to next element in an array.







Step 4:- Continuing the same way, the resultant array C is having all the elements of C in sorted manner.

Merge Sort Algorithm

1. ctrA=L1; ctrB=L2; ctrC=L3
2. while ctrA<=U1 and ctrB<=U2 perform steps 3 through 10
   {
3. if A[ctrA] <=B[ctrB] then] then
   {
4. C[ctrC]=A[ctrA]
5. ctrC=ctrC+1
6. ctrA=ctrA+1
   }
7. Else
   {
8. C[ctrC]=B[ctrB]
9. ctrC=ctrC+1
10. ctrB=ctrB+1
   }
   } /*end of while loop */

11. If ctrA > U1 then
   {
12. While ctrB<=U2 perform steps 13 through 15
13. { C[ctrC]=B[ctrB]
14. ctrC=ctrC+1
15. ctrB=ctrB+1
   }}
16. If ctrB>U2 then
17. { while ctrA <=U1 perform steps 18 to 20
18. { C[ctrC]=A[ctrA]
19. ctrC=ctrC+1
20. ctrA=ctrA+1
   }
   }

Heap Sort

- ✓ The heap is used in an elegant sorting algorithm called heap sort
- ✓ Let 'H' be heap and 'N' be the node. 'H' is maintained in the memory by linear array using sequential representation not the linked representation.
- ✓ 'H' is called heap , if each node N of H should satisfy the following property
  - The value at N is greater than or equal to the value at any of the descendants.

Steps in Heap Sort

1. Building the heap tree
2. Repeatedly deletion the root element and place in the array.

Example

Apply heap sort for the following numbers

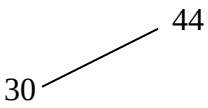
44 30 50 22 60 55 77

STEP 1 [Building the heap tree]

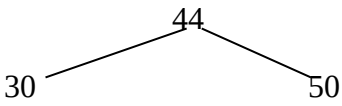
a) **Insert 44**

44

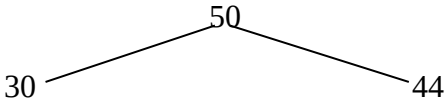
b) **Insert 30**



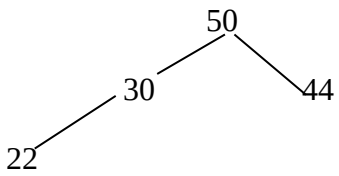
c) **Insert 50**



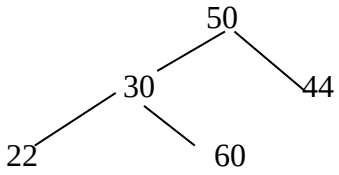
Apply Reheap since 44 is less than 50



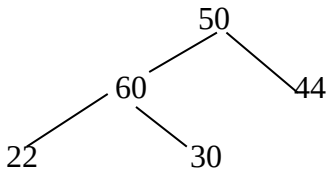
d) **Insert 22**



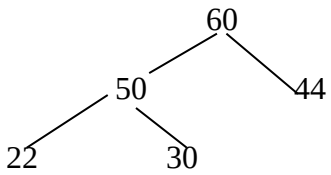
e) **Insert 60**



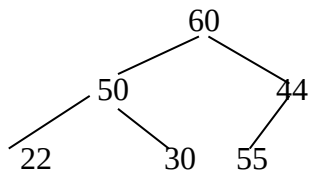
Apply Reheap since 60 is greater than 30



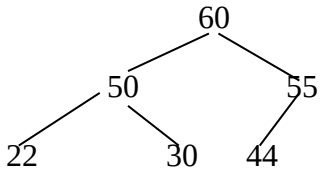
Once again apply reheap since 60 is greater than 50 also



