LECTURE NOTES

ON

DATA STRUCTURES THROUGH C

ACADEMIC YEAR 2021-22

I B.Tech.–II SEMESTER(R20)

V.Divya,Assistant Professor

DEPARTMENT OF HUMANITIES AND BASIC SCIENCES

V S M COLLEGE OF ENGINEERING
RAMCHANDRAPURAM
E.G DISTRICT
533255

I Year II Semester		L	T	P	C
		3	0	0	3
DATA STRUCTURES THROUGH C					

Preamble:
This course is core subject developed to help the student understand the data structure principles used in power systems, machines and control systems. This subject covers linear data structures, linked lists, trees, graphs, searching and sorting.

Course Objectives:

- Operations on linear data structures and their applications.
- The various operations on linked lists.
- The basic concepts of Trees, Traversal methods and operations.
- Concepts of implementing graphs and its relevant algorithms.
- Sorting and searching algorithms.

Unit-1: Linear Data Structures: Arrays, Stacks and Queues

Data Structures -Operations-Abstract Data Types-Complexity of Algorithms-Time and Space-Arrays-Representation of Arrays-Linear Arrays-Insertion–Deletion and Traversal of a Linear Array-Array as an Abstract Data Type-Multi-Dimensional Arrays-Strings-String Operations-Storing Strings-String as an Abstract Data Type

Stack -Array Representation of Stack-Stack Abstract Data Type-Applications of Stacks: Prefix-Infix and Postfix Arithmetic Expressions-Conversion-Evaluation of Postfix Expressions-Recursion-Towers of Hanoi-Queues-Definition-Array Representation of Queue-The Queue Abstract Data Type-Circular Queues-Dequeues-Priority Queues.

Unit-II: Linked Lists

Pointers-Pointer Arrays-Linked Lists-Node Representation-Single Linked List-Traversing and Searching a Single Linked List-Insertion into and Deletion from a Single Linked List-Header Linked Lists-Circularly Linked Lists-Doubly Linked Lists-Linked Stacks and Queues-Polynomials-Polynomial Representation-Sparse Matrices.

Unit-III: Trees

Terminology-Representation of Trees-Binary Trees-Properties of Binary Trees-Binary Tree Representations-Binary Tree Traversal-Preorder-In-order and Post-order Traversal-Threads-Thread Binary Trees-Balanced Binary Trees-Heaps-Max Heap-Insertion into and Deletion from a Max Heap-Binary Search Trees-Searching-Insertion and Deletion from a Binary Search Tree-Height of Binary Search Tree, m-way Search Trees, B-Trees.

Unit-IV: Graphs

Graph Theory Terminology-Graph Representation-Graph Operations-Depth First Search-Breadth First Search-Connected Components-Spanning Trees-Biconnected Components-Minimum Cost Spanning Trees-Kruskal's Algorithm-Prism's Algorithm-Shortest Paths-Transitive Closure-All-Pairs Shortest Path-Warshall's Algorithm.

Unit-V: Searching and Sorting

Searching -Linear Search-Binary Search-Fibonacci Search-Hashing-Sorting-Definition-Bubble Sort-Insertion sort-Selection Sort-Quick Sort-Merging-Merge Sort-Iterative and Recursive Merge Sort-Shell Sort-Radix Sort-Heap Sort.

Course Outcomes:

After the completion of the course the student should be able to:

- data structures concepts with arrays, stacks, queues.
- linked lists for stacks, queues and for other applications.
- traversal methods in the Trees.
- various algorithms available for the graphs.
- sorting and searching in the data ret retrieval applications.

Text Books:

1. Fundamentals of Data Structures in C, 2nd Edition, E.Horowitz, S.Sahni and Susan Anderson Freed, Universities Press Pvt. Ltd.
2. Data Structures with C, Seymour Lipschutz, Schaum's Outlines, Tata McGraw Hill.

VSM COLLEGE OF ENGINEERING

RAMACHANDRAPURAM

DEPARTMENT OF BASIC SCIENCES AND HUMANITIES

Course Title	Year/Sem	Branch	Periods per Week
DATA STRUCTURES THROUGH C	1/11	EEE BRANCH	6

Course Outcomes:

- Operations on linear data structures and their applications
- The various operations on linked lists.
- The basic concepts of Trees, Traversal methods and operations.
- Concepts of implementing graphs and its relevant algorithms.
- Sorting and searching algorithms.

Unit No	**Outcomes**	**Name of the Topic**	**No. of Periods required**	**Total Periods**	**Reference Book**	**Methodology to be adopted**
		Unit-1				
I	CO 1	Data Structures -Operations-Abstract Data Types	1	15	T1, T2 R20	Black Board
		Complexity of Algorithms-Time and Space	2			Black Board
		Arrays-Representation of Arrays-Linear Arrays-Insertion–Deletion	1			Black Board
		d Traversal of a Linear Array-Array as an Abstract Data	1			Black Board
		Multi-Dimensional Arrays	1			Black Board
		Strings-String Operations Storing Strings-String as an Abstract Data Type	1			Black Board
		Stack -Array Representation of Stack-Stack Abstract Data Type	2			Black Board
		Applications of Stacks: Prefix Infix and Postfix	1			
		Arithmetic Expressions-Conversion-Evaluation of Postfix Expressions	1			Black Board
		Recursion-Towers of Hanoi	1			Black Board
		Queues-Definition-Array Representation of Queue-The Queue Abstract Data Type	1			Black Board
		Circular Queues- Dequeue -Priority Queues.	2			Black Board

		Unit-2				
II	CO2	Pointers-Pointer Arrays-Linked Lists-Node Representation-Single Linked List-Traversing and Searching a Single Linked List	2	12	T1, T2 R20	Black Board
		-Insertion into and Deletion from a Single Linked List-Header Linked Lists	2			Black Board
		Circularly Linked Lists	2			Black Board
		Doubly Linked Lists-Linked	2			Black Board
		Stacks and Queues	2			Black Board
		Polynomials-Polynomial Representation-Sparse Matrices.	2			Black Board
		Unit-3				
III	CO3	Terminology-Representation of Trees-Binary Trees-Properties of Binary Trees	2	12	T1, T2 R20	Black Board
		Binary Tree Representations-Binary Tree Traversal-Preorder-In-order and Post-order Traversal	2			Black Board
		Threads Thread Binary Trees-Balanced Binary Trees	2			Black Board
		Heaps-Max Heap-Insertion into and Deletion from a Max Heap	2			Black Board
		Binary Search Trees-Searching-Insertion and Deletion from a Binary Search Tree	2			
		Height of Binary Search Tree, m-way Search Trees, B-Trees.	2			Black Board

IV	CO4	**Unit-4**		12		
		Graph Theory Terminology-Graph Representation	2			Black Board
		Graph Operations-Depth First Search-Breadth First Search	2			Black Board
		Connected Components-Spanning Trees- Bi-connected Components	2			Black Board
		Minimum Cost Spanning Trees-Kruskal's Algorithm	2			Black Board
		Prism's Algorithm-Shortest Paths	2			Black Board
		Transitive Closure- AllPairs Shortest Path -Warshall's Algorithm.	2			Black Board

V	CO5	**Unit-5**		08	T1, T2 R20	
		Searching -Linear Search-Binary Search Fibonacci Search	2			Black Board
		Hashing Sorting-Definition-Bubble Sort	2			Black Board
		Insertion sort-Selection Sort Quick Sort-Merging	2			E-Classroom
		Merge Sort-Iterative and Recursive Merge Sort Shell Sort-Radix Sort-Heap Sort.	2			Black Board

Text Books:

1. Fundamentals of Data Structures in C, 2nd Edition, E.Horowitz, S.Sahni and Susan Anderson Freed, Universities Press Pvt. Ltd.
2. Data Structures with C, Seymour Lipschutz, Schaum's Outlines, Tata McGraw Hill.

Course Outcomes:

After the completion of the course the student should be able to:

- Data structures concepts with arrays, stacks, queues.
- Linked lists for stacks, queues and for other applications.
- Traversal methods in the Trees.
- Various algorithms available for the graphs.
- Sorting and searching in the data ret retrieval applications.

Faculty Member **Head of the Department** **Principal**

MODULE 1: INTRODUCTION

DATA STRUCTURES

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms

Basic Terminology: Elementary Data Organization:

Data: Data are simply values or sets of values.

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

Data items: Data items refers to a single unit of values.

Data items that are divided into sub-items are called **Group items**. Ex: An Employee Name may be divided into three subitems- first name, middle name, and last name.

Data items that are not able to divide into sub-items are called **Elementary items.** Ex: SSN

Entity: An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.

Ex:	Attributes-	Names,	Age,	Sex,	SSN
	Values-	Rohland Gail,	34,	F,	134-34-5533

Entities with similar attributes form an **entity set**. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute.

The term “information” is sometimes used for data with given attributes, of, in other words meaningful or processed data.

Field is a single elementary unit of information representing an attribute of an entity.

Record is the collection of field values of a given entity.

File is the collection of records of the entities in a given entity set.

Each record in a file may contain many field items but the value in a certain field may uniquely determine the record in the file. Such a field K is called a primary key and the values k1, k2, ….. in such a field are called keys or key values.

Records may also be classified according to length.

A file can have fixed-length records or variable-length records.

- In fixed-length records, all the records contain the same data items with the same amount of space assigned to each data item.

- In variable-length records file records may contain different lengths.

Example: Student records have variable lengths, since different students take differe nt numbers of courses. Variable-length records have a minimum and a maximum length.

The above organization of data into fields, records and files may not be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures.

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally classified into

- Primitive data Structures

- Non-primitive data Structures

1. **Primitive data Structures:** Primitive data structures are the fundamental data types which are supported by a programming language. Basic data types such as integer, real, character and Boolean are known as Primitive data Structures. These data types consists of characters that cannot be divided and hence they also called simple data types.

Non- Primitive data Structures: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs Based on the structure and arrangement of data, non-primitive data structures is further classified into

1. Linear Data Structure

2. Non-linear Data Structure

1. Linear Data Structure:

A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

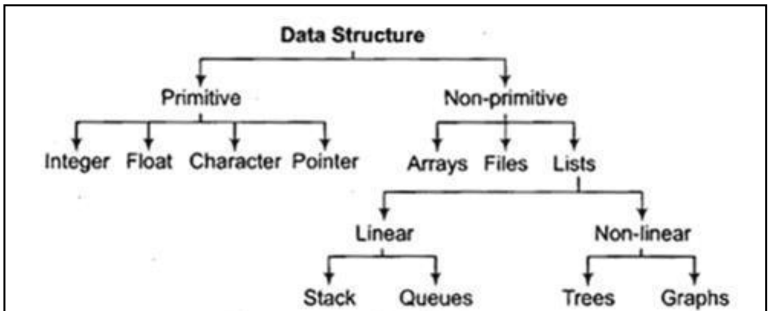
1. One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.

2. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

2. Non-linear Data Structure:

A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.



Data structure classification

Arrays:

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2, 3 . . . . *n*. if **A** is chosen the name for the array, then the elements of **A** are denoted by subscript notation a1, a2, a3….. an

by the bracket notation A [1], A [2], A [3] . . . . . . A [n]

Trees

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or a tree. Some of the basic properties of tree are explained by means of examples

1. Stack: A stack, also called a fast-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the top. This structure is similar in its operation to a stack of dishes on a spring system as shown in fig.

Note that new 4 dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the Stack

Queue: A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "from'' of the list, and insertions can take place only at the other end of the list, the “rear” of the list.

This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. the first person in line is the first person to board the bus. Another analogy is with automobiles waiting to pass through an intersection the first car in line is the first car through.

Graph: Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines in Fig. The data structure which reflects this type of relationship is called a graph.

DATA STRUCTURES OPERATIONS

The data appearing in data structures are processed by means of certain operations.

The following four operations play a major role in this text:

1. **Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "**visiting**" the record.)

2. **Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.

3. **Inserting:** Adding a new node/record to the structure.

4. **Deleting:** Removing a node/record from the structure.

The following two operations, which are used in special situations:

1. **Sorting:** Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)

Merging: Combining the records in two different sorted files into a single sorted file

Traversing in Linear Array

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms.
Traverse − print all the array elements one by one.or process the each element one by one . Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A with given property. This can be accomplished by traversing A, that is, by accessing and processing (frequently called visiting) each element of An exactly once.

Algorithm

```
Step 1 :   [Initialization]  Set I = LB

Step 2 :    Repeat Step 3 and Step 4 while I  < = UB

step 3 :     [ processing ] Process the A[I] element
```

Step 4 :   [Increment the counter] I = I + 1
[End of the loop of step 2]

Inserting

- Let A be a collection of data elements stored in the memory of the computer. Inserting refers to the operation of adding another element to the collection A.

- Inserting an element at the “end” of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.

- Inserting an element in the middle of the array, then on average, half of the elements must be moved downwards to new locations to accommodate the new element and keep the order of the other elements.

Algorithm:

INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K ≤ N. This algorithm inserts an element ITEM into the K^{th} position in LA.

```
1. [Initialize counter]              set J:= N
2. Repeat step 3 and 4               while J ≥ K
3.    [Move $J^{th}$ 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
```

Searching

search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Deleting

- Deleting refers to the operation of removing one element to the collection A.

- Deleting an element at the "end" of the linear array can be easily done with difficulties.

- If element at the middle of the array needs to be deleted, then each subsequent elements be moved one location upward to fill up the array.

Algorithm

DELETE (LA, N, K, ITEM)

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

```
1. Set ITEM:= LA[K]
2. Repeat for J = K to N – 1
        [Move J + 1 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
```

Sorting

Sorting refers to the operation of rearranging the elements of a list. Here list be a set of n elements. The elements are arranged in increasing or decreasing order.

Ex: suppose A is the list of n numbers. Sorting A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that,

$$A[I] < A[2] < A[3] < ... < A[N]$$

For example, suppose A originally is the list

8, 4, 19, 2, 7, 13, 5, 16

After sorting, A is the list

2, 4, 5, 7, 8, 13, 16, 19

Bubble Sort

Suppose the list of numbers A[l], A[2], ... , A[N] is in memory. The bubble sort algorithm works as follows:

Algorithm: Bubble Sort – BUBBLE (DATA, N)

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

DATA.

```
1.  Repeat Steps 2 and 3 for K = 1 to N - 1.
2.          Set PTR: = 1.                     [Initializes pass pointer PTR.]
3.          Repeat while PTR ≤ N - K:         [Executes pass.]
                (a) If DATA[PTR]  > DATA[PTR  + 1], then:
                        Interchange DATA  [PTR] and DATA [PTR + 1].
                        [End of If structure.]
                (b) Set PTR: = PTR + 1.
```

```
            [End of inner loop.]

        [End of Step 1 outer loop.]

    4. Exit.
```

Merge two arrays

1. Create an array arr3[] of size n1 + n2.
2. Simultaneously traverse arr1[] and arr2[].
    - Pick smaller of current elements in arr1[] and arr2[], copy this smaller element to next position in arr3[] and move ahead in arr3[] and the array whose element is picked.
3. If there are are remaining elements in arr1[] or arr2[], copy them also in arr3[].

Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type. Now we’ll define three ADTs namely List ADT, StackADT, Queue ADT.

List ADT

A list contains elements of same type arranged in sequential order and following operations can be performed on the list.

get() – Return an element from the list at any given position.

insert() – Insert an element at any position of the list.

remove() – Remove the first occurrence of any element from a non-empty list.

removeAt() – Remove the element at a specified location from a non-empty list.

replace() – Replace an element at any position by another element.

size() – Return the number of elements in the list.

isEmpty() – Return true if the list is empty, otherwise return false.
isFull() – Return true if the list is full, otherwise return false.

Stack ADT

A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:
push() – Insert an element at one end of the stack called top.
pop() – Remove and return the element at the top of the stack, if it is not empty.
peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
size() – Return the number of elements in the stack.
isEmpty() – Return true if the stack is empty, otherwise return false.
isFull() – Return true if the stack is full, otherwise return false

Algorithm INTRODUCTION

What is an Algorithm?

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Formal Definition:

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
Properties of an algorithm

INPUT Zero or more quantities are externally supplied.
OUTPUT At least one quantity is produced.
DEFINITENESS each instruction is clear and unambiguous.
FINITENESS if we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
EFFECTIVENESS every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Performance of a program: time and space tradeoff

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.
The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:
Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.
Data space: Data space is the space needed to store all constant and variable values.
Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.
Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- the target computer.

Algorithm Design Goals

The three basic design goals that one should strive for in a program are:
1. Try to save Time
2. Try to save Space
3. Try to save Face

a program that runs faster is a better program, so saving time is an obvious goal. Likewise, a program that saves space over a competing program is considered desirable. We want to “save face” by preventing the program from locking up or generating reams of garbled data.

Algorithm Specification

Algorithm can be described in three ways.

1. Natural language like English: When this way is choosed care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithm is small& simple.

3. Pseudo-code Method: In this method, we should typically describe algorithms as program, which resembles programming language constructs

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Bocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

```
Node. Record { data type – 1 data-1; . . data type – n data – n; node * link;
.                                        }
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.
   - Logical Operators AND, OR, NOT

Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed. For, while and repeat-until

While Loop:

```
While < condition > do

{ <statement-1> . . . <statement-n>  }
```

For Loop:

```
For variable: = value-1 to value-2 step step do

{ <statement-1> . . . <statement-n>  }
```

repeat-until:

```
repeat <statement-1> . . . <statement-n>  until<condition>
```

8. A conditional statement has the following forms.
   - If <condition> then <statement>
   - If <condition> then <statement-1> Else <statement-1>
   - **Case statement:**

```
Case

{ : <condition-1> : <statement-1>

. . .

: <condition-n>  : <statement-n>

: else : <statement-n+1>

 }
```

Input and output are done using the instructions read & write.

Orders Of Growth

♦A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.

♦When we have to compute, for example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid‘s algorithm is compared to the other two algorithms discussed in previous section or even why we should care which of them is faster and by how much. It is only when we have to find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important.

For large values of *n*, it is the function‘s order of growth that counts:

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size ‘n’ of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size ‘n’. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size ‘n’ of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.

2. Average Case : The expected value of f(n).

3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.
Asymptotic notations
The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

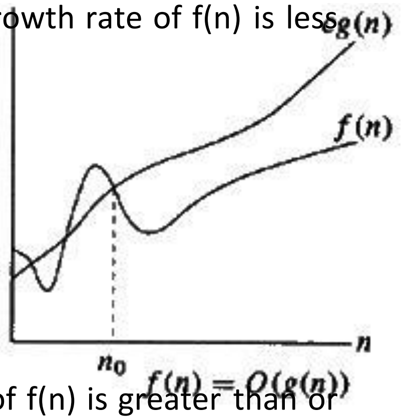
1. Big–OH (O)

2. Big–OMEGA (Ω),

3. Big–THETA (θ) and

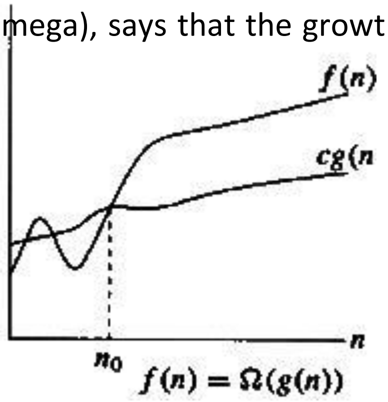
Big–OH O (Upper Bound)

f(n) ≤O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (≤) that of g(n).



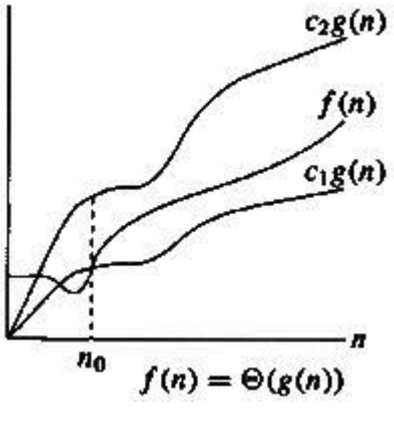
Big–OMEGA (Lower Bound)

f(n) ≥ (g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (≥) that of g(n).



Big–THETA (Same order)

g1(n)≤f(n)≤g2(n) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = (g(n)].



STRING

BASIC TERMINOLOGY:

Each programming languages contains a character set that is used to communicate with the computer. The character set include the following:

Alphabet:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits:	0 1 2 3 4 5 6 7 8 9
Special characters:	+ - / * ( ) , . $ = ‘ _ (Blank space)

String: A finite sequence S of zero or more Characters is called string.

Length: The number of characters in a string is called length of string.

Empty or Null String: The string with zero characters.

Concatenation: Let S1 and S2 be the strings. The string consisting of the characters of S1 followed by the character S2 is called Concatenation of S1 and S2. Ex: ‘THE’ // ‘END’ = ‘THEEND’

‘THE’ // ‘ ’ // ‘END’ = ‘THE END’

Substring: A string Y is called substring of a string S if there exist string X and Z such that S = X // Y // Z

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

Ex: ‘BE OR NOT’ is a substring of ‘TO BE OR NOT TO BE’
‘THE’ is an initial substring of ‘THE END’

STRINGS IN C

In C, the strings are represented as character arrays terminated with the null character \0.

Declaration 1:

```
#define MAX_SIZE  100              /* maximum size of string */
char s[MAX_SIZE]     = {“dog”};
char t[MAX_SIZE]     = {“house”};
```

s[0]	s[1]	s[2]	s[3]
d	o	g	\0

t[0]	t[1]	t[2]	t[3]	t[4]	t[4]
h	o	u	s	e	\0

The above figure shows how these strings would be represented internally in memory

Declaration 2:

char s[ ] = {“dog”};

char t[ ] = {“house”};

Using these declarations, the C compiler will allocate just enough space to hold each word including the null character.

STORING STRINGS

Strings are stored in three types of structures

1. Fixed length structures
2. Variable length structures with fixed maximum
3. Linked structures

Record Oriented Fixed length storage:

In fixed length structures each line of print is viewed as a record, where all have the same length i.e., where each record accommodates the same number of characters.

Example: Suppose the input consists of the program. Using a record oriented, fixed length storage medium, the input data will appear in memory as pictured below.

The main advantages of this method are

1. The ease of accessing data from any given record
2. The ease of updating data in any given record (as long as the length of the new data does not exceed the record length)

The main disadvantages are

1. Time is wasted reading an entire record if most of the storage consists of inessential blank spaces.
2. Certain records may require more space than available
3. When the correction consists of more or fewer characters than the original text, changing a misspelled word requires record to be changed.

Variable length structures with fixed maximum

The storage of variable-length strings in memory cells with fixed lengths can be done in two general ways

1. One can use a marker, such as two dollar signs ($$), to signal the end of the string
2. One can list the length of the string—as an additional item in the pointer array

Linked Storage

- Most extensive word processing applications, strings are stored by means of linked lists.
- In a one way linked list, a linearly ordered sequence of memory cells called nodes, where each node contains an item called a ***link,*** which points to the next node in the list, i.e., which consists the address of the next node.

STRING OPERATION

Substring

Accessing a substring from a given string requires three pieces of information:

(1) The name of the string or the string itself

(2) The position of the first character of the substring in the given string

(3) The length of the substring or the position of the last character of the substring.

Syntax: SUBSTRING (string, initial, length)

The syntax denote the substring of a string S beginning in a position K and having a length L.

Ex: SUBSTRING ('TO BE OR NOT TO BE', 4, 7) = 'BE OR N'

SUBSTRING ('THE END', 4, 4) = ' END'

Indexing

Indexing also called pattern matching, refers to finding the position where a string pattern P first appears in a given string text T. This operation is called INDEX

Syntax: INDEX (text, pattern)

If the pattern P does not appears in the text T, then INDEX is assigned the value 0.

The arguments "text" and "pattern" can be either string constant or string variable.

Concatenation

Let S1 and S2 be string. The concatenation of S1 and S2 which is denoted by S1 // S2, is the string consisting of the characters of S1 followed by the character of S2. Ex:

(a) Suppose S1 = 'MARK' and S2= 'TWAIN' then

S1 // S2 = 'MARKTWAIN'

Concatenation is performed in C language using ***strcat*** function as shown below strcat (S1, S2);

Concatenates string S1 and S2 and stores the result in S1

strcat ( ) function is part of the ***string.h*** header file; hence it must be included at the time of pre- processing

C Program to Concat Two Strings without Using Library Function

```
#include<stdio.h>
#include<string.h>
void concat(char[], char[]);
int main() {
    char s1[50], s2[30];
    printf("\nEnter String 1 :");
    gets(s1);
    printf("\nEnter String 2 :");
    gets(s2);
    concat(s1, s2);
    printf("\nConcated string is :%s", s1);
    return (0);
}
void concat(char s1[], char s2[]) {
    int i, j;
    i = strlen(s1);
    for (j = 0; s2[j] != '\0'; i++, j++) {
            s1[i] = s2[j];
    }
    s1[i] = '\0';
        }
```

```
Enter String 1 : Ankit
Enter String 2 : Singh
Concated string is : AnkitSingh
```

Length

The number of characters in a string is called its length.

Syntax: LENGTH (string)

Ex: LENGTH (‘computer’) = 8

String length is determined in C language using the ***strlen( )*** function, as shown below:

X = strlen ("sunrise");

strlen function returns an integer value 7 and assigns it to the variable X

Similar to ***strcat, strlen*** is also a part of string.h, hence the header file must be included at the time of pre-processing.

```
C program to find the length of a string without using the
 * built-in function
 */
#include <stdio.h>

void main()
{
   char string[50];
   int i, length = 0;

   printf("Enter a string \n");
   gets(string);
   /*  keep going through each character of the string till its end */
   for (i = 0; string[i] != '\0'; i++)
   {
      length++;
   }
   printf("The length of a string is the number of characters in it \n");
   printf("So, the length of %s = %d\n", string, length);
}
```

```
Enter a string
hello
The length of a string is the number of characters in it
So, the length of hello = 5
```

C strcmp()

The strcmp() function compares two strings and returns 0 if both strings are identical.

C strcmp() Prototype

```
int strcmp (const char* str1, const char* str2);
```

The strcmp() function takes two strings and return an integer.

The strcmp() compares two strings character by character. If the first character of two strings are equal, next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

It is defined in string.h header file.

Return Value from strcmp()

Return Value	Remarks
0	if both strings are identical (equal)
Negative	if the ASCII value of first unmatched character is less than second.
positive integer	if the ASCII value of first unmatched character is greater than second.

C program to compare two strings without using string functions

```
#include<stdio.h>

int stringCompare(char[],char[]);
int main(){
    char str1[100],str2[100];
    int compare;

    printf("Enter first string: ");
    scanf("%s",str1);

    printf("Enter second string: ");
    scanf("%s",str2);

    compare = stringCompare(str1,str2);

    if(compare == 1)
        printf("Both strings are equal.");
    else
        printf("Both strings are not equal");

    return 0;
}

int stringCompare(char str1[],char str2[]){
    int i=0,flag=0;

    while(str1[i]!='\0' && str2[i]!='\0'){
        if(str1[i]!=str2[i]){
            flag=1;
            break;
        }
        i++;
    }

    if (flag==0 && str1[i]=='\0' && str2[i]=='\0')
        return 1;
    else
        return 0;

}
```

Sample output:

```
Enter first string: HELLO
Enter second string: HELLO
Both strings are equal.
```

C strcpy()

The strcpy() function copies the string to the another character array.

strcpy() Function prototype

```
char* strcpy(char* destination, const char* source);
```

The strcpy() function copies the string pointed by `source` (including the null character) to the character array `destination`.

This function returns character array `destination`.

The strcpy() function is defined in `string.h` header file.

String copy without using strcpy in c programming language

```
#include<stdio.h>

void stringCopy(char[],char[]);

int main(){

    char str1[100],str2[100];

    printf("Enter any string: ");
    scanf("%s",str1);

    stringCopy(str1,str2);

    printf("After copying: %s",str2);

    return 0;
}

void stringCopy(char str1[],char str2[]){
   int i=0;

   while(str1[i]!='\0'){
       str2[i] = str1[i];
       i++;
   }

   str2[i]='\0';
}
```

```
Sample output:
Enter any string:HELLO
After copying: HELLO
```

PATTERN MATCHING ALGORITHMS

Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T. The length of P does not exceed the length of T.

First Pattern Matching Algorithm

- The first pattern matching algorithm is one in which comparison is done by a given pattern P with each of the substrings of T, moving from left to right, until a match is found.

$$\mathbf{W_K = SUBSTRING\ (T,\ K,\ LENGTH\ (P))}$$

- Where, WK denote the substring of T having the same length as P and beginning with the K^{th} character of T.
- First compare P, character by character, with the first substring, W1. If all the characters are the same, then P = W1 and so P appears in T and INDEX (T, P) = 1.
- Suppose it is found that some character of P is not the same as the corresponding character of W1. Then P ≠ W1

- Immediately move on to the next substring, W2 That is, compare P with W2. If P ≠ W2 then compare P with W3 and so on.

- The process stops, When P is matched with some substring WK and so P appears in T and INDEX(T,P) = K or When all the WK'S with no match and hence P does not appear in T.

- The maximum value MAX of the subscript K is equal to LENGTH(T) -LENGTH(P) +1.

Algorithm: (Pattern Matching)

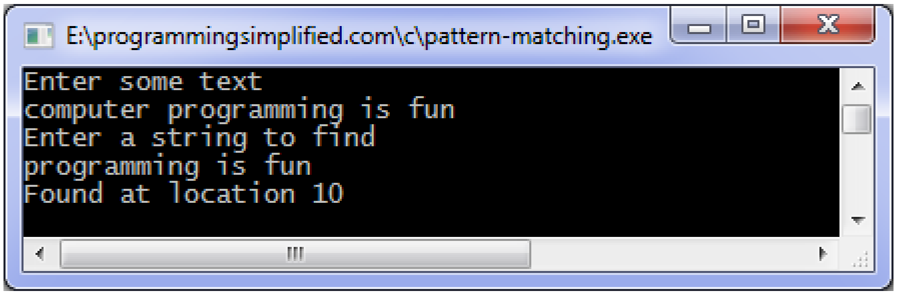
P and T are strings with lengths R and S, and are stored as arrays with one character per element. This algorithm finds the INDEX of P in T.

```
1. [Initialize.] Set K: = 1 and MAX: = S - R + 1

2. Repeat Steps 3 to 5 while K ≤ MAX

3.      Repeat for L = 1 to R: [Tests each character
             of P] If P[L] ≠ T[K + L – l], then: Go to
             Step 5

        [End of inner loop.]
4.      [Success.] Set INDEX = K, and Exit

5.      Set K := K + 1

   [End of Step 2 outer loop]

6. [Failure.] Set INDEX = O

7. Exit
```

PATTERN MATCHING PROGRAM

```
#include <stdio.h>
#include <string.h>

int match(char [], char []);
int main() {
  char a[100], b[100];
  int position;

  printf("Enter some text\n");
  gets(a);

  printf("Enter a string to find\n");
  gets(b);

  position = match(a, b);

  if (position != -1) {
    printf("Found at location: %d\n", position + 1);
  }
  else {
    printf("Not found.\n");
  }

  return 0;
}

int match(char text[], char pattern[]) {
  int c, d, e, text_length, pattern_length, position = -1;

  text_length    = strlen(text);
  pattern_length = strlen(pattern);

  if (pattern_length > text_length) {
    return -1;
  }

  for (c = 0; c <= text_length - pattern_length; c++) {
    position = e = c;

    for (d = 0; d < pattern_length; d++) {
      if (pattern[d] == text[e]) {
        e++;
      }
      else {
        break;
      }
    }
    if (d == pattern_length) {
      return position;
    }
  }

  return -1;
}
```

OUTPUT

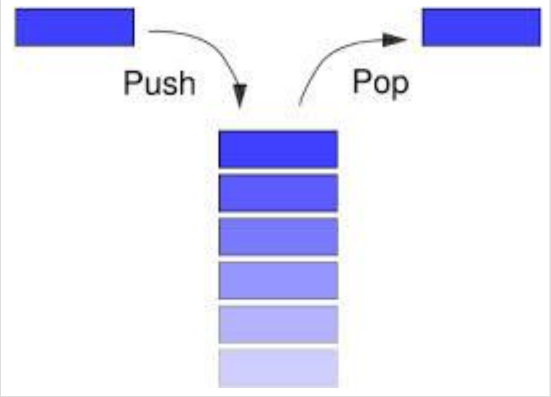


```
Enter some text
computer programming is fun
Enter a string to find
programming is fun
Found at location 10
```

STACKS AND QUEUES

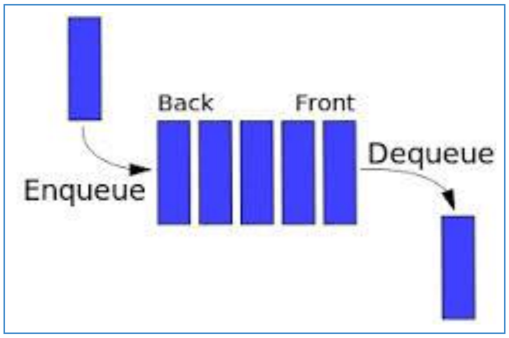
Stack:

In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



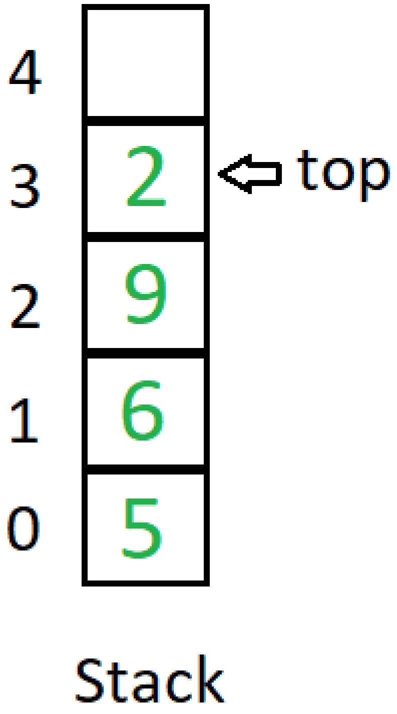
Queue:

An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access. 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.



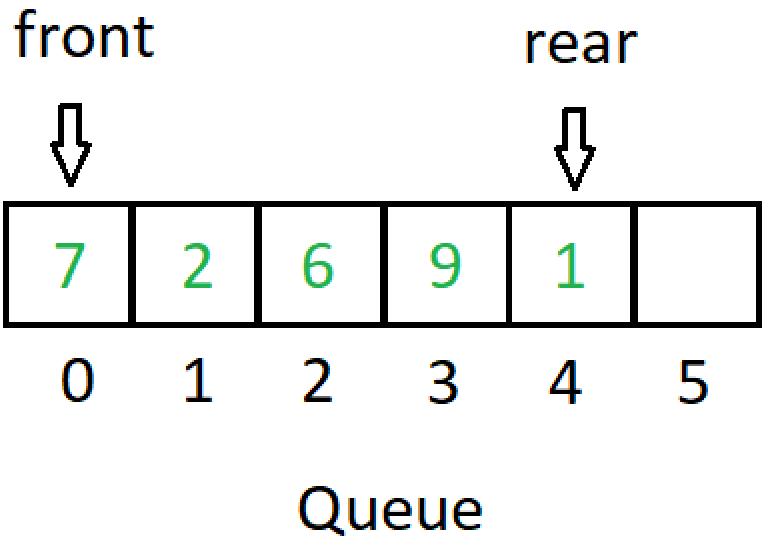
Difference between Stack and Queue Data Structures

Stack:- A stack is a linear data structure in which elements can be inserted and deleted only from one side of the list, called the **top**. A stack follows the **LIFO** (Last In First Out) principle, i.e., the element inserted at the last is the first element to come out. The insertion of an element into stack is called **push** operation, and deletion of an element from the stack is called **pop** operation. In stack we always keep track of the last element present in the list with a pointer called **top**.
The diagrammatic representation of stack is given below:



Queue:- A queue is a linear data structure in which elements can be inserted only from one side of the list called **rear**, and the elements can be deleted only from the other side called the **front**. The queue data structure follows the **FIFO** (First In First Out) principle, i.e. the element inserted at first in the list, is the first element to be removed from the list. The insertion of an element in a queue is called an **enqueue** operation and the deletion of an element is called a **dequeue** operation. In queue we always maintain two pointers, one pointing to the element which was inserted at the first and still present in the list with the **front** pointer and the second pointer pointing to the element inserted at the last with the **rear** pointer.

The diagrammatic representation of queue is given below:



Difference between Stack and Queue Data Structures

STACKS	QUEUES
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.

STACKS	QUEUES
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
begin
    if top = n then stack full
    top = top + 1
    stack (top) : = item;
end
```

Time Complexity : o(1)

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
    if top = 0 then stack empty;
    item := stack(top);
    top = top - 1;
end;
```

Time Complexity : o(1)

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

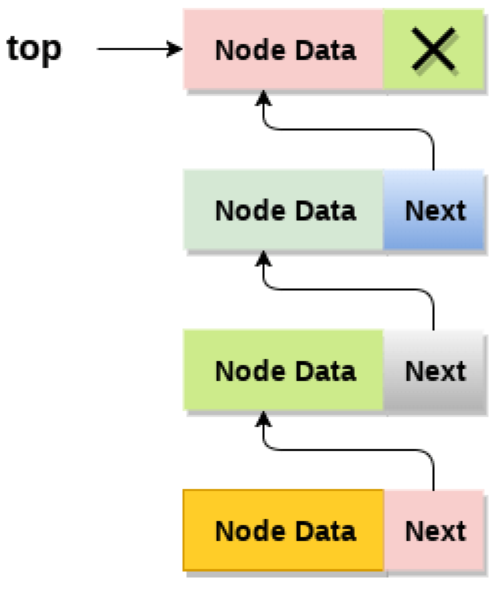
```
Begin
    if top = -1 then stack empty
    item = stack[top]
    return item
End
```

Time complexity: o(n)

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

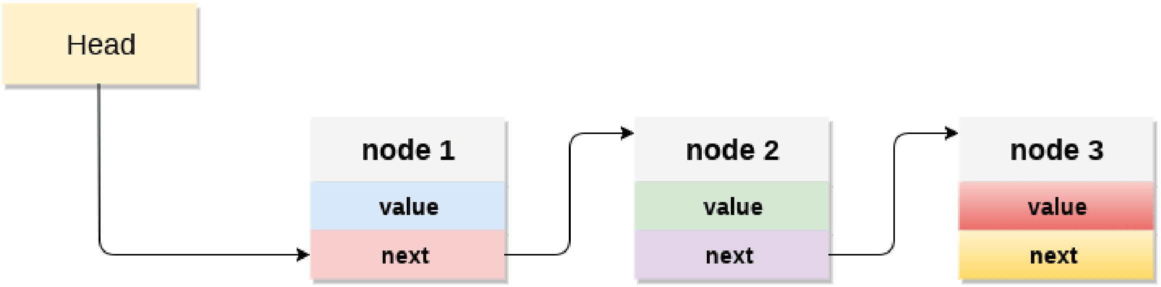
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

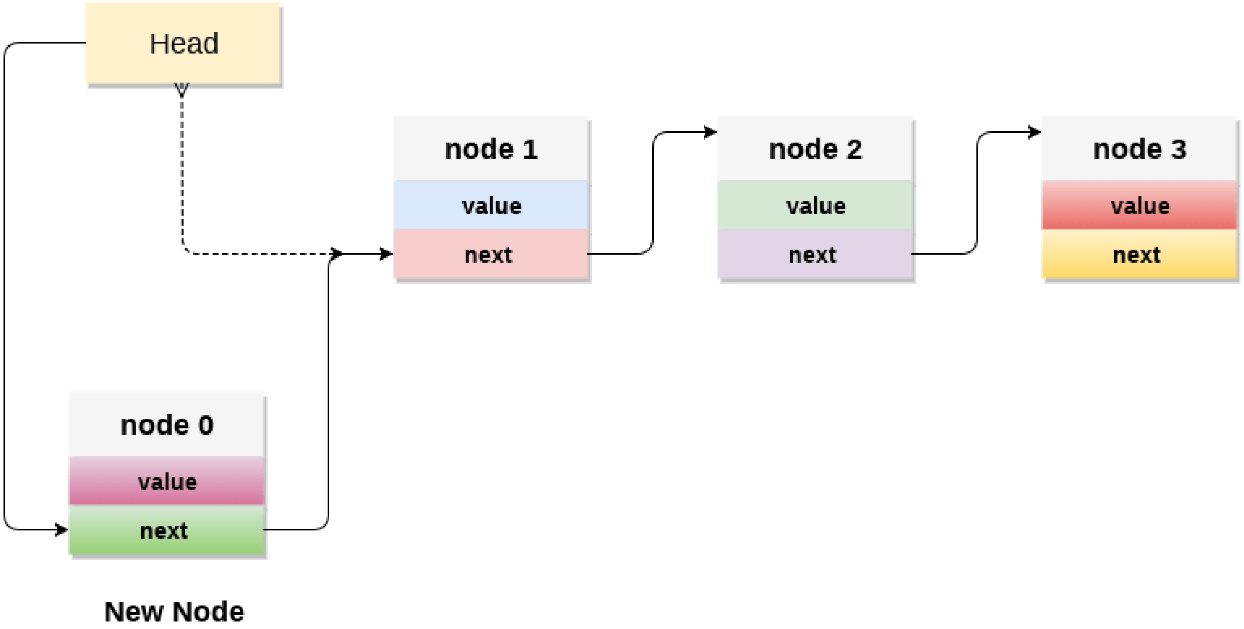
Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

   Time Complexity : o(1)





Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

Display the nodes (Traversing)

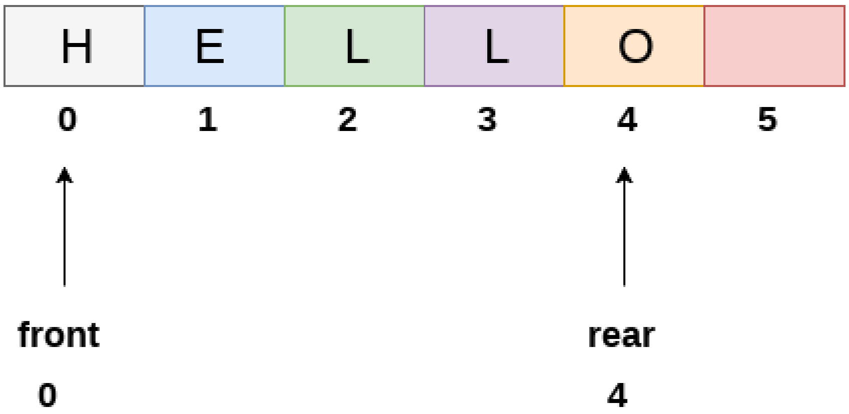
Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

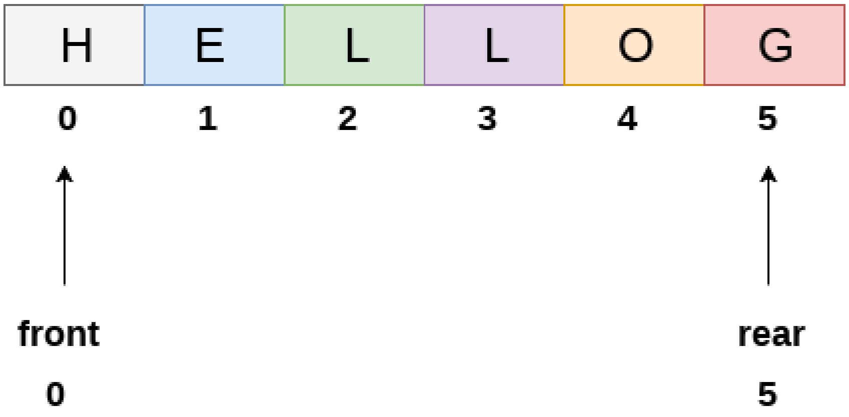
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



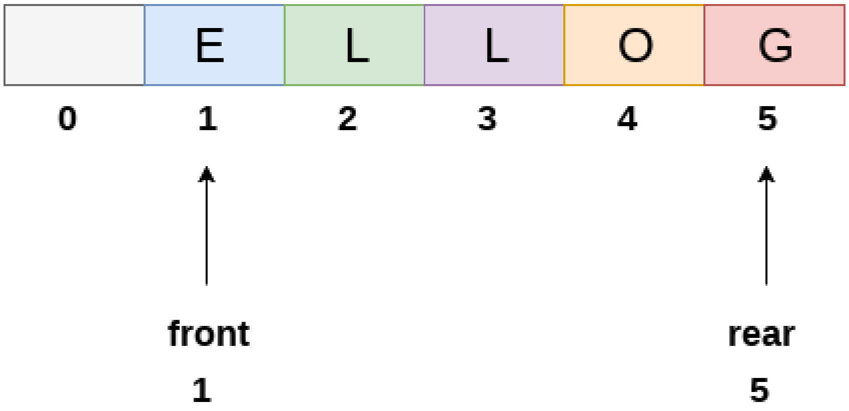
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
Step 1: IF REAR = MAX - 1
    Write OVERFLOW
    Go to step
    [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
    SET FRONT = REAR = 0
    ELSE
    SET REAR = REAR + 1
    [END OF IF]
Step 3: Set QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

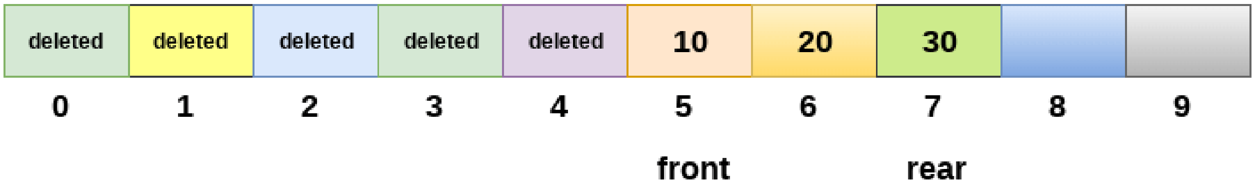
Algorithm

```
Step 1: IF FRONT = -1 or FRONT > REAR
    Write UNDERFLOW
    ELSE
    SET VAL = QUEUE[FRONT]
    SET FRONT = FRONT + 1
    [END OF IF]
Step 2: EXIT
```

Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



- **limitation of array representation of queue**

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

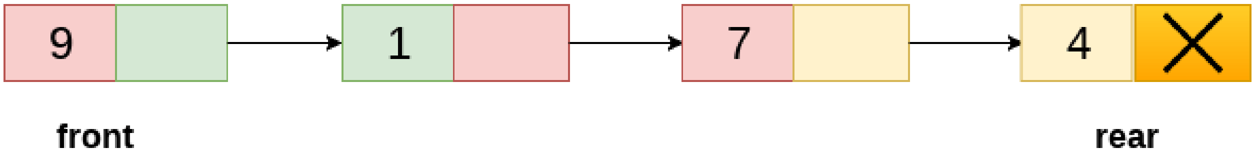
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

```
Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
            rear = ptr;
```

```
    rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

```
Step 1: Allocate the space for the new node PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
        SET FRONT = REAR = PTR
        SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
        SET REAR -> NEXT = PTR
        SET REAR = PTR
        SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
    front = front -> next;
    free(ptr);
```

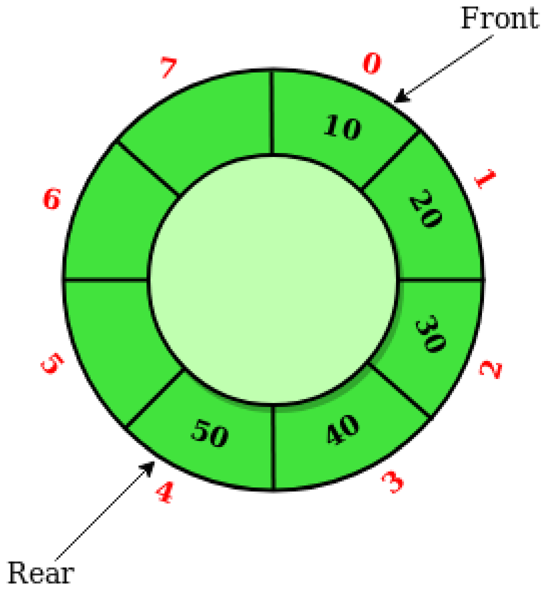
The algorithm and C function is given as follows.

Algorithm

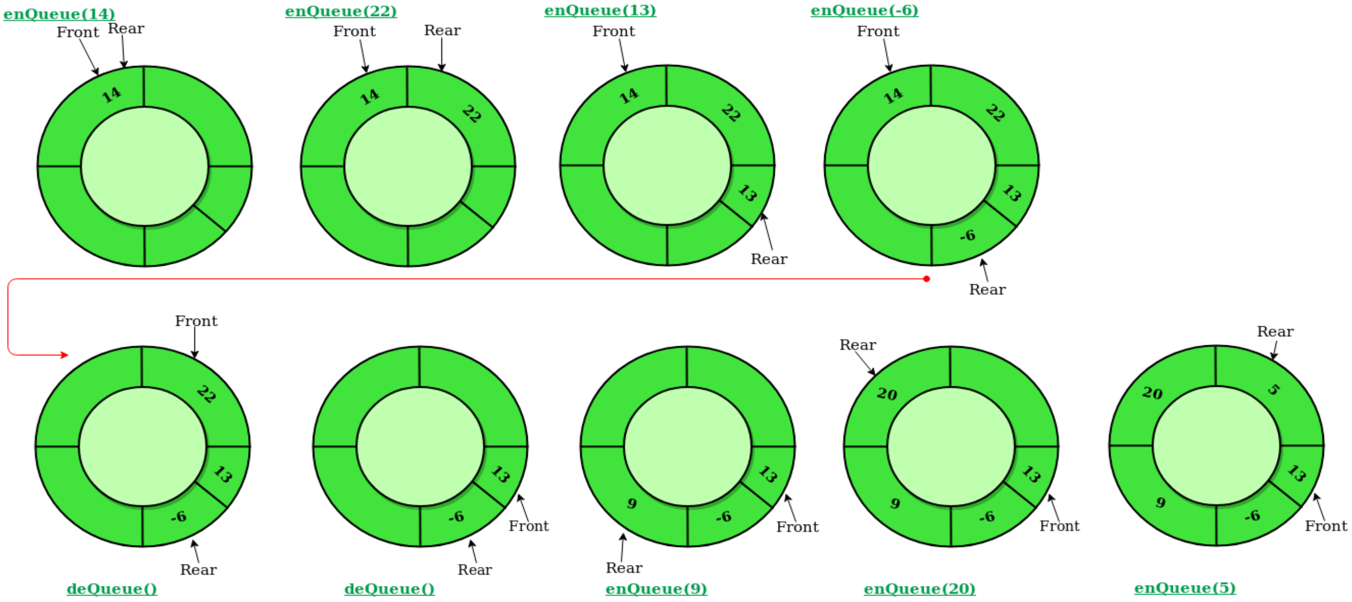
```
Step 1: IF FRONT = NULL
        Write " Underflow "
        Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

CIRCULAR QUEUES

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.

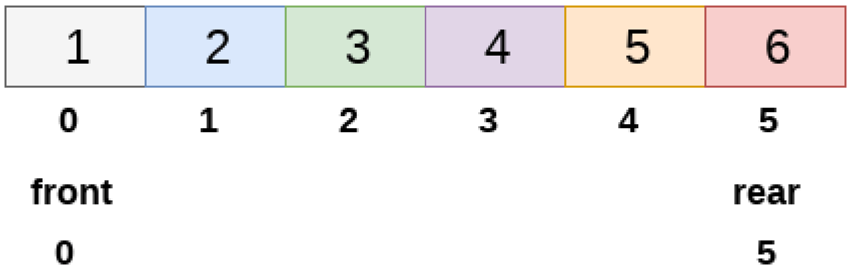


In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



- ➢ Deletions and insertions can only be performed at front and rear end respectively, as far as linear queue is concerned.

Consider the queue shown in the following figure.

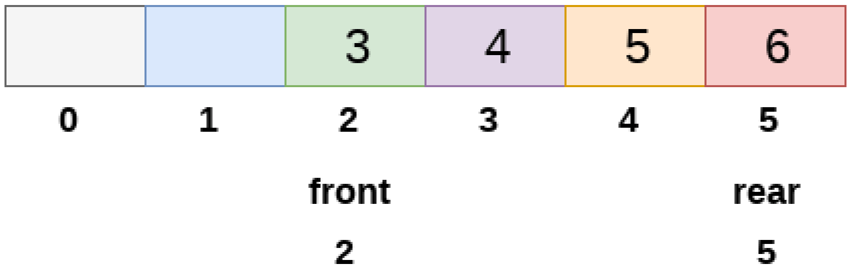


Queue

The Queue shown in above figure is completely filled and there can't be inserted any more element due to the condition **rear == max - 1 becomes true**.

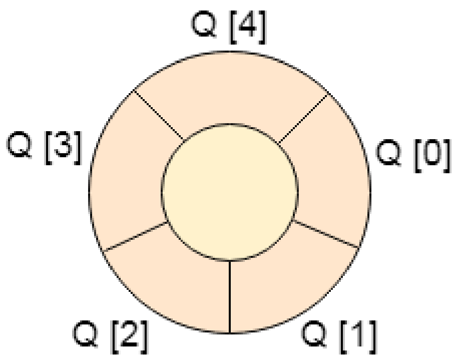
However, if we delete 2 elements at the front end of the queue, we still can not insert any element since the condition **rear = max -1 still holds**.

This is the main problem with the linear queue, although we have space available in the array, but we can not insert any more element in the queue. This is simply the memory wastage and we need to overcome this problem.



Queue after deletion of first 2 elements

One of the solution of this problem is circular queue. In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure.



Circular queue will be full when **front = -1** and **rear = max-1**. Implementation of circular queue is similar to that of a linear queue. Only the logic part that is implemented in the case of insertion and deletion is different from that in a linear queue.

Complexity

Time Complexity

Front	O(1)
Rear	O(1)
enQueue()	O(1)
deQueue()	O(1)

Insertion in Circular queue

There are three scenario of inserting an element in a queue.

1. **If (rear + 1)%maxsize = front**, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
2. **If rear != max - 1**, then rear will be incremented to the **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
3. **If front != 0 and rear = max - 1**, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

Algorithm to insert an element in circular queue

```
○ Step 1: IF (REAR+1)%MAX = FRONT
      Write " OVERFLOW "
      Goto step 4
      [End OF IF]
○ Step 2: IF FRONT = -1 and REAR = -1
      SET FRONT = REAR = 0
      ELSE IF REAR = MAX - 1 and FRONT ! = 0
      SET REAR = 0
      ELSE
      SET REAR = (REAR + 1) % MAX
      [END OF IF]
○ Step 3: SET QUEUE[REAR] = VAL
○ Step 4: EXIT
```

Algorithm to delete an element from a circular queue

To delete an element from the circular queue, we must check for the three following conditions.

1. If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.
2. If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.
3. If front = max -1 then, the value is deleted from the front end the value of front is set to 0.
4. Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

Algorithm

```
○ Step 1: IF FRONT = -1
  Write " UNDERFLOW "
  Goto Step 4
  [END of IF]
○ Step 2: SET VAL = QUEUE[FRONT]
○ Step 3: IF FRONT = REAR
  SET FRONT = REAR = -1
  ELSE
  IF FRONT = MAX -1
  SET FRONT = 0
  ELSE
  SET FRONT = FRONT + 1
  [END of IF]
  [END OF IF]
○ Step 4: EXIT
```

Priority Queue

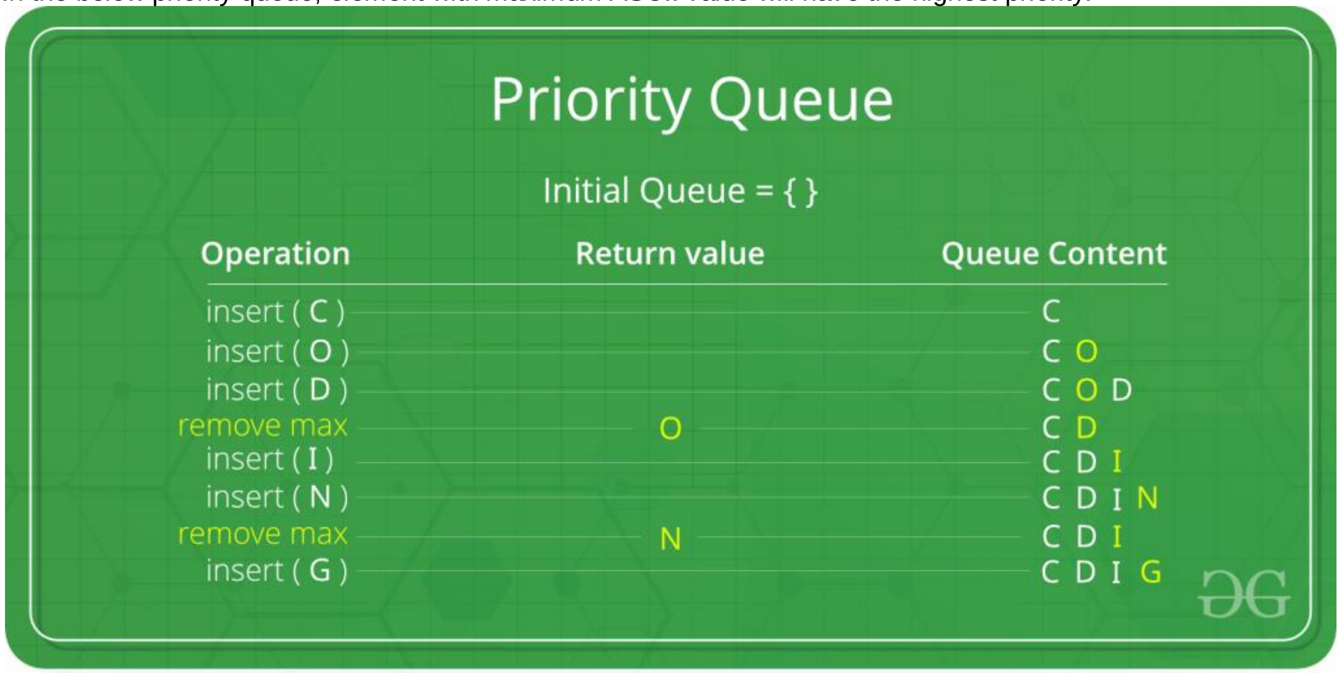
a **priority queue** is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. In some implementations, if two elements have the same priority, they are served according to the order in which they were enqueued, while in other implementations, ordering of elements with the same priority is undefined.

OR

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

In the below priority queue, element with maximum ASCII value will have the highest priority.



Priority Queue

Initial Queue = { }

Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G

A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.

How to implement priority queue?
Using Array: A simple implementation is to use array of following structure.

```
struct item {
   int item;
   int priority;
}
```

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.

deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don’t have to move items.

UNIT – II

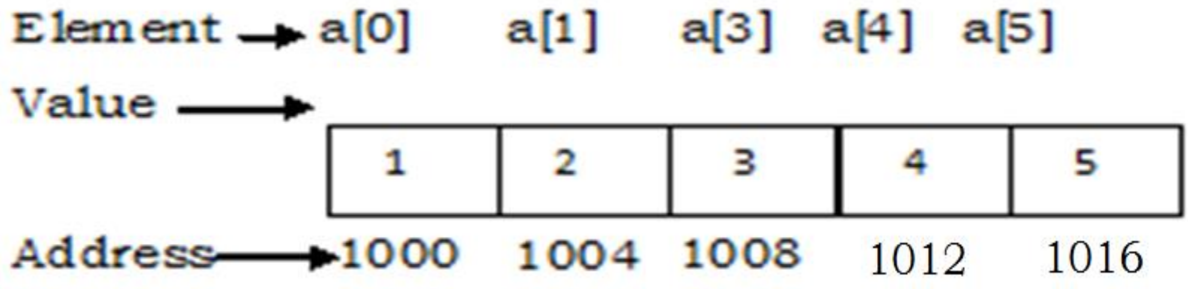
POINTERS

POINTER ARRAYS

- ✓ When an array is declared, the compiler allocates a base address and sufficient amount of memory to contain all the elements of the array in continuous memory locations.
- ✓ The base address is the location of the first element of the array denoted by **a[0]**.
- ✓ The compiler also defines the array name as constant pointer to the first element.
- ✓ For Example: -

int a [5] = {1, 2, 3, 4, 5};

- ✓ Here if the base address is 1000 for "a" and integer occupies 4 bytes then the five elements requires 20 bytes as shown below.



Element →	a[0]	a[1]	a[3]	a[4]	a[5]
Value →	1	2	3	4	5
Address →	1000	1004	1008	1012	1016

- ✓ The name of the array is "a" and it is defined as a constant pointer pointing to the first element of the array and it is a[0] whose base address is 1000 becomes the value of "a". It is represented as

a = &a[0] = 1000;

- ✓ If p is a pointer of integer type then p to point the array a is given by the assignment statement

p = a;

which is equivalent to

p = &a[0];

- ✓ Now it is possible to access every value of a using p++ to move from one element to another element as

P	**= &a[0]**	**=1000**
P+1	**= &a[1]**	**= 1004**
P+2	**= &a[2]**	**= 1008**
P+3	**= &a[3]**	**= 1012**
P+4	**= &a[4]**	**= 1016**

- ✓ The address of the element is calculated by using the formula

address of a[3] = base address + (3 * scale factor of int)

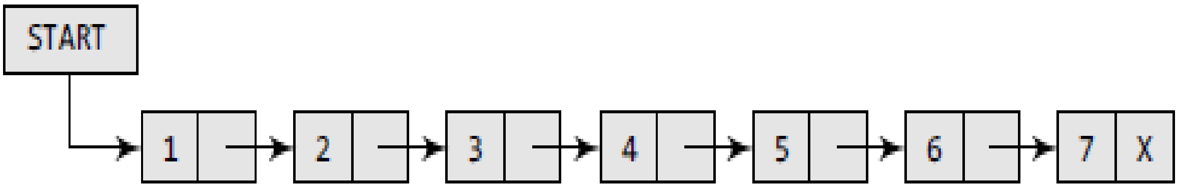
- ✓ When handling arrays we can use pointers to access the array elements. Hence *(p+3) gives the value of a[3].
- ✓ Pointers can also be used to manipulate two dimensional arrays. In one dimensional array “a” the expression.

***(a+i) or *(p+i)**

LINKED LIST

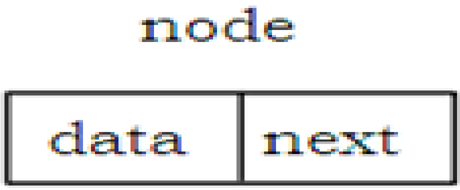
- ✓ It is a collection of linear list of data elements.
- ✓ The data elements are called **nodes**.
- ✓ Each node contains **two** parts: **data** and **link**.
- ✓ The data represents **integers** and link is a **pointer** that points to next node.
- ✓ The last node of the linked list is not connected to any node so it stores the value **NULL** in link part.
- ✓ Here NULL is defined as **-1**
- ✓ NULL pointer denotes **end** of the list.
- ✓ It contains pointer variable called start node that contains the address of first node in the list
- ✓ We can traverse the list starting from start node that contains first node address and in turn first node contains second node address and so on thus forming chain of nodes.
- ✓ If **start == NULL** then the list is empty.

- ✓ The diagrammatic representation of linked list is shown below:



Simple linked list

NODE REPRESENTATION



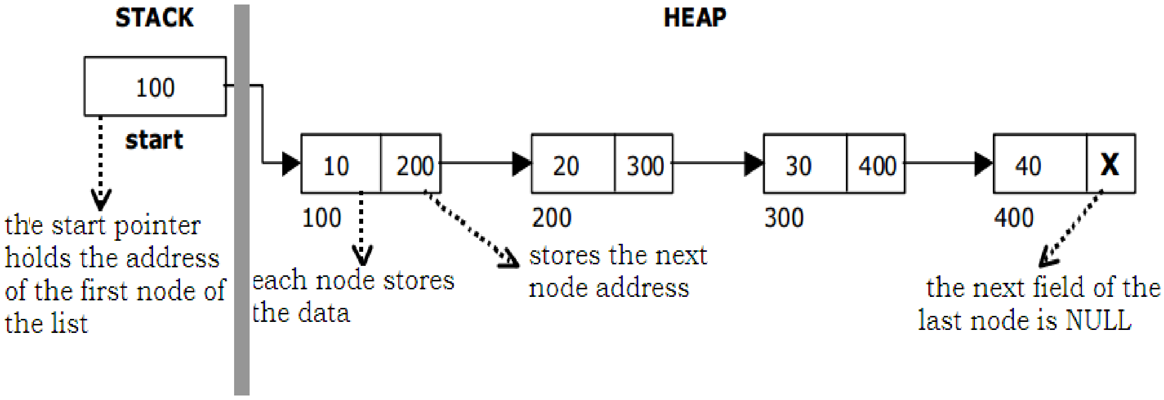
- ✓ In C language, the code for the linked list is

```
struct node
{
        int data;
        struct node *next;
 }
```

SINGLE LINKED LIST

- ✓ **“A single linked list is a linked list in which each node contains only one link pointing to the next node in the list”.**
- ✓ A linked list allocates space for each element separately in its own block of memory called a "**node**".
- ✓ The list gets an overall structure by using **pointers** to connect all its nodes together.
- ✓ Each node contains **two** fields - a "**data**" field to store element, and a "**next**" field which is a pointer used to connect to the next node.
- ✓ Each node is allocated in the **heap** using **malloc()** and it is explicitly de-allocated using **free()**.
- ✓ The single linked list starts with a pointer to the **“start”** node.
- ✓ The single linked list is called as **linear list** or **chain**.

- ✓ The **traversing** of data can be in **one** direction only.


STACK
HEAP
100
start
the start pointer holds the address of the first node of the list
10
200
100
each node stores the data
20
300
200
stores the next node address
30
400
300
40
X
400
the next field of the last node is NULL

Single Linked List Representation

- ✓ The beginning of the linked list is stored in a "**start**" pointer which points to the first node.
- ✓ The first node contains a pointer to the second node. The second node contains a pointer to the third node and so on.
- ✓ The last node in the list has its next field set to **NULL** to mark the end of the list.

ADT FOR SINGLE LINKED LIST

AbstractDataType SlinkedList

{

instances:

finite collection of zero or more elements linked by pointers

operations:

Count( ): Count the number of elements in the list.

Addatbeg(x): Add x to the beginning of the list.

Addatend(x): Add x at the end of the list.

Insert(k, x): Insert x just after kth element.

Delete(k): Delete the kth element.

Search(x): Return the position of x in the list otherwise return -1 if not found

Traverse( ): Display all elements of the list

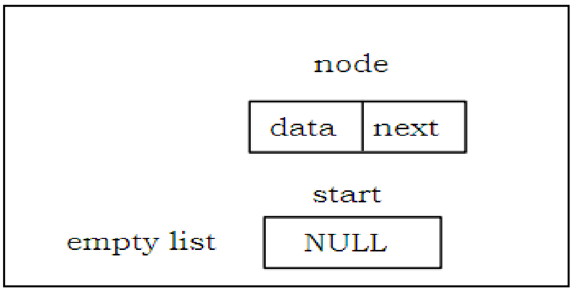
}

IMPLEMENTATION OF SINGLE LINKED LIST

Before writing the code to build the list, we need to create a **start** node**,** used to create and access other nodes in the linked list.

- ✓ Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as **self-referential structure.**
- ✓ Initialize the start pointer to be **NULL.**

```
struct slinklist
{
        int data;
        struct slinklist* next;
};
typedef struct slinklist node;
node *start = NULL;
```



BASIC OPERATION PERFORMED ON SINGLE LINKED LIST

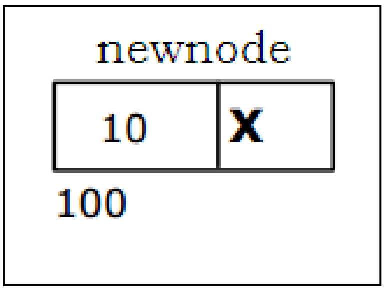
The different operations performed on the single linked list are listed as follows.

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching

Creating a node for Single Linked List

- ✓ Creating a singly linked list starts with creating a node.
- ✓ Sufficient memory has to be allocated for creating a node.
- ✓ The information is stored in the memory, allocated by using the **malloc()** function.
- ✓ The function **getnode()**, is used for creating a node, after allocating memory for the node, the information for the node data part has to be read from the user and set next field to **NULL** and finally return the node.

```
node* getnode()
{
        node* newnode;
        newnode = new node;
        printf(“Enter data”);
        scanf(“%d”, &newnode  -> data;
        newnode -> next = NULL;
        return newnode;
}
```



Creating a Singly Linked List with ‘n’ number of nodes

The following steps are to be followed to create ‘n’ number of nodes.

1. Get the new node using getnode().

    newnode = getnode();

2. If the list is empty, assign new node as start.

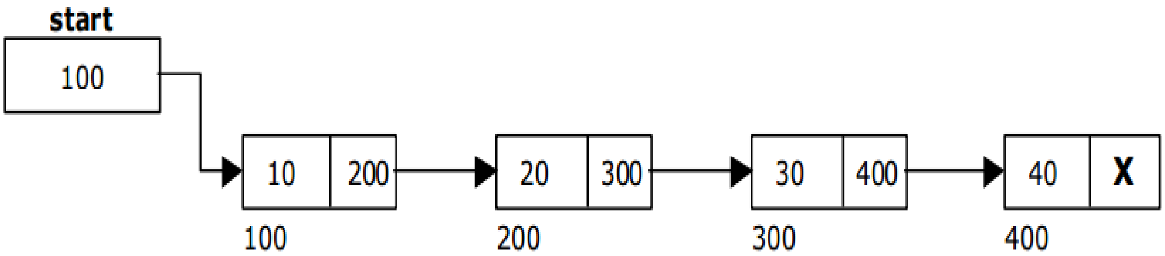
    start = newnode;

3. If the list is not empty, follow the steps given below.

    ✓ The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.

    ✓ The start pointer is made to point the new node by assigning the address of the new node.

4. Repeat the above steps ‘n’ times.



Single Linked List with 4 nodes

The function createlist(), is used to create ‘n’ number of nodes

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n ; i++)
        {
                newnode = getnode();
                if(start = = NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                                temp -> next = newnode;
                        }
        }
}
```

INSERTION OF A NODE

- ✓ One of the most important operations that can be done in a singly linked list is the insertion of a node.
- ✓ Memory is to be allocated for the newnode before reading the data.
- ✓ The newnode will contain empty data field and empty next field.
- ✓ The data field of the newnode is then stored with the information read from the user.
- ✓ The next field of the newnode is assigned to NULL.
- ✓ The newnode can then be inserted at three different places namely:
    - ✓ **Inserting a node at the beginning.**
    - ✓ **Inserting a node at the end.**
    - ✓ **Inserting a node at specified position.**

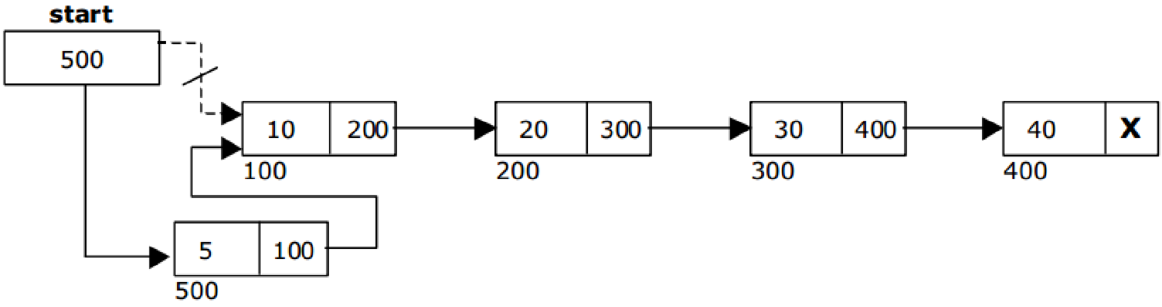
INSERTING A NODE AT THE BEGINNING

The following steps are to be followed to insert a newnode at the beginning of the list:

1. Get the newnode using getnode() then newnode = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty, follow the steps given below:

newnode -> next = start;

start = newnode;



Inserting a node at the begining of the list

The function insert_at_beg(), is used for inserting a node at the beginning.

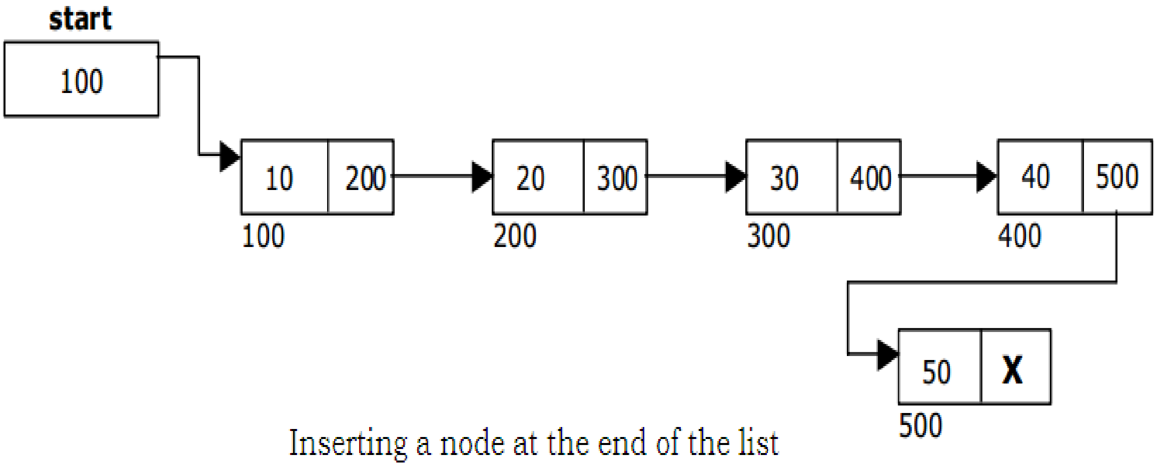
```
void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode  -> next = start;
                start = newnode;
        }
}
```

INSERTING A NODE AT THE END

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode() then newnode = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```



Inserting a node at the end of the list

The function insert_at_end(), is used for inserting a node at the end.

```
void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
```

```
                        temp = temp -> next;
                temp -> next = newnode;
        }
}
```

INSERTING A NODE AT SPECIFIED POSITION

The following steps are followed, to insert a new node in an intermediate position in the list:

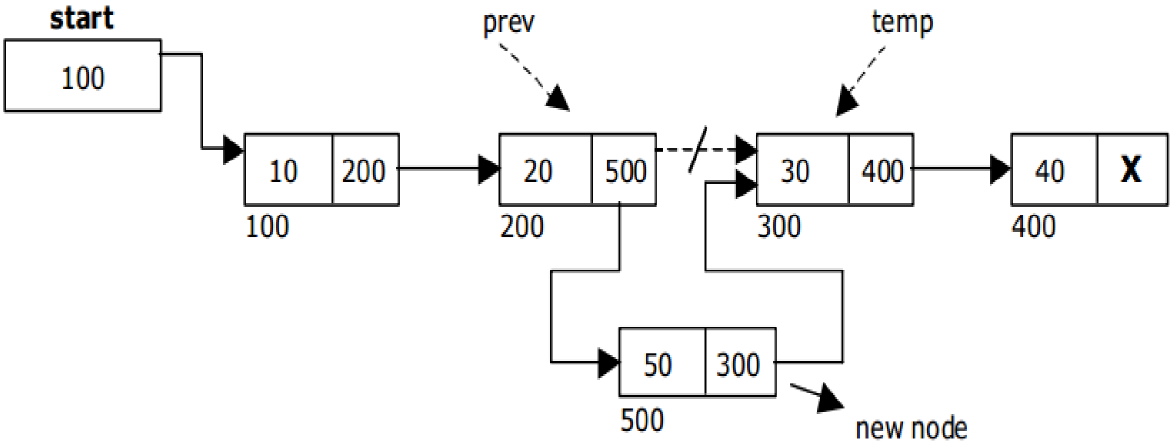
1. Get the new node using getnode() then newnode = getnode();

2. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

3. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

4. After reaching the specified position, follow the steps given below:

```
        prev -> next = newnode;
        newnode  -> next = temp;
```



Inserting a node at specified position

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        printf(" Enter the position");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
        {
                printf("%d", pos);
        }
}
```

DELETION OF A NODE

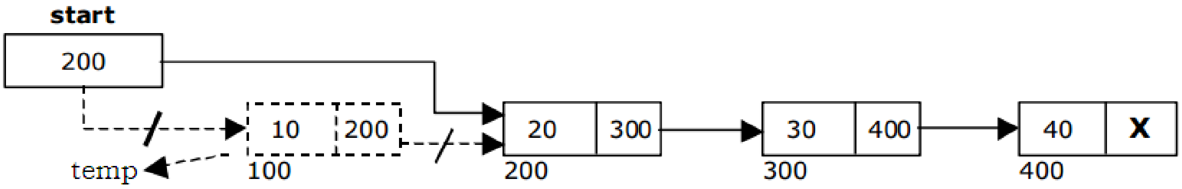
- ✓ Another operation that can be done in a singly linked list is the deletion of a node.
- ✓ Memory is to be released for the node to be deleted.
- ✓ It is done by using **free()** function.
- ✓ A node can be deleted from the list from three different places.
    - ✓ **Deleting a node at the beginning.**
    - ✓ **Deleting a node at the end.**
    - ✓ **Deleting a node at specified position.**

DELETING A NODE AT THE BEGINNING

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
free(temp);
```



deleting a node at the begining

The function delete_at_beg(), is used for deleting the first node in the list.

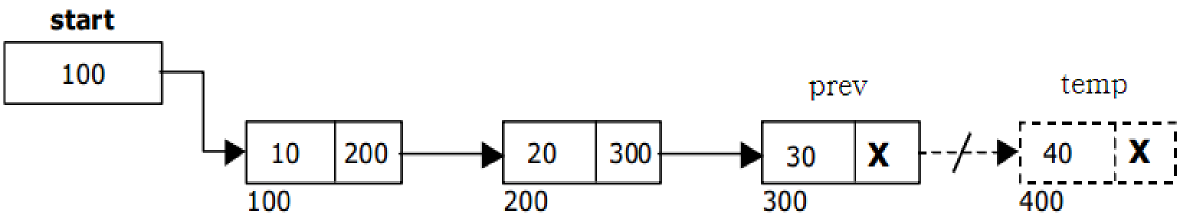
```
void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf(" Empty List ");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("Node deleted");
        }
}
```

DELETING A NODE AT THE END

The following steps are followed to delete a node at the end of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = prev = start;
while(temp -> next != NULL)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = NULL;
free(temp);
```



deleting a node at the end

The function delete_at_last(), is used for deleting the last node in the list.

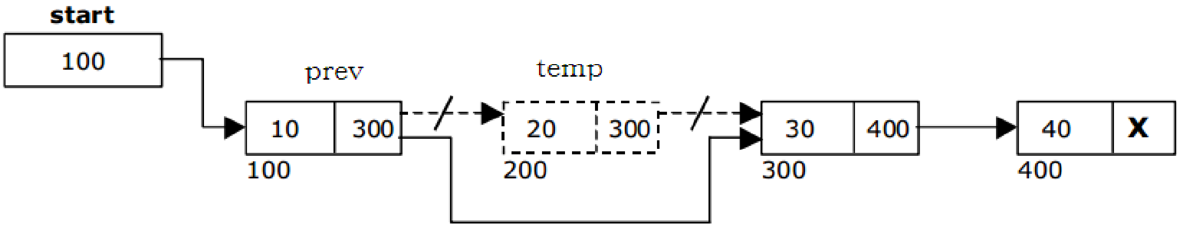
```
void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf(" Empty List ");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("Node deleted");
        }
}
```

DELETING A NODE AT SPECIFIED POSITION

The following steps are followed, to delete a node from the specified position in the list.

1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps given below.

```
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                ctr = 1;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = temp -> next;
                free(temp);
                printf("Node deleted");
        }
```



deleting a node at the specified position

The function delete_at_mid(), is used for deleting the specified position node in the list.

```
void delete_at_mid()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf(" Empty List ");
                return ;
        }
        else
        {
                printf(" Enter position of node to delete ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf(" This node doesnot exist: ");
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                        ctr ++;
                    }
                    prev -> next = temp -> next;
                    free(temp);
                    printf("Node deleted");
                }
                else
                    printf("Invalid position");
            }
        }
```

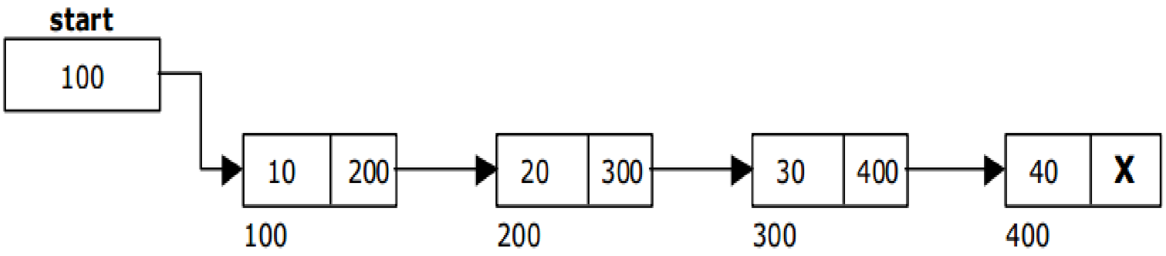
TRAVERSAL AND DISPLAYING A LIST (LEFT TO RIGHT)

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps.

1. Assign the address of start pointer to a temp pointer.
2. Display the information from the data field of each node.

The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
    node *temp;
    temp = start;
    printf(" The contents of List (Left to Right) ");
    if(start == NULL )
        printf(" Empty List ");
    else
    {
        while (temp != NULL)
        {
            printf("%d", temp -> data);
            temp = temp -> next;
        }
    }
    printf("%d", X);
}
```

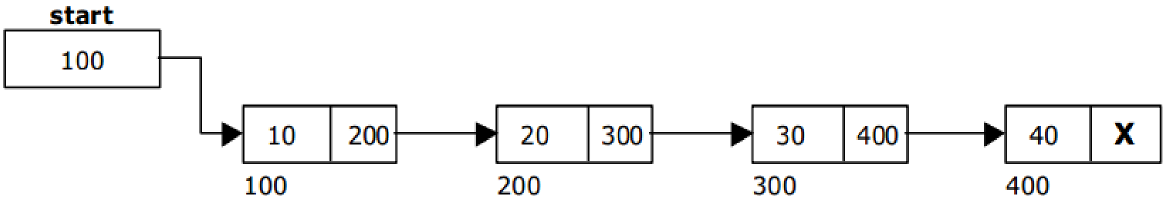


Single Linked List with 4 nodes

SEARCHING A NODE IN A SINGLE LINKED LIST

- ✓ Searching a single linked list means to find a particular element in the single linked list.
- ✓ A single linked list consists of nodes which are divided into two parts, the data part and the next part.
- ✓ So searching means finding whether a given value is present in the data part of the node or not.
- ✓ If it is present, then display element found otherwise element not found.

```
void search()
{
    node *temp;
    int value = 30;
    temp = start;
    if(start == NULL )
        printf(" Empty List ");
    else
    {
        while (temp != NULL)
        {
            if(value = temp->data)
            {
                printf(" Element found ");
                return;
            }
            temp = temp -> next;
        }
        printf(" Element not found ");
    }
}
```



Single Linked List with 4 nodes

ADVANTAGES OF SINGLE LINKED LIST

- ✓ Insertions and Deletions can be done easily.
- ✓ It does not need movement of elements for insertion and deletion.
- ✓ The space is not wasted as we can get space according to our requirements.
- ✓ Its size is not fixed.
- ✓ It can be extended or reduced according to requirements.
- ✓ Elements may or may not be stored in consecutive memory available, even then we can store the data in computer.
- ✓ It is less expensive.

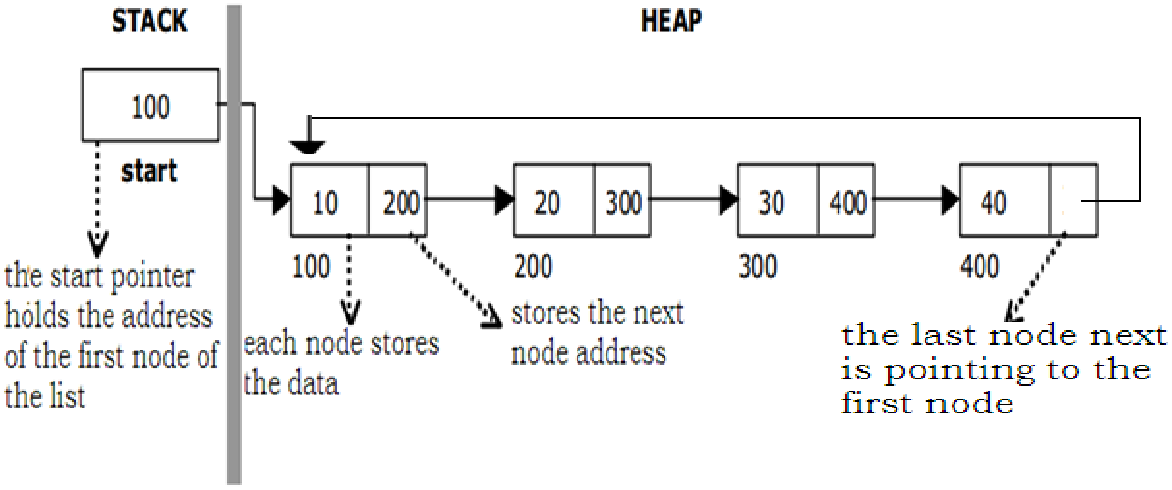
DISADVANTAGES OF SINGLE LINKED LIST

- ✓ It requires more space as pointers are also stored with information.
- ✓ Different amount of time is required to access each element.
- ✓ If we have to go to a particular element then we have to go through all those elements that come before that element.

- ✓ We cannot traverse it from last.
- ✓ It is not easy to sort the elements stored in the Single linked list.

CIRCULAR LINKED LISTS

- ✓ Circular linked list is a linked list which consists of collection of nodes each of which has two parts, namely the data part and the next part.
- ✓ The data part contains the value of the node and the next part has the address of the next node.
- ✓ The last node of list has the next pointer pointing to the first node thus making circular traversal possible in the list. A circular linked list has no beginning and no end.
- ✓ In circular linked list no null pointers are used, hence all pointers contain valid address.


STACK
HEAP
100
start
the start pointer holds the address of the first node of the list
10
200
20
300
30
400
40
100
200
300
400
each node stores the data
stores the next node address
the last node next is pointing to the first node

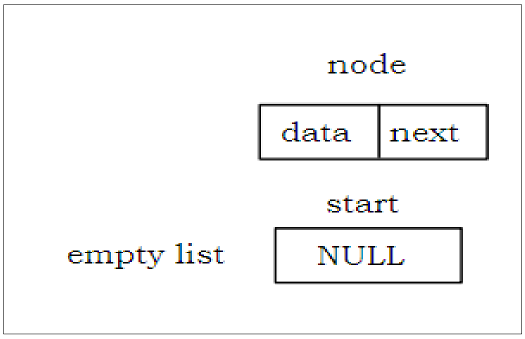
Circular Linked List Reresentation

IMPLEMENTATION OF CIRCULAR LINKED LIST

Before writing the code to build the list, we need to create a **start** node**,** used to create and access other nodes in the linked list.

- ✓ Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as **self-referential structure.**
- ✓ Initialize the start pointer to be **NULL.**

```
struct clinklist
{
        int data;
        struct clinklist* next;
};
typedef struct clinklist node;
node *start = NULL;
```



BASIC OPERATION PERFORMED ON CIRCULAR LINKED LIST

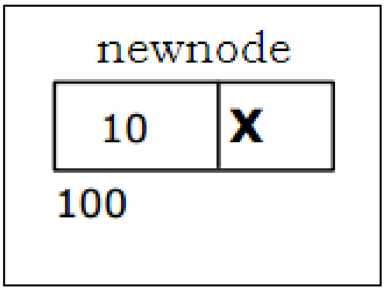
The operations on the circular linked list are listed as follows.

1. Creation
1. Insertion
2. Deletion
3. Traversing
4. Display

CREATING A NODE FOR CIRCULAR LINKED LIST

- ✓ Creating a circular linked list starts with creating a node. Sufficient memory has to be allocated for creating a node.
- ✓ The information is stored in the memory, allocated by using the **malloc()** function.
- ✓ The function **getnode(),** is used for **creating a node**, after allocating memory for the node, the information for the node **data part** has to be read from the user and set **next** field to **NULL** and finally return the **node.**

```
node* getnode()
{
        node* newnode;
        newnode = new node;
        printf(" Enter data ");
        scanf("%d", &newnode  -> data);
        newnode -> next = NULL;
```



```
        return newnode;
    }
```

Creating a Circular Linked List with 'n' number of nodes

The following steps are to be followed to create 'n' number of nodes.

1. Get the new node using getnode().

newnode = getnode();

2. If the list is empty, assign new node as start.

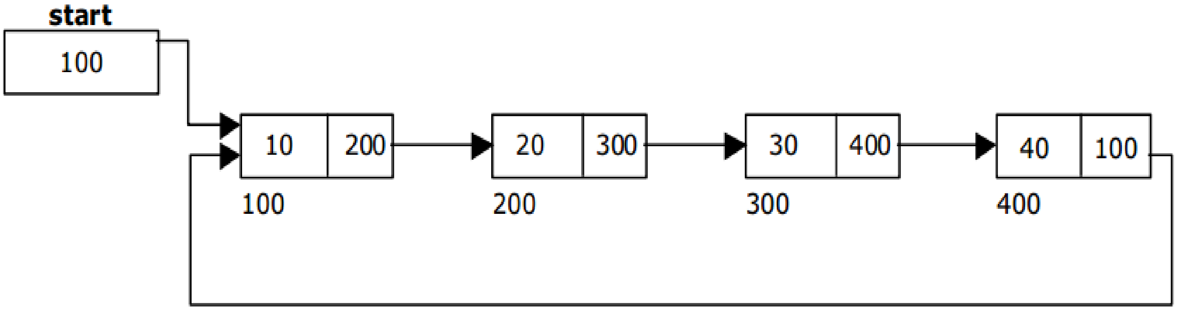
```
    start = newnode;
```

3. If the list is not empty, follow the steps given below.

```
    temp = start;
    while(temp -> next != NULL)
        temp = temp -> next;
    temp -> next = newnode;
```

4. Repeat the above steps 'n' times.

5. **newnode -> next = start;**



Circular Linked List with 4 nodes

The function createlist(), is used to create 'n' number of nodes

```
void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n ; i++)
    {
            newnode = getnode();
            if(start = = NULL)
            {
                    start = newnode;
            }
            else
            {
                    temp = start;
                    while(temp -> next != NULL)
                            temp = temp -> next;
                            temp -> next = newnode;
            }
            newnode -> next = start;
        }
}
```

INSERTING A NODE

- ✓ One operation performed on circular linked list is the insertion of a node.
- ✓ Memory is to be allocated for the newnode before reading the data.
- ✓ The newnode will contain empty data field and empty next field. The data field of the newnode is then stored with the information read from the user. The next field of the newnode is assigned to NULL.
- ✓ The newnode can then be inserted at three different positions:
    - ✓ **Inserting a node at the beginning.**
    - ✓ **Inserting a node at the end.**

INSERTING A NODE AT THE BEGINNING

The following steps are to be followed to insert a new node at the beginning of the circular list:

1. Get the new node using getnode().

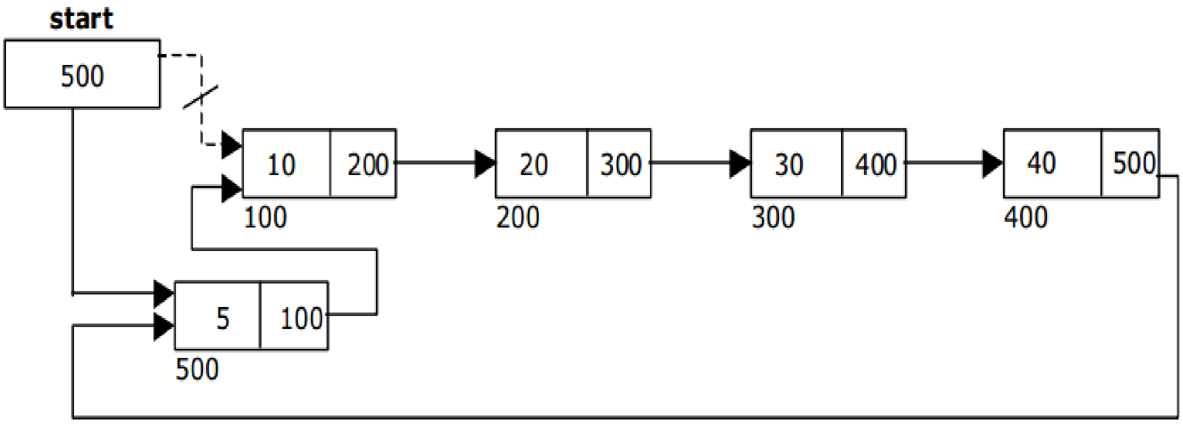
    newnode = getnode();

2. If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

3. If the list is not empty, follow the steps given below:

```
last = start;
while(last -> next != start)
    last = last -> next;
newnode -> next = start;
start = newnode;
last -> next = start;
```



Inserting a node at the beginning

INSERTING A NODE AT THE END

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode().

```
newnode = getnode();
```

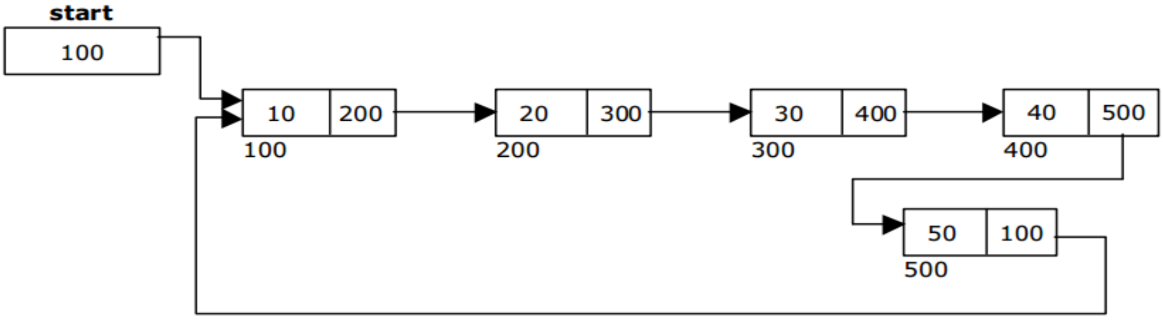
2. If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

3. If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != start)
```

```
        temp = temp -> next;
temp -> next = newnode;
newnode -> next = start;
```



Inserting a node at the end

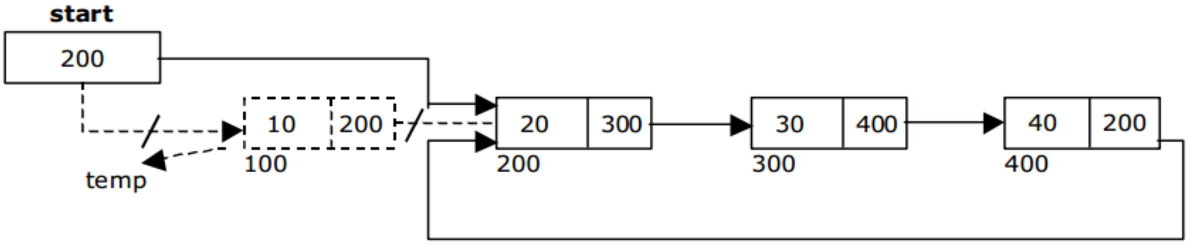
DELETING A NODE AT THE BEGINNING

The following steps are followed, to delete a node at the beginning of the list:

1. If the list is empty, display a message 'Empty List'.
2. If the list is not empty, follow the steps given below:

```
last = temp = start;
while(last -> next != start)
        last = last -> next;
start = start -> next;
last -> next = start;
```

3. After deleting the node, if the list is empty then **start = NULL.**



Deleting a node at beginning

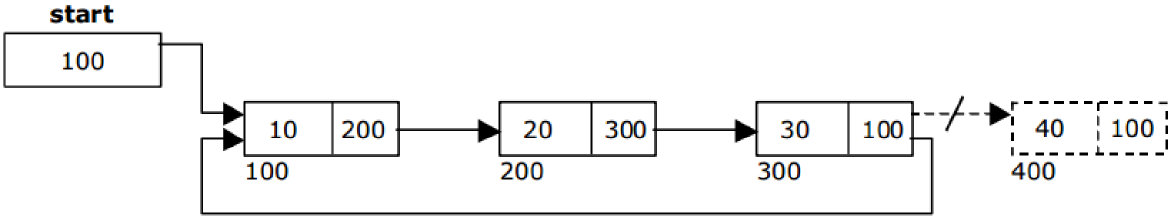
DELETING A NODE AT THE END

The following steps are followed to delete a node at the end of the list:

1. If the list is empty, display a message 'Empty List'.
2. If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = start;
```

4. After deleting the node, if the list is empty then **start = NULL**



Deleting a node at the end.

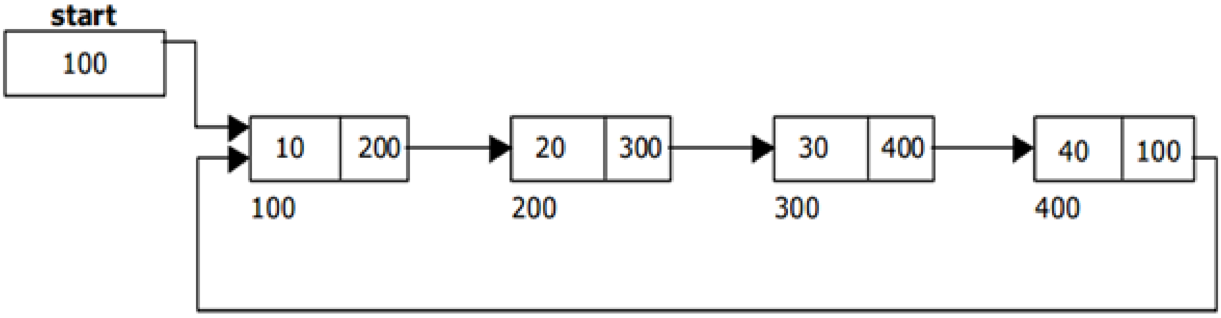
TRAVERSING A CIRCULAR LINKED LIST FROM LEFT TO RIGHT

- ✓ To display the list, we have to traverse (move) the circular linked list, node by node from the first node, until the end of the list is reached.
- ✓ Traversing a list involves the following steps.

  1. Assign the address of start pointer to a temp pointer.
  2. Display the information from the data field of each node.
- ✓ The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
        node *temp;
        temp = start;
        printf(" The contents of List (Left to Right)");
        if(start == NULL )
                printf(" Empty List ");
```

```
            else
            {
                     do
                     {
                              printf("%d", temp -> data);
                              temp = temp -> next;
                     } while (temp != start);
            }
}
```



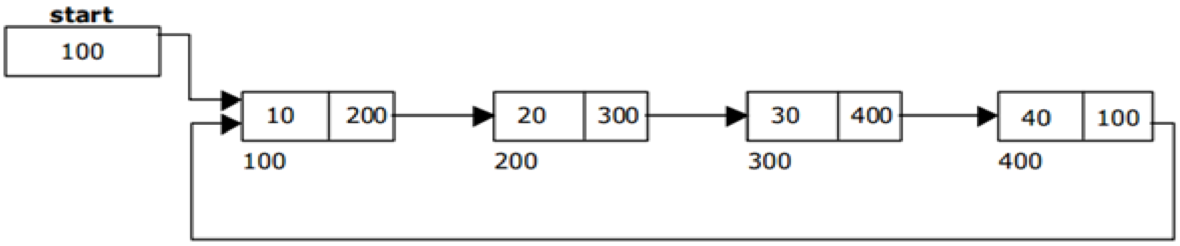
Circular Linked List with 4 nodes

SEARCHING A NODE IN A CIRCULAR LINKED LIST

- ✓ Searching a circular linked list means to find a particular element in the circular linked list.
- ✓ A circular linked list consists of nodes which are divided into two parts, the data part and the next part.
- ✓ So searching means finding whether a given value is present in the data part of the node or not.
- ✓ If it is present, then display element found otherwise element not found.

```
void search()
{
         node *temp;
         int value = 30;
         temp = start;
```

```
            if(start == NULL )
                    printf(" Empty List ");
            else
            {
                    while (temp->next != start)
                    {
                            if(value = temp->data)
                            {
                                    printf(" Element found ");
                                    return;
                            }
                            temp = temp -> next;
                    }
                    printf(" Element not found ");
            }
    }
```



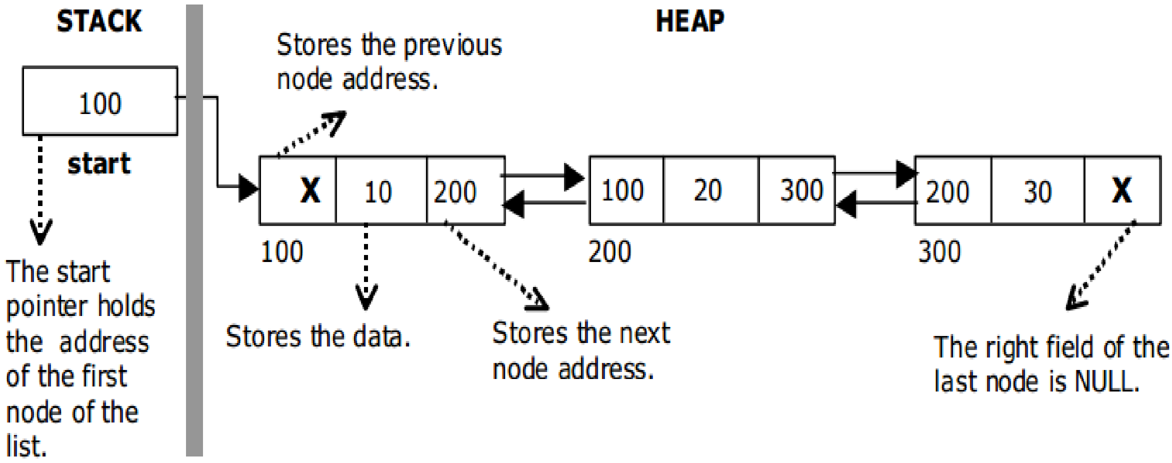
Circular Linked List with 4 nodes

DOUBLY LINKED LSITS

- ✓ A double linked list is a **two-way** list in which all nodes will have **two** links.
- ✓ This helps in accessing both **successor node** and **predecessor node** from the given node position.
- ✓ It provides **bi-directional** traversing.
- ✓ Each node has **three** fields namely
    - ✓ Left link
    - ✓ Data

- ✓ Right link

- ✓ The **left link** points to the predecessor node and the **right link** points to the successor node. The **data** field stores the required data. The beginning of the double linked list is stored in a "**start**" pointer which points to the first node.
- ✓ The first node's left link and last node's right link is set to **NULL**.


STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
Stores the previous node address.
X
10
200
100
20
300
200
30
X
100
200
300
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

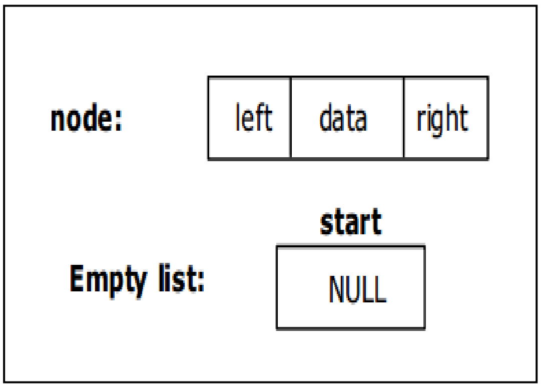
Doubly Linked List Representation

IMPLEMENTATION OF DOUBLY LINKED LIST

Before writing the code to build the list, we need to create a **start** node**,** used to create and access other nodes in the linked list.

- ✓ Creating a structure with one data item and a right pointer, which will be pointing to next node of the list and left pointer pointing to the previous node. This is called as **self-referential structure**.
- ✓ Initialize the start pointer to be **NULL**.

```
struct dlinklist
{
        struct dlinklist * left;
        int data;
        struct dlinklist * right;
};
typedef struct dlinklist node;
node *start = NULL;
```



BASIC OPERATION PERFORMED ON DOUBLY LINKED LIST

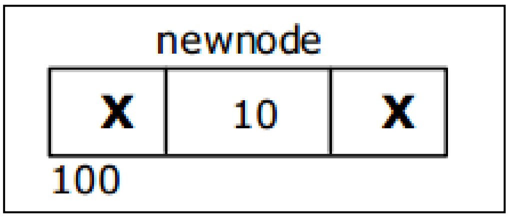
The different operations performed on the doubly linked list are listed as follows.

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Display

Creating a node for Doubly Linked List

- ✓ Creating a double linked list starts with creating a node.
- ✓ Sufficient memory has to be allocated for creating a node.
- ✓ The information is stored in the memory, allocated by using the **malloc()** function.
- ✓ The function **getnode(),** is used for **creating a node**, after allocating memory for the node, the information for the node **data part** has to be read from the user and set **left** and **right** fields to **NULL** and finally return the **node.**

```
node* getnode()
{
        node* newnode;
        newnode = new node;
        printf(" Enter data ");
        scanf("%d", &newnode  -> data);
        newnode -> left = NULL;
        newnode -> right = NULL;
        return newnode;
}
```



Creating a Doubly Linked List with 'n' number of nodes

The following steps are to be followed to create 'n' number of nodes.

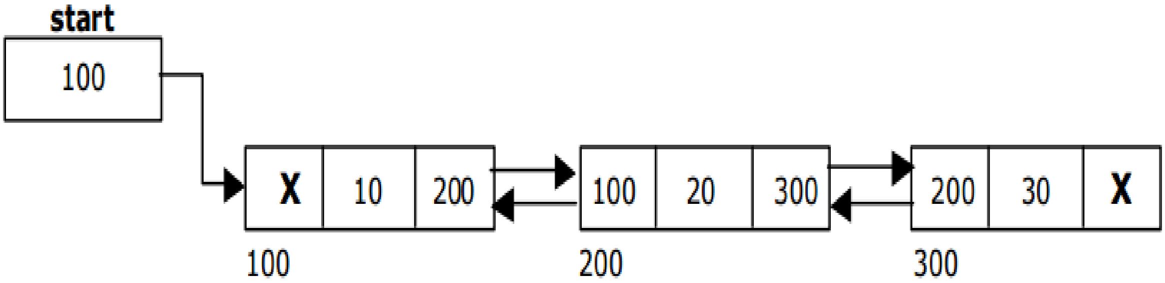
1. Get the new node using getnode().

newnode = getnode();

2. If the list is empty, assign new node as start.

start = newnode;

3. If the list is not empty, follow the steps given below.
   - ✓ The left field of the new node is made to point the previous node.
   - ✓ The previous nodes right field must be assigned with address of the new node.
4. Repeat the above steps ‘n’ times.



Double Linked List with 3 nodes

The function createlist(), is used to create ‘n’ number of nodes

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n ; i++)
        {
                newnode = getnode();
                if(start = = NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
```

```
                    while(temp -> right != NULL)
                    {
                            temp = temp -> right;
                    }
                    temp -> right = newnode;
                    newnode -> left = temp;
            }
    }
}
```

INSERTION OF A NODE

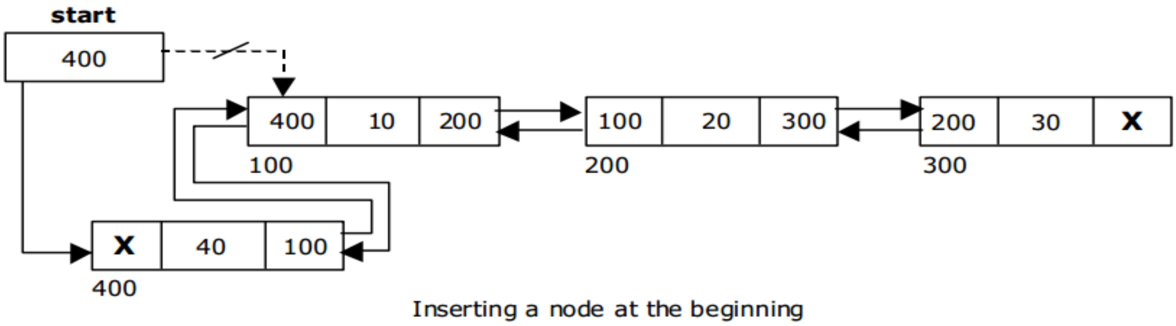
- ✓ One of the most important operation that can be done in a doubly linked list is the insertion of a node.
- ✓ Memory is to be allocated for the **newnode** before reading the data.
- ✓ The **newnode** will contain **empty data** field and **empty left** and **right fields.**
- ✓ The data field of the newnode is then stored with the information read from the user.
- ✓ The left and right fields of the newnode are set to **NULL.**
- ✓ The newnode can then be inserted at three different places namely:
    - ✓ **Inserting a node at the beginning.**
    - ✓ **Inserting a node at the end.**
    - ✓ **Inserting a node at specified position.**

INSERTING A NODE AT THE BEGINNING

The following steps are to be followed to insert a newnode at the beginning of the list:

1. Get the newnode using getnode() then newnode = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty, follow the steps given below:

```
newnode -> right = start;
start -> left = newnode;
start = newnode;
```



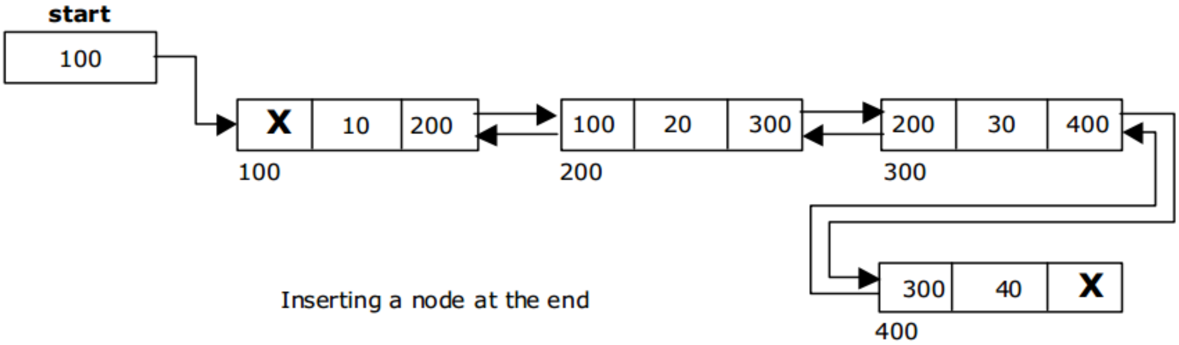
Inserting a node at the beginning

INSERTING A NODE AT THE END

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode() then newnode = getnode();
2. If the list is empty then start = newnode.
3. If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
        temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```



Inserting a node at the end

INSERTING A NODE AT SPECIFIED POSITION

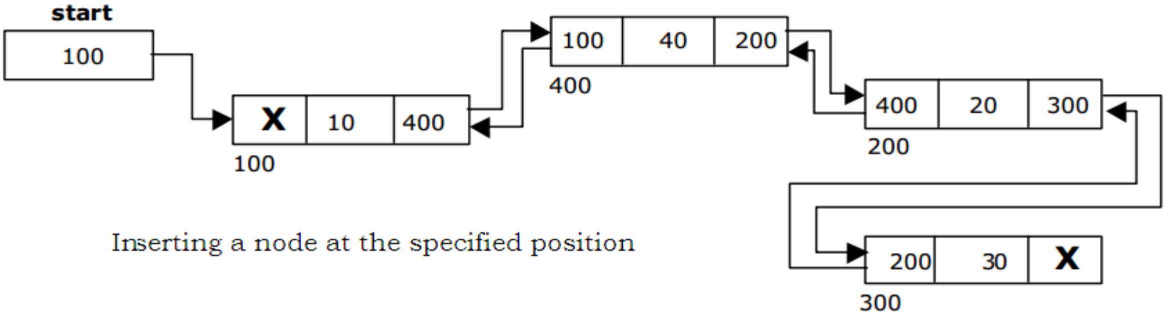
The following steps are followed, to insert a new node in an intermediate position in the list:

1. Get the new node using getnode() then newnode = getnode();
2. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

3. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

4. After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode ->right  = temp ->right;
temp -> right ->left = newnode;
temp  -> right = newnode;
```



Inserting a node at the specified position

DELETION OF A NODE

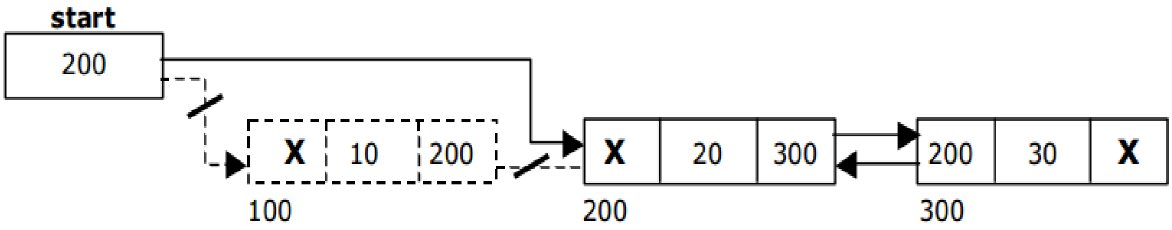
- ✓ Another operation that can be done in a doubly linked list is the deletion of a node.
- ✓ Memory is to be released for the node to be deleted.
- ✓ A node can be deleted from the list from three different places.
  - ✓ **Deleting a node at the beginning.**
  - ✓ **Deleting a node at the end.**
  - ✓ **Deleting a node at specified position.**

DELETING A NODE AT THE BEGINNING

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display ‘Empty List’ message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```



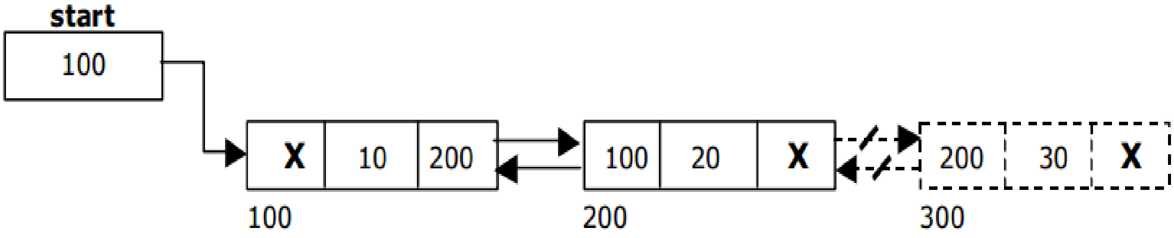
Deleting a node at beginning

DELETING A NODE AT THE END

The following steps are followed to delete a node at the end of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
        temp = temp ->right;
}
temp –> left -> right = NULL;
free(temp);
```



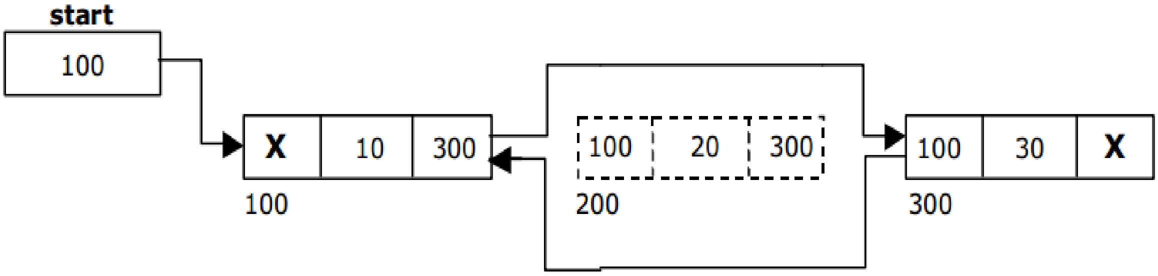
Deleting a node at the end

DELETING A NODE AT SPECIFIED POSITION

The following steps are followed, to delete a node from the specified position in the list.

1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps given below.

```
if(pos > 1 && pos < nodectr)
{
        temp = start;
        ctr = 1;
        while(ctr < pos)
        {
                temp = temp -> right;
                ctr++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
}
```



Deleting a node at the specified position

TRAVERSAL AND DISPLAYING A LIST

- ✓ To display the list, we have to traverse (move) the double linked list, node by node from the first node, until the end of the list is reached.
- ✓ To traverse double linked list from left to rightwe have the following steps:

  1. If list is empty then display **'Empty List'** message.
  2. If the list is not empty, follow the steps given below:

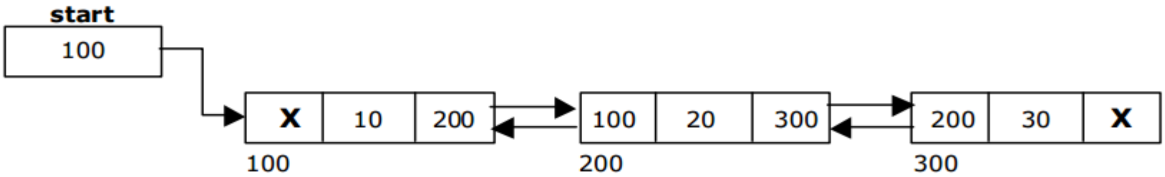
```
        temp = start;
        while(temp != NULL)
        {
                printf("%d", temp -> data);
                temp = temp -> right;
}
```

- ✓ To display the list, we have to traverse (move) the double linked list, node by node from the first node, until the end of the list is reached.

- ✓ The following steps are followed, to traverse a list from left to right:

  1. If list is empty then display 'Empty List' message.
  2. If the list is not empty, follow the steps given below:

```
        temp = start;
        while(temp!= NULL)
        {
                printf("%d", temp -> data);
                temp = temp -> right;
}
```



Double Linked List with 3 nodes

COUNTING THE NUMBER OF NODES

The following code will count the number of nodes exist in the list (using recursion).

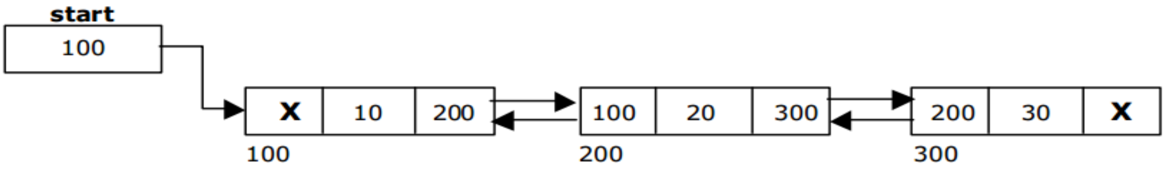
```
int countnode(node *start)
{
        if(start = = NULL)
                return 0;
        else
                return(1 + countnode(start ->right ));
}
```

SEARCHING A NODE IN A DOUBLE LINKED LIST

- ✓ Searching a double linked list means to find a particular element in the double linked list.
- ✓ A double linked list consists of nodes which are divided into two parts, the data part and the next part.
- ✓ So searching means finding whether a given value is present in the data part of the node or not.

- ✓ If it is present, then display element found otherwise element not found.

```
void search()
{
        node *temp;
        int value = 30;
        temp = start;
        if(start == NULL )
                printf(" Empty List ");
        else
        {
                while (temp->right != NULL)
                {
                        if(value = temp->data)
                        {
                                printf(" Element found ");
                                return;
                        }
                        temp = temp -> right;
                }
                printf(" Element not found ");
        }
}
```



Double Linked List with 3 nodes

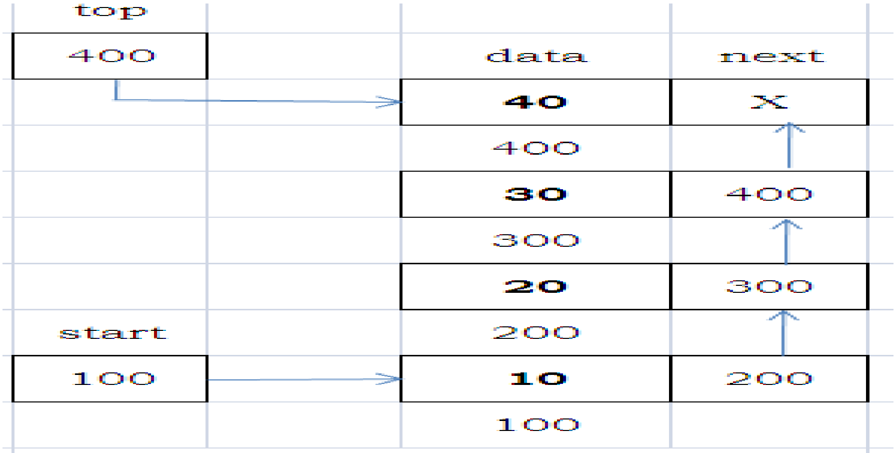
LINKED STACKS

- ✓ A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end.
- ✓ This end is known as **top** of stack.

- ✓ When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop.
- ✓ Stack is also called as **Last-In-First-Out** (LIFO) list.
- ✓ The element that is inserted last is the first element to be removed from the stack.
- ✓ Stack can be implemented using linked list and the same operations can be performed at the end of the list using top pointer.

REPRESENTATION OF STACK USING LINKED LIST

- ✓ A stack is represented using an array is easy, but the drawback is that the array must be declared to have some fixed size.
- ✓ In case the stack is a very small or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.
- ✓ But if the array size cannot be determined in advance, then linked representation is used.
- ✓ The storage requirement of linked representation of the stack with n elements is O(n), and the time requirement for the operations is O(1).
- ✓ In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node.
- ✓ The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP.
- ✓ If TOP = NULL, then it indicates that the stack is empty.



Linked Stack Representation

OPERATIONS ON LINKED STACKS

- ✓ There are three possible operations performed on a stack. They are push, pop and peek.
    - ✓ **Push: Allows adding an element at the top of the stack.**
    - ✓ **Pop: Allows removing an element from the top of the stack.**
    - ✓ **Peek: it returns the value of topmost element of the stack**

Push Operation

- ✓ Create a temporary node and store the value of x in the data part of the node.
- ✓ Now make next part of temp point to top and then top point to temp.
- ✓ That will make the newnode as the topmost element in the stack.

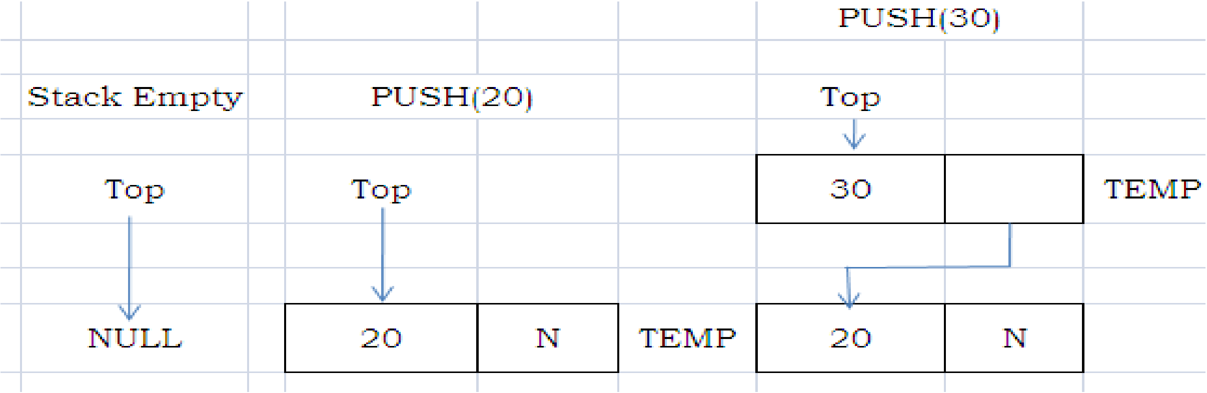
Algorithm for PUSH Operation

```
Step 1: Allocate memory for the temporary node and name it as temp
Step 2: Set temp - > data = x
Step 3: if top = NULL
                Set temp - > next = NULL
                Set top = temp
        else
                Set temp - > next = top
                Set top = temp
Step 4: Exit
```

EXAMPLE

- ✓ The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.
- ✓ To insert an element with value 20, we first check if top=NULL. Then we allocate memory for a newnode(temp), store the value in its data part and NULL in its next part.

- ✓ The newnode(temp) will then be called top. However, if top!=NULL, then we insert the newnode(temp) at the beginning of the linked stack and name this newnode(temp) as top.



Pop Operation

- ✓ The data in the topmost node of the stack is first stored in a variable called x.
- ✓ Then a temporary pointer is created to point to top.
- ✓ The top is now safely moved to the next node below it in the stack.
- ✓ Temp node is deleted and the item is returned.

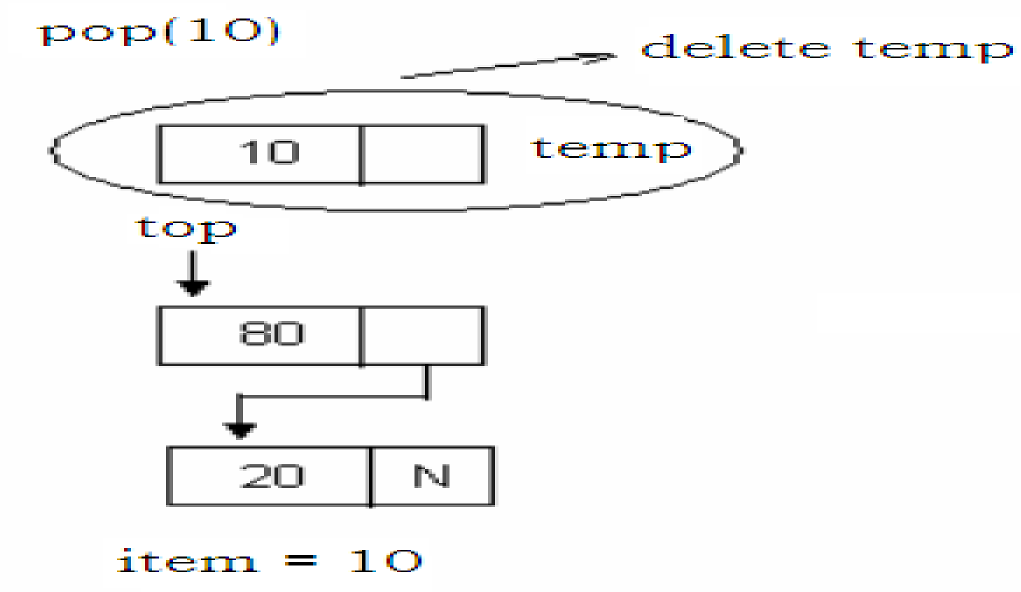
Algorithm for POP Operation

```
Step 1: if top = NULL
                display Underflow and goto step 6
Step 2: Set x = top - > data
Step 3: Set temp = top
Step 4: Set top = top - > next
Step 5: free temp
Step 6: Exit
```

EXAMPLE:

- ✓ The pop operation is used to delete the topmost element from a stack. Before deleting the value, we must first check if top=NULL, then we display stack is empty and no more deletions can be done.
- ✓ If an attempt is made to delete a value from a stack that is already empty, an underflow message is printed.

- ✓ In case top!=NULL, then we will delete the node pointed by top, and make top point to the second element of the linked stack.



IMPLEMENTATION OF STACKS USING LINKED LIST

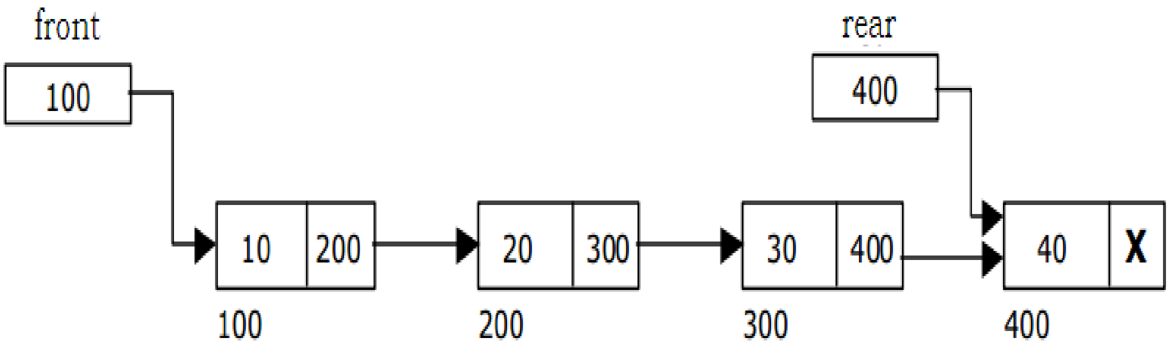
```
#include<stdio.h>
struct node
{
        int data;
        struct node *next;
}*top = NULL;
void push(int);
void pop();
void display();
int main(void)
{
        int choice, value;
        clrscr();
        printf("\n:: Stack using Linked List ::\n");
        while(1)
        {
            printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
            printf("Enter your choice: ");
```

```
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:         printf("Enter the value to be insert: ");
                                        scanf("%d", &value);
                                        push(value);
                                        break;
                        case 2:         pop(); break;
                        case 3:         display(); break;
                        case 4:         exit(0);
                        default:        printf("\nWrong selection!!! Please try
                                        again!!!\n");
                }
        }
}
void push(int value)
{
        struct node *newnode;
        newnode = (struct node*)malloc(sizeof(struct node));
        newnode->data = value;
        if(top == NULL)
                newnode->next = NULL;
        else
                newnode->next = top;
        top = newnode;
        printf("\nInsertion is Success!!!\n");
}
void pop()
{
        if(top == NULL)
                printf("\nStack is Empty!!!\n");
        else
        {
                struct node *temp = top;
                printf("\nDeleted element: %d", temp->data);
                top = temp->next;
                free(temp);
        }
}
void display()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else
    {
        struct node *temp = top;
        while(temp->next != NULL)
        {
                printf("%d--->",temp->data);
                temp = temp -> next;
        }
        printf("%d--->NULL",temp->data);
    }
}
```

LINKED QUEUES AND ITS REPRESENTATION

- ✓ Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end.
- ✓ The end at which the deletion of an element take place is called **front**, and the end at which insertion of a new element can take place is called **rear**.
- ✓ The deletion or insertion of elements can take place only at the front or rear end called **dequeue** and **enqueue**.

- ✓ The first element that gets added into the queue is the first one to get removed from the queue.
- ✓ Hence the queue is referred to as **First-In-First-Out** list (FIFO).
- ✓ We can perform the similar operations on two ends of the list using two pointers **front pointer** and **rear pointer**.



Linked Queue Representation

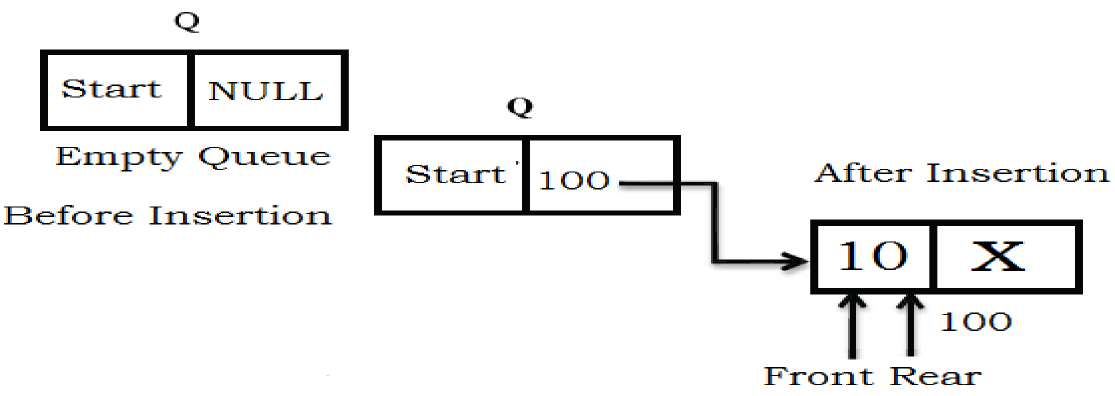
OPERATIONS ON QUEUES USING LINKED LIST

Enqueue operation

- ✓ In linked list representation of queue, the addition of new element to the queue takes place at the rear end.
- ✓ It is the normal operation of adding a node at the end of a list.

Algorithm for Enqueue(inserting an element)

```
Allocate memory for the new node and name it as temp
        set newnode - > data = value
        set newnode -> next = NULL
        if (front = NULL) then
                set rear = front = newnode
                set rear - > next = front -> next = NULL
        else
                set rear - > next = temp
                set rear = rear - > next
                set rear -> next = NULL
```

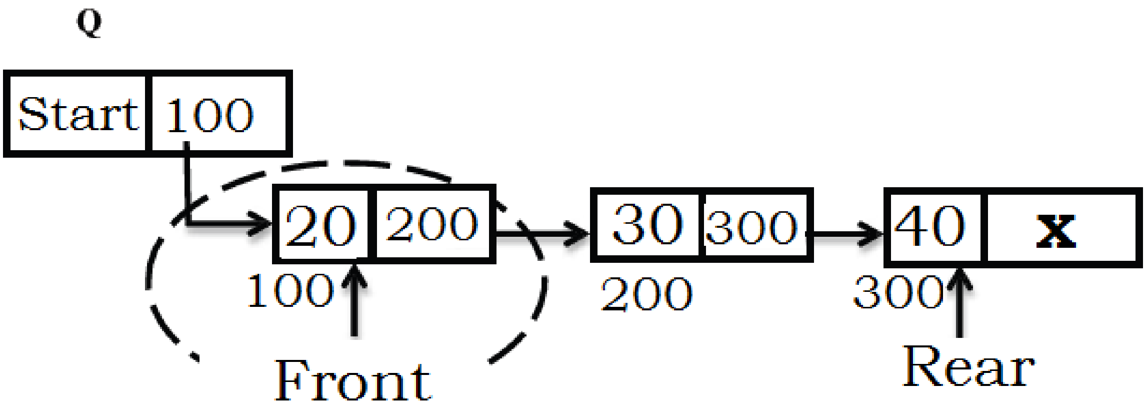


Dequeue operation

- ✓ The dequeue operation deletes the first element from the front end of the queue.
- ✓ Initially it is checked, if the queue is empty.
- ✓ If it is not empty, then return the value in the node pointed by front, and moves the front pointer to the next node.

Algorithm for Dequeue(deleting an element)

```
if (front = NULL)
        display "Queue is empty"
        return
else
        while(front != NULL)
                temp = front
                front = front - > next
                free(temp)
```



IMPLEMENTATION OF QUEUES USING LINKED LIST

```
#include<stdio.h>
#include <stdlib.h>
struct queue
{
        int data;
        struct queue *next;
};
typedef struct queue node;
node *front = NULL;
node *rear = NULL;
node* getnode()
{
        node *temp;
        temp = (node *) malloc(sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void insertQ()
{
        node *newnode;
        newnode = getnode();
        if(newnode == NULL)
        {
                printf("\n Queue Full");
                return;
        }
        if(front == NULL)
        {
                front = newnode;
                rear = newnode;
        }
        else
        {
                rear -> next = newnode;
                rear = newnode;
        }
        printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t Empty Queue..");
                return;
        }
        temp = front;
        front = front -> next;
        printf("\n\n\t Deleted element from queue is %d ", temp ->data);
        free(temp);
}
void displayQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t\t Empty Queue ");
        }
        else
        {
                temp = front;
        printf("\n\n\n\t\t Elements in the Queue are: ");
        while(temp != NULL )
        {
            printf("%5d ", temp -> data);
            temp = temp -> next;
        }
    }
}
char menu()
{
    char ch;
    clrscr();
    printf("\n \t..Queue operations using pointers.. ");
    printf("\n\t -----------**********-------------\n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice: ");
    ch = getche();
    return ch;
}
int main(void)
{
    char ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case '1' :
                    insertQ();
```

```
                                        break;
                        case '2' :
                                        deleteQ();
                                        break;
                        case '3' :
                                        displayQ();
                                        break;
                        case '4':
                                        return;
                }
        } while(ch != '4');
        return 0;
}
```

POLYNOMIALS

A polynomial is of the form

$$\sum_{i=0}^{n} c_i \, x^i$$

Where, ci is the coefficient of the i^{th} term and n is the degree of the polynomial. Some examples are:

$5x^2 + 3x + 1$

$12x^3 + 4$

4x6 + 10x4 – 5x + 3

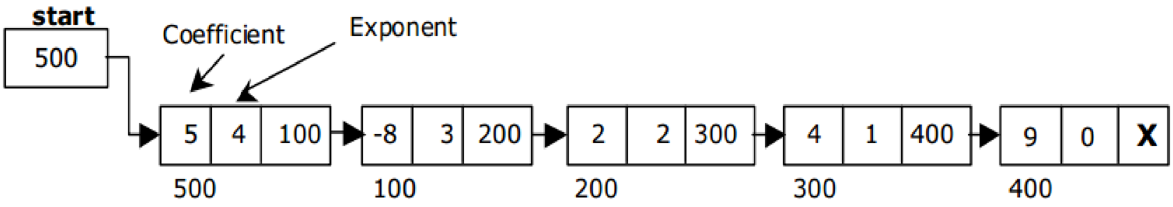
$5x^4 - 8x^3 + 2x^2 + 4x^1 + 9$

$23x^9 + 18x^7 - 41x^6 + 163x^4 - 5x + 3$

REPRESENTATION OF POLYNOMIALS

- ✓ It is not necessary to write terms of the polynomials in decreasing order of degree.
- ✓ In other words the two polynomials 1 + x and x + 1 are equivalent.
- ✓ The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent.

- ✓ Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures.
- ✓ A linked list structure that represents polynomials $5x^4 – 8x^3 + 2x^2 + 4x^1 + 9$



Single Linked List for the polynomial $F(x) = 5x^4 - 8x^3 + 2x^2 + 4x^1 + 9$

Advantages

- ✓ Save space
- ✓ Easy to maintain
- ✓ Do not need to allocate memory size initially

Disadvantages

- ✓ It is difficult to back up to the start of the list
- ✓ It is not possible to jump to the beginning of the list from the end of the list

POLYNOMIAL ADDITION

- ✓ To add two polynomials we need to scan them once.
- ✓ If we find terms with the same exponent in the two polynomials then we add the coefficients otherwise we copy the term of larger exponent into the sum and go on.
- ✓ When we reach at the end of one of the polynomial then remaining part of the other is copied into the sum.
- ✓ To add two polynomials follow the following steps:
  - ✓ Read two polynomials.
  - ✓ Add them.
  - ✓ Display the resultant polynomial.

```
#include<stdio.h>
#include<stdlib.h>
```

```
struct node
{
    int coeff;
    int pow;
    struct node *next;
};
void readpolynomial(struct node** poly)
{
    int coeff, exp, mterms;
    struct node* temp = (struct node *) malloc(sizeof(struct node));
    *poly = temp;
    do
    {
            printf(“\n Coefficient: “);
            scanf(“%d”, &coeff);
            printf(“\n Exponent: “);
            scanf(%d”, &pow);
            temp -> coeff = coeff;
            temp -> pow = exp;
            temp -> next = NULL;
            printf(“Have more terms: 1 for Y and 0 for N”);
            scanf(“%d”, &mterms);
            if(mterms)
            {
                    temp -> next = (struct node *) malloc(sizeof(struct node));
                    temp -> next = NULL;
            }
    }while(mterms);
}
void displaypolynomial(struct node* poly)
{
    printf(“\n Polynomial Expression is  “);
    while(poly!=NULL)
    {
            printf(“%dX^%d”, poly -> coeff, poly -> pow);
            poly = poly -> next;
            if(poly!=NULL)
                    printf(“ + “);
    }
}
void addpolynomial(struct node**result, struct node* first, struct node* second)
{
    struct node* temp = (struct node *)malloc(sizeof(struct node));
    temp -> next = NULL;
    *result = temp;
    while(first && second)
    {
            if(first -> pow > second -> pow)
            {
                    temp -> coeff = first -> coeff;
                    temp -> pow = first -> pow;
                    first = first -> next;
            }
            else if(first -> pow < second -> pow)
            {
                    temp -> coeff = second -> coeff;
                    temp -> pow = second -> pow;
                    second = second -> next;
            }
            else
            {
                    temp -> coeff = first -> coeff + second -> coeff;
                    temp -> pow = first -> pow;
```

```
                first = first -> next;
                second = second -> next;
            }
            if(first && second)
            {
                temp->next = (struct Node*)malloc(sizeof(struct Node));
                temp = temp->next;
                temp->next = NULL;
            }
        }
        while(first || second)
        {
            temp -> next = (struct Node*)malloc(sizeof(struct Node));
            temp  = temp -> next;
            temp -> next = NULL;
            if(first)
            {
                temp -> coeff = first -> coeff;
                temp -> pow = first -> pow;
                first = first -> next;
            }
            else if(second)
            {
                temp -> coeff = second -> coeff;
                temp -> pow = second -> pow;
                second = second -> next;
            }
        }
    }
    int main(void)
    {
        struct node* first = NULL;
        struct node* second = NULL;
        struct node* result = NULL;
        printf("\nEnter the corresponding data:-\n");
        printf("\nFirst polynomial:\n");
        readpolynomial(&first);
        displaypolynomial(first);
        printf("\nSecond polynomial:\n");
        readpolynomial(&second);
        displaypolynomial(second);
        addpolynomials(&result, first, second);
        displaypolynomial(result);
        return 0;
    }
```

SPARSE MATRIX

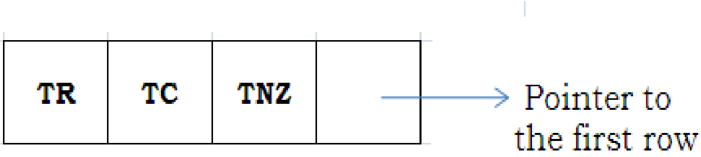
- ✓ “A matrix that contains very few number of non-zero elements is called sparse matrix”
- ✓ “A matrix that contains more number of zero values when compared with non-zero values is called a sparse matrix”

SPARSE MATRIX REPRESENTATION

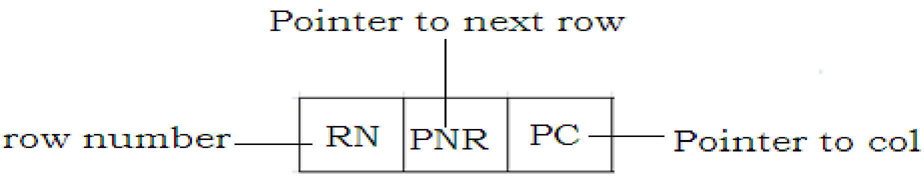
For linked representation, we need three structures.

1. head node

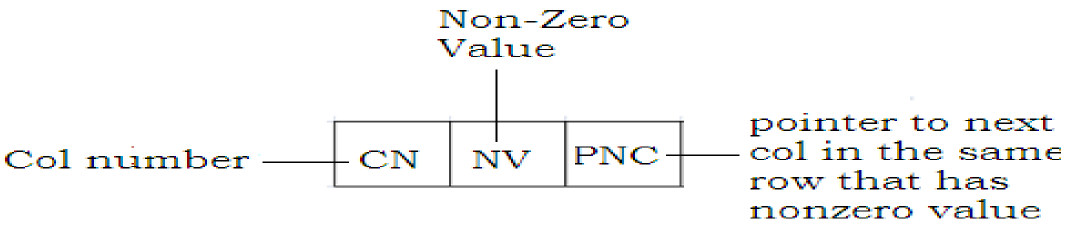
where TR - total no. of rows
TC - total no. of cols
TNZ - total no. of
Non-Zero values



2. row node



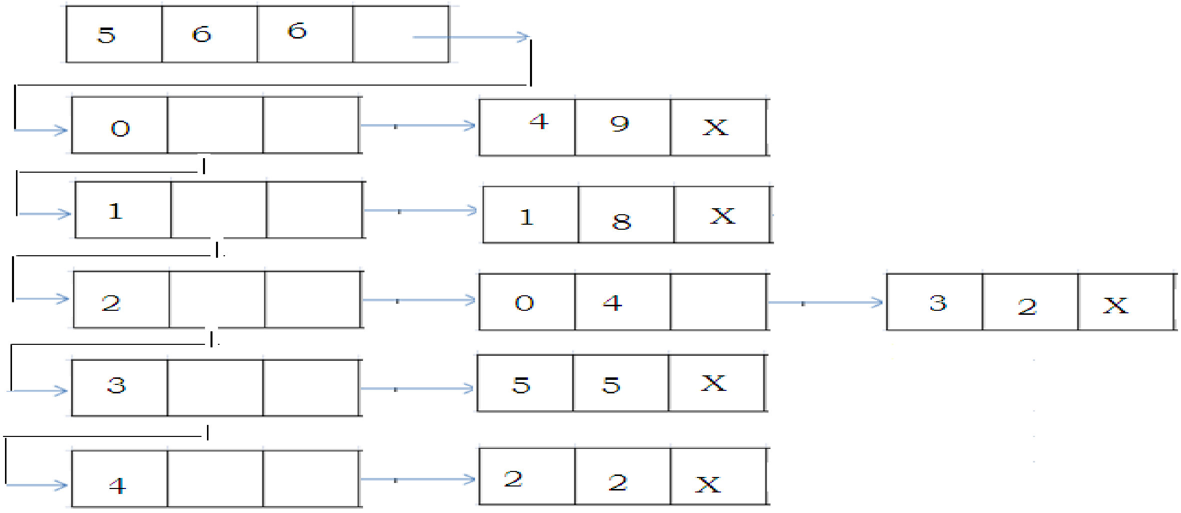
3. column node



The matrix representation for the sparse matrix is shown below for example.

$$\begin{array}{c|cccccc|} & c0 & c1 & c2 & c3 & c4 & c5 \\ r0 & 0 & 0 & 0 & 0 & 9 & 0 \\ r1 & 0 & 8 & 0 & 0 & 0 & 0 \\ r2 & 4 & 0 & 0 & 2 & 0 & 0 \\ r3 & 0 & 0 & 0 & 0 & 0 & 5 \\ r4 & 0 & 0 & 2 & 0 & 0 & 0 \end{array}_{5 \times 6}$$

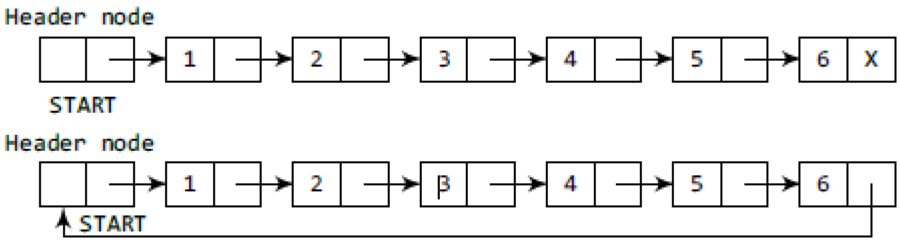
In the above matrix representation there are 5 rows, 6 columns and 6 non-zero values. The linked representation is as follows:



HEADER LINKED LIST

- ✓ A header linked list is a special type of linked list which contains a header node at the beginning of the list.
- ✓ In a header linked list, **START** will not point to the first node of the list but START will contain the address of the header node.

- ✓ The following are the two types of a header linked list:
  - ✓ ***Grounded header linked*** *list* which stores NULL in the next field of the last node.
  - ✓ ***Circular header linked list*** which stores the address of the header node in the next field of the last node. So header node will denote the end of the list.



Header linked list

- ✓ In other linked lists, if **START = NULL**, then it is an empty header linked list.
- ✓ Let us see how a grounded header linked list is stored in the memory. In a grounded header linked list, a node has two fields, DATA and NEXT.
- ✓ The DATA field will store the information part and the NEXT field will store the address of the node in sequence.
- ✓ Note that START stores the address of the header node. The NEXT field of the header node stores the address of the first node of the list.
- ✓ This node stores H. The corresponding NEXT field stores the address of the next node.
- ✓ Hence, we see that the first node can be accessed by writing **FIRST_NODE = START -> NEXT** and not by writing **START = FIRST_ NODE.**
- ✓ Let us now see how a circular header linked list is stored in the memory. The last node in this case stores the address of the header node (instead of –1).
- ✓ Hence, we see that the first node can be accessed by writing FIRST_NODE = START -> NEXT and not writing START = FIRST_NODE.

Algorithm for Insertion

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET PTR = START -> NEXT
Step 5: SET NEW_NODE -> DATA = VAL
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
Step 8: NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Algorithm for Deletion

```
Step 1: SET PTR = START->NEXT
Step 2: Repeat Steps 3 and 4 while
            PTR DATA != VAL
Step 3: SET PREPTR = PTR
Step 4: SET PTR = PTR -> NEXT
Step 5: SET PREPTR -> NEXT = PTR -> NEXT
Step 6: FREE PTR
Step 7: EXIT
```

198A1A0515

UNIT IV
Trees

Introduction Terminology
Representation of trees,
Binary trees abstract data type
Properties of binary trees
Binary tree representation
Binary tree traversals: In order, preorder, post order
Binary search trees Definition
Operations:searching BST, insert into BST, delete from a BST, Height of a BST.

Trees: Non-Linear data structure

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

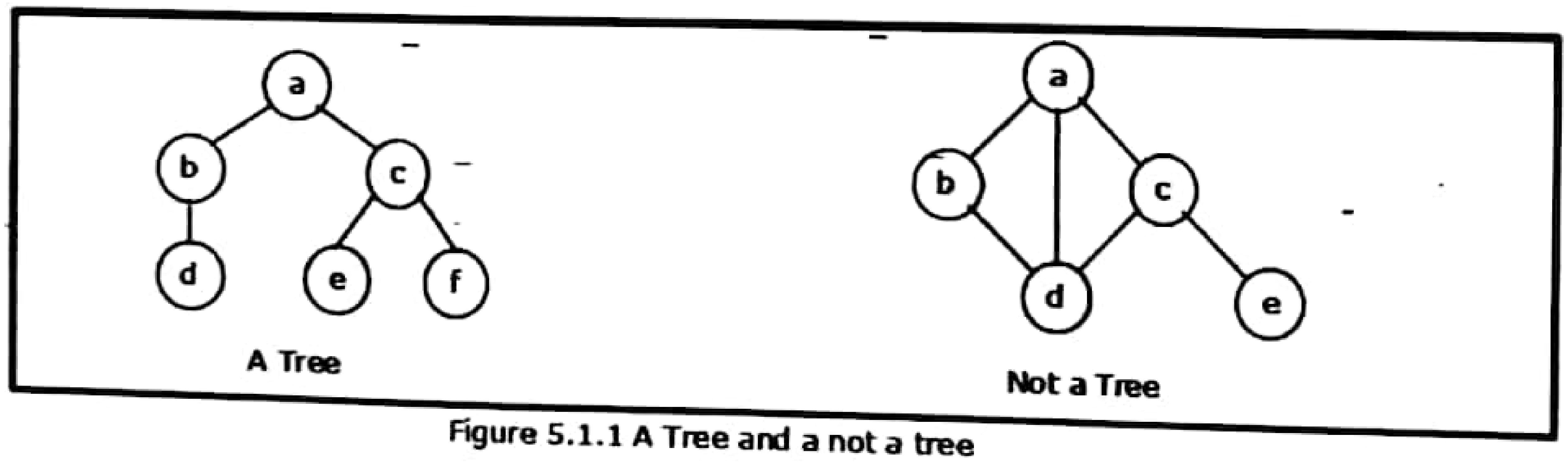


Figure 5.1.1 A Tree and a not a tree

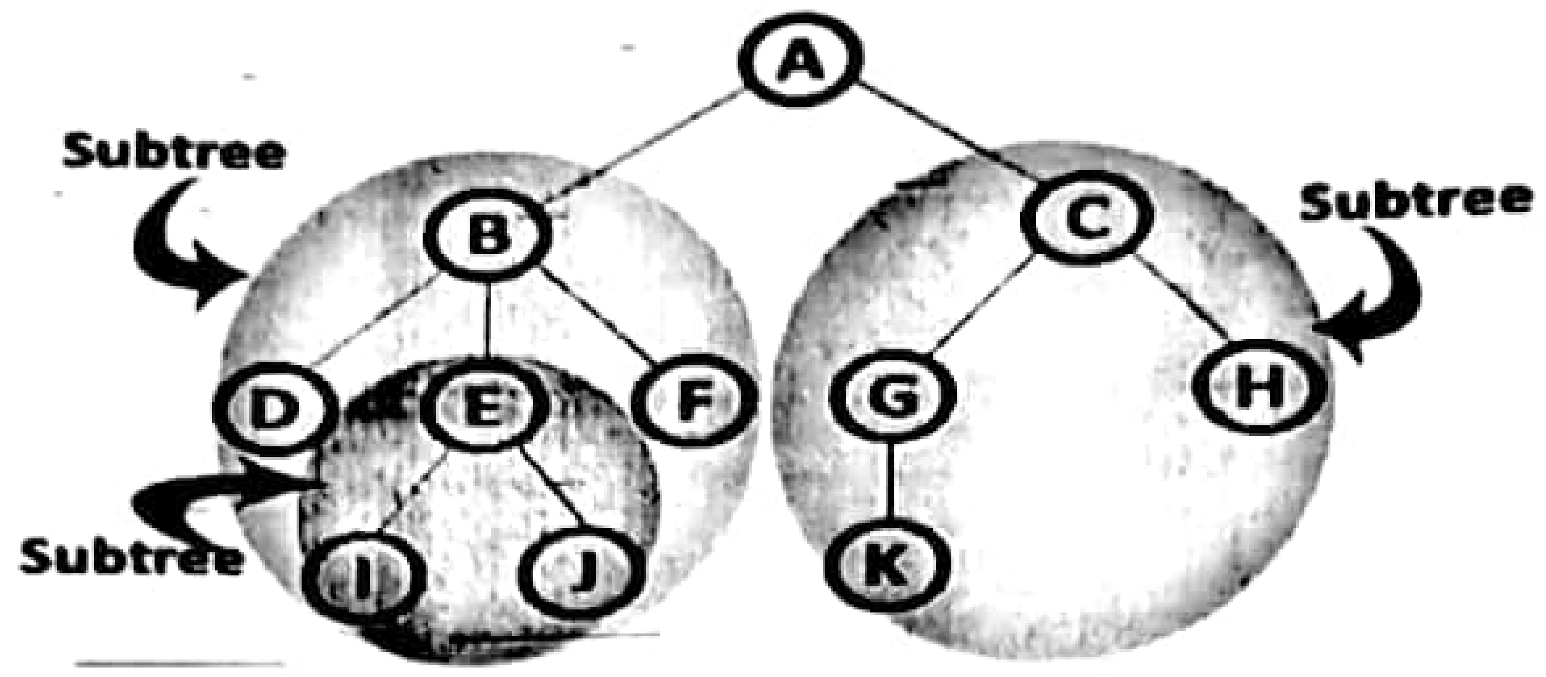
Tree is a popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

A tree is a finite set of one or more nodes such that:

There is a specially designated node called the root. The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree. We call T1, ..., Tn are the subtrees of the root.



A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

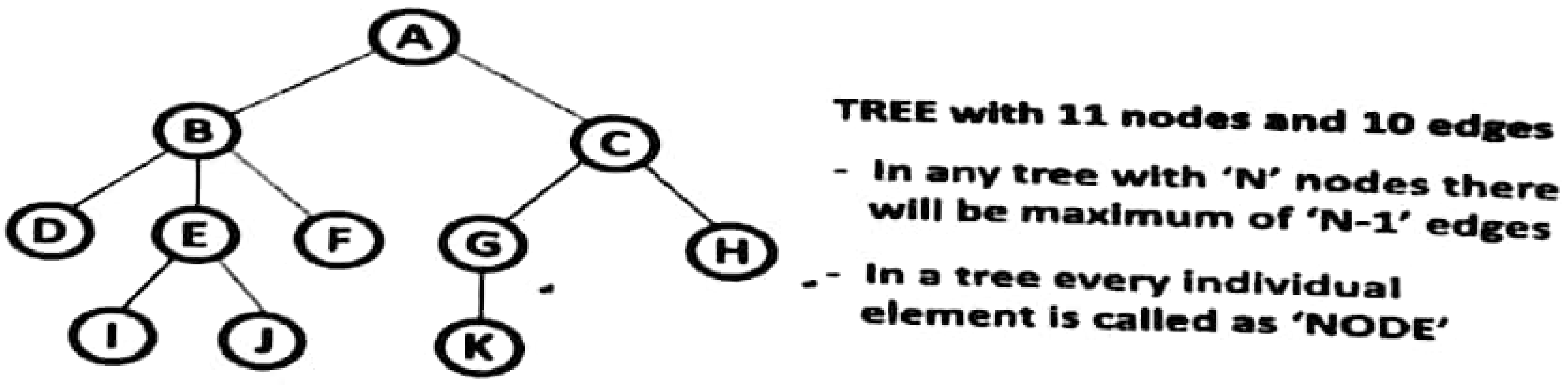
Advantages of trees

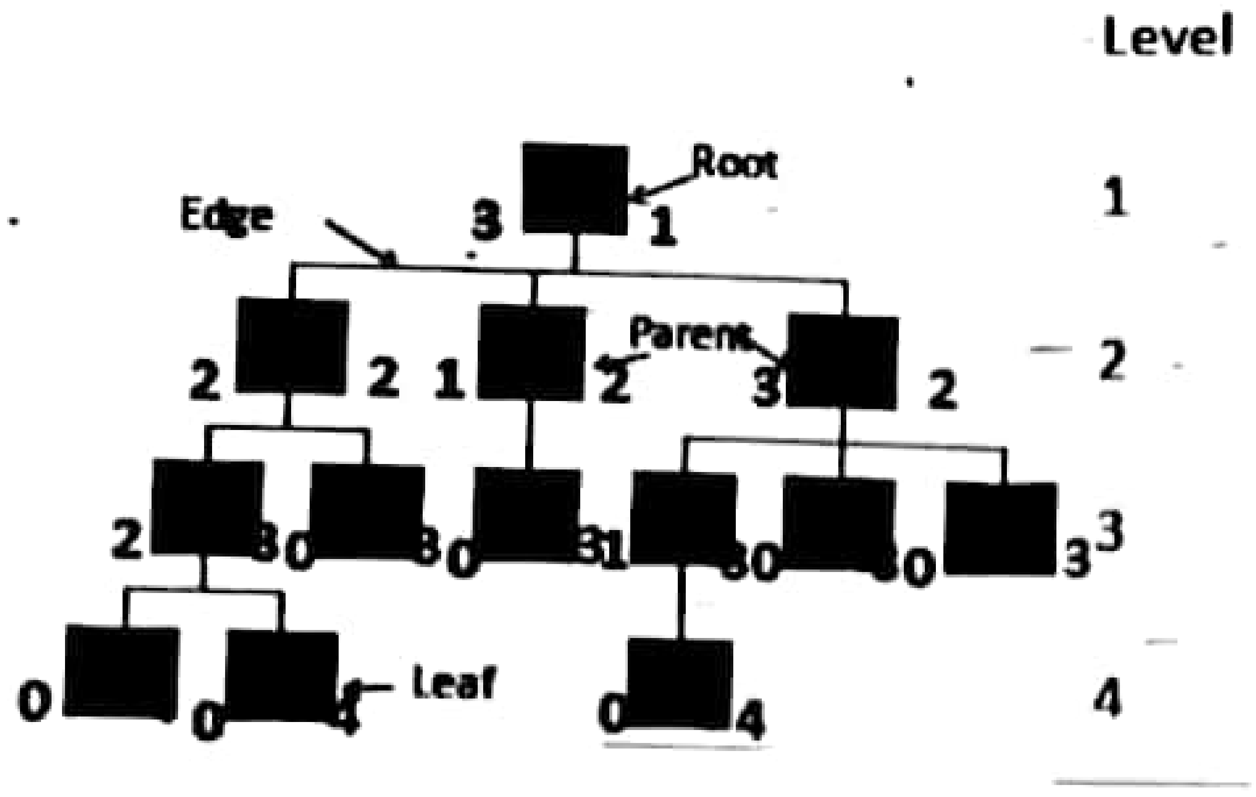
Trees are so useful and frequently used, because they have some very serious advantages:

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

Introduction Terminology

In a Tree, 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. Example


A
B
C
D
E
F
G
H
I
J
K
TREE with 11 nodes and 10 edges
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'



1. Root

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

2. Edge

In a tree data structure, 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

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children". e.g., Parent (A,B,C,D).

4. Child

In a tree data structure, the node 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. e.g., Children of D are (H, I,J).

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes. Ex: Siblings (B,C, D)

6. Leaf

In a tree data structure, the node which does not have a child (or) node with degree zero is called as LEAF Node. In simple words, a leaf is a node with no child.

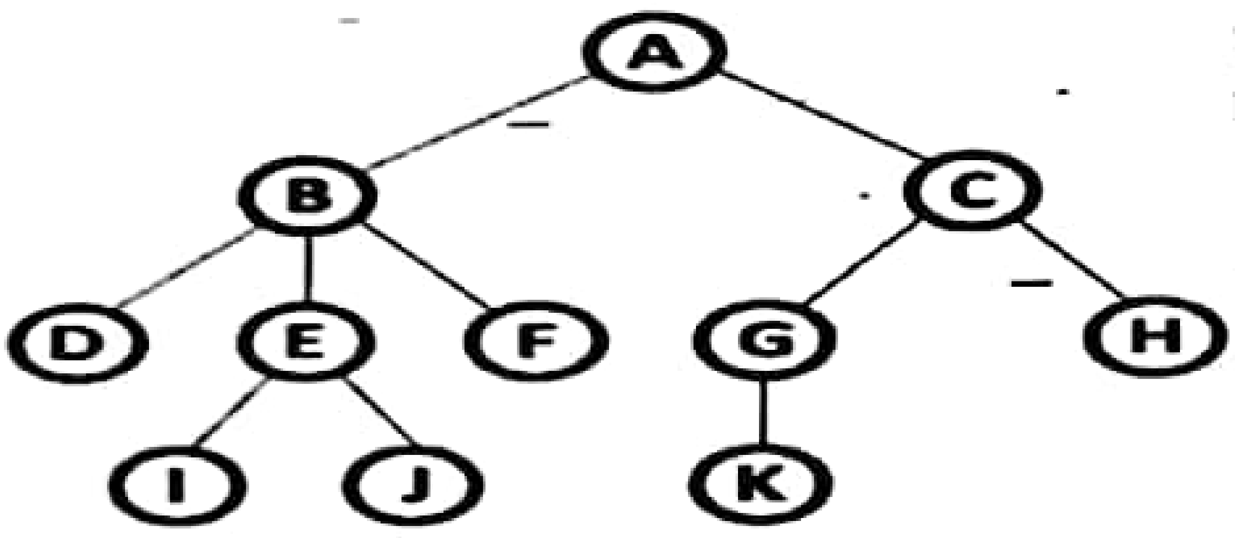
In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node. Ex: (K,L,F,G,M,I,J)

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast 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. Ex:B,C,D,E,H

8. Degree

In a tree data structure, the total number of children of a node (or)number of subtrees 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'



Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0

- **In any tree, 'Degree' a node is total number of children it has.**

9. 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 a Level and the Level count starts with '0' and incremented by one at each level (Step). Some authors start root level with 1.

10. Height

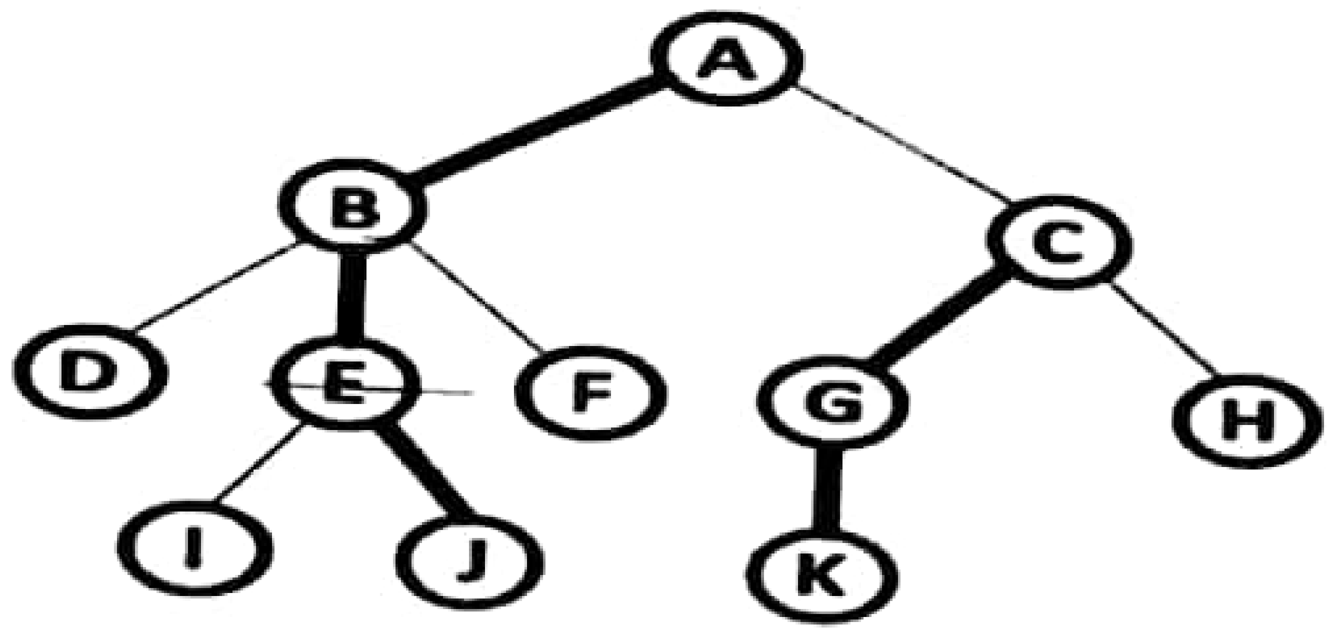
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

11. Depth

In a tree data structure, the total number of edges 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'.

12. 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 - J has length 4.



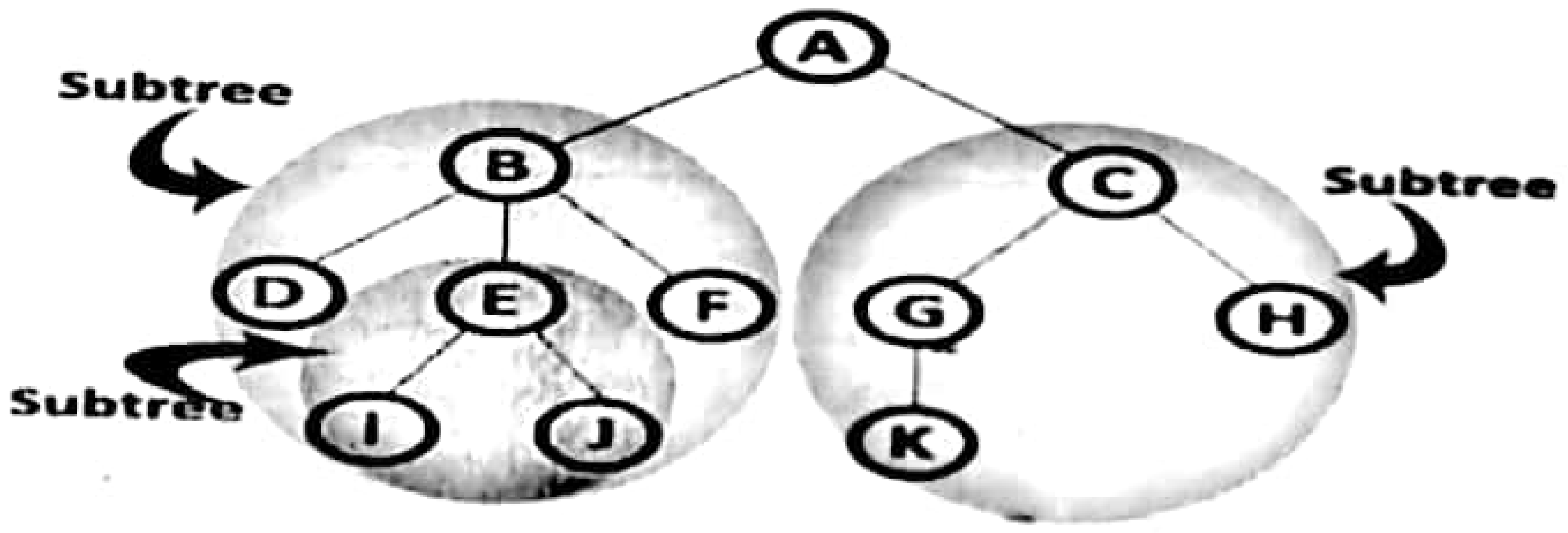
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is A - B - E - J

Here, 'Path' between C & K is C - G - K

13. 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.



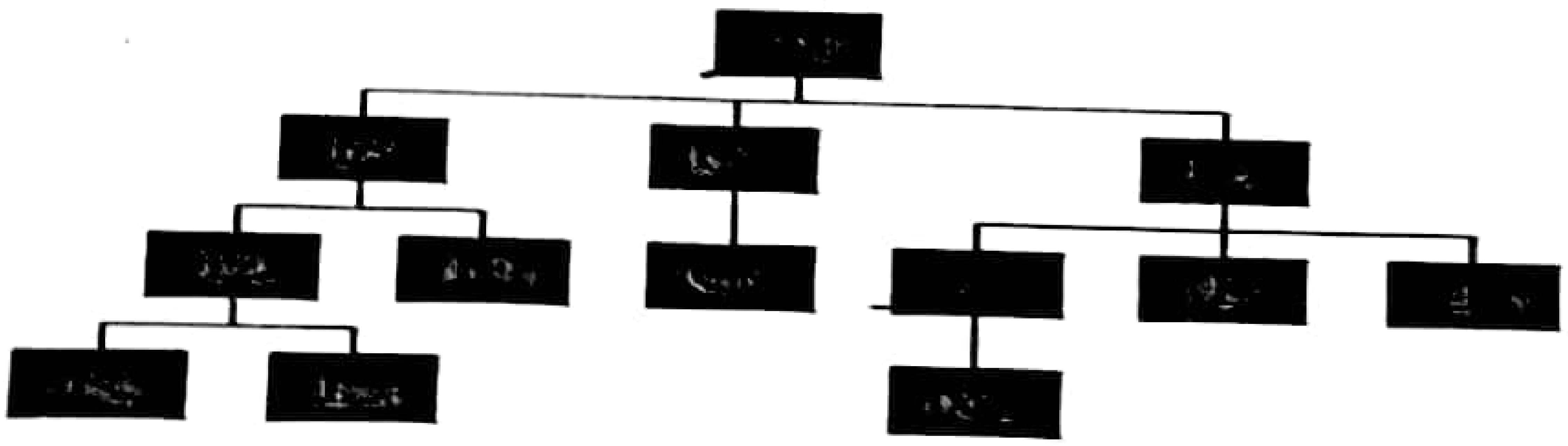
Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1.List Representation

2. Left Child - Right Sibling Representation

Consider the following tree...



1. List Representation

In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...

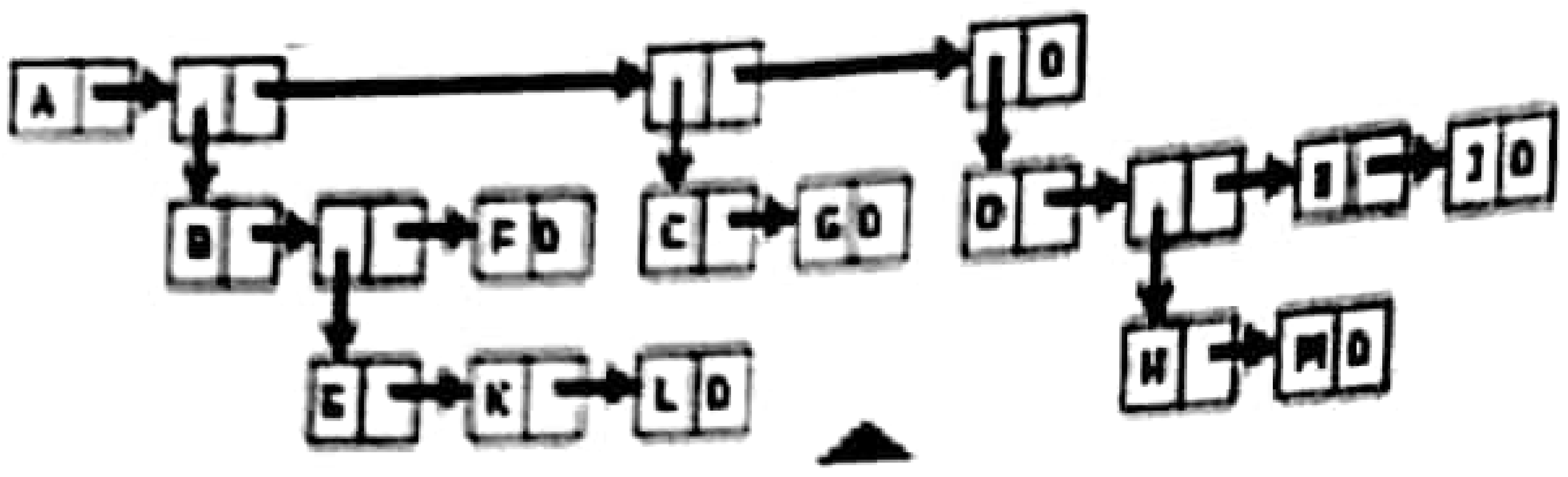


Fig: List representation of above Tree

List Representation

- (A(B(E(K,L),F),C(G),D(H(M),I,J)))
- The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link k

Fig: Possible node structure for a tree of degree k

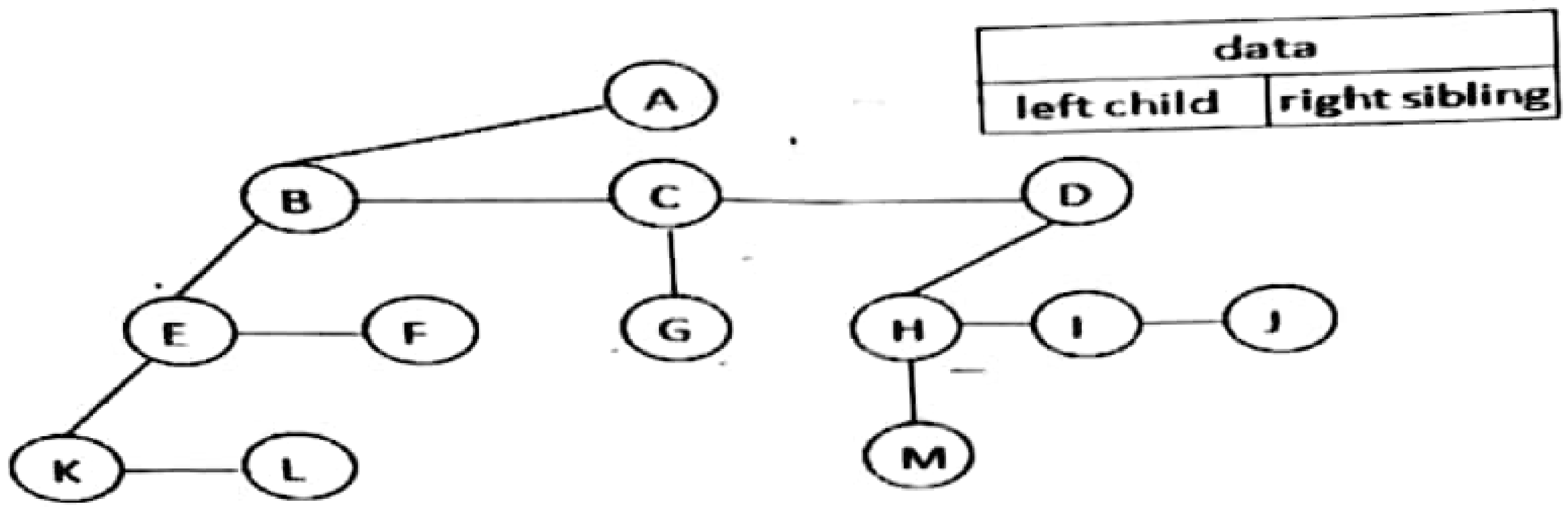
2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...

Data	
Left Child	Right Sibling

In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL.