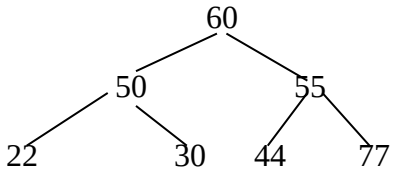
f) **Insert 55**



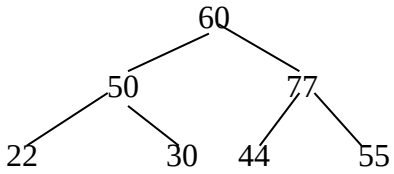
Apply reheap since 55 is greater than 44



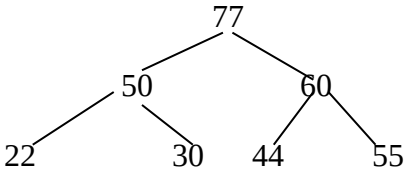
g) Insert 77



Apply reheap since 77 is greater than 55



Once again apply reheap since 77 is greater than 60 also

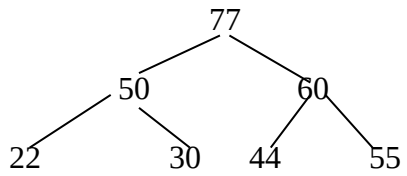


The above tree is called heap tree.

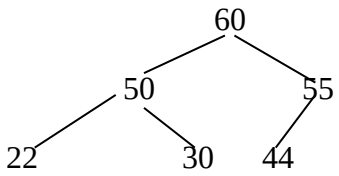
- In the heap tree we can observe that the root element is the greater number among the given number
- By repeatedly deleting the root element in the heap tree we can perform the sorting.

STEP2[Repeatedly deleting the root element]

Deleting the root element continuously and stored in the array as follows

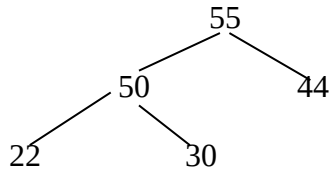


Delete 77



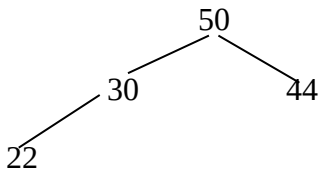
						77

Delete 60



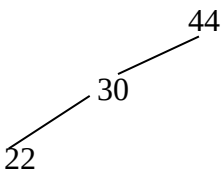
					60	77

Delete 55



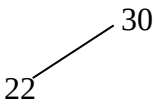
				55	60	77

Delete 50



			50	55	60	77

Delete 44



		44	50	55	60	77

Delete 30

22

	30	44	50	55	60	77

Delete 22

22	30	44	50	55	60	77

Now the array contains sorted elements.

Heap Sort Algorithm

1. Start
2. For i= 1 to n
    a. Insert data[i] into heap
3. For i=n-1 down to 1
    a. data[i]←delete max from heap /* delete heap include reheaping also.
4. end

Searching:

Searching is a technique to find the data element is present in the data structure or not.

The following are the searching techniques.

1. **Linear Search or Sequential Search.**
2. **Binary Search.**

Linear Search or Sequential Search

In linear search each element is compared with the given item to be searched for one by one. This method which traverses the array sequentially to locate the given item, is called Linear search or sequential search. It is applicable to both sorted and unsorted arrays

Algorithm

/* Initialize counter by assigning lower bound value of the array */

1. Set ctr=L //In C++ ,L is 0
   /* Search for the ITEM */
2. Repeat steps 3 through 4 until ctr >U
3. If Ar[ctr]==ITEM then
   { print "Search Succesfull"
   printf ctr,"is the location of ", ITEM
   break
   }
4. ctr=ctr+1
   /*end of repeat */
5. If ctr>U then
   printf "Search unsuccessful"
6. end

Example: Write the steps to find no 25 in the following array namely **A**

0	1	2	3	4	5
10	45	25	90	89	08

Lower Bound (L)=0 UpperBound(U)=5

ITEM=25 [number to be searched]

STEP 1

ctr=L→ ctr=0
while(0>5)
{
If (A [0] == ITEM)
10 25
// since the above condition is false the IF block will not execute
}
ctr=ctr+1→0+1=1

STEP 2

Now ctr=1

while(1<5)
{
If A[1]==ITEM)
45 ==25
// since the above condition is false the IF block will not execute
}
ctr=ctr+1→1+1=2

STEP 3

Now ctr=2
while(2<5)
{
If A[2]==ITEM)
25 ==25
PRINT "SEARCH SUCCESSFUL"
exit // program will be terminated after the execution of this statement
}
Now we get the output

"SEARCH SUCCESSFUL" 2 is the location of the element 25

Binary Search

Binary Search is the popular search technique which searches the given ITEM in minimum possible comparisons. The binary search requires the array, to be scanned , must be sorted in any order(for instance , it may be ascending order). In binary search, the ITEM is searched for in smaller segment (nearly half the previous segment) after each stage. For the first stage, the segment will contain the entire array.

To search for ITEM in a sorted array(in ascending order), the ITEM is compared with middle element of the segment(i.e., in the entire array for the first time. If the ITEM is more than the middle element, latter part of the segment becomes new segment to be scanned; if the item is less than the middle element, former part becomes new segment to be scanned, the same process is repeated for the new segment(s) until either the ITEM is found(search successful) or the segment is reduced to the single element and still the ITEM is not found(search unsuccessful.

Algorithm

Case 1: Array A [L:U] is stored in the ascending order

/* Initialize the segment variables */

1. Set beg=L, last=U
2. REPEAT steps 3 through 6 UNTIL beg>last
3. mid=INT(beg+last/2)
4. If [mid]==ITEM then
   { print “Search successful”
      print ITEM, “found at “,mid
      break
5. If A[mid]<ITEM then
      beg=mid+1
6. If A[mid] >ITEM then
      Last=mid-1
   /*end of repeat */
7. If beg !=last
   print “Unsuccessful search”
8. End

Case 2: Array A[L:U] is stored in the descending order

1. Set beg=L, last=U
2. REPEAT steps 3 through 6 UNTIL beg>last
3. Mid=INT(beg+last/2)
4. If [mid]==ITEM then

{ print "Search successful"
print ITEM, "found at ",mid
break

5. If A[mid]<ITEM then
last=mid+1
6. If A[mid] >ITEM then
beg=mid+1

/*end of repeat */

7. If beg !=last

print "Unsuccessful search"

8. End

Example: Write the steps to search 44 in the following array

1	2	3	4	5	6	7	8	9	10	11	12
10	12	14	21	23	28	31	37	42	44	49	53

beg=1 last=12

Solution:

Step 1

beg=1 last=12
mid=INT(1+12)/2=int (6.5)=6

1	2	3	4	5	6	7	8	9	10	11	12
					28						

Step 2

data [mid] i.e., data[6] is 28
28<44 then
beg=mid+1→6+1=7

Step 3

7	8	9	10	11	12
		42			

beg=7 last=12
Now mid=INT(beg+last)/2=int(7+12/2)=int(9)=9
data [mid] i.e., data[9] is 42
42<44 then
beg=mid+1→9+1=10

Step 4

10	11	12
	49	

beg=10 last=12
mid=INT(10+12/2)=int(11)=11
data [mid] i.e., data[11] is 49
49>44 then
last=mid-1→11-1=10

Step 5
mid=INT(beg+last)/2=10+10/2=20/2=10
data[10] ie., 44=44
Search successful at the location number 10

Reference

1. Seymour Lipschutz (Schaum's Series), Data Structures, McGraw Hill Education (India) Private Limited Ltd., New Delhi, Revised First Edition, 2013.

2. https://www.geeksforgeeks.org/stack-data-structure/

3. **http://www.it.griet.ac.in/wp-content/uploads/2014/08/UNIT-V_QA.pdf**

4. **https://medium.com/learning-python-programming-language/sorting-algorithms-insertion-sort-selection-sort-quick-sort-merge-sort-bubble-sort-4f23bda6f37a**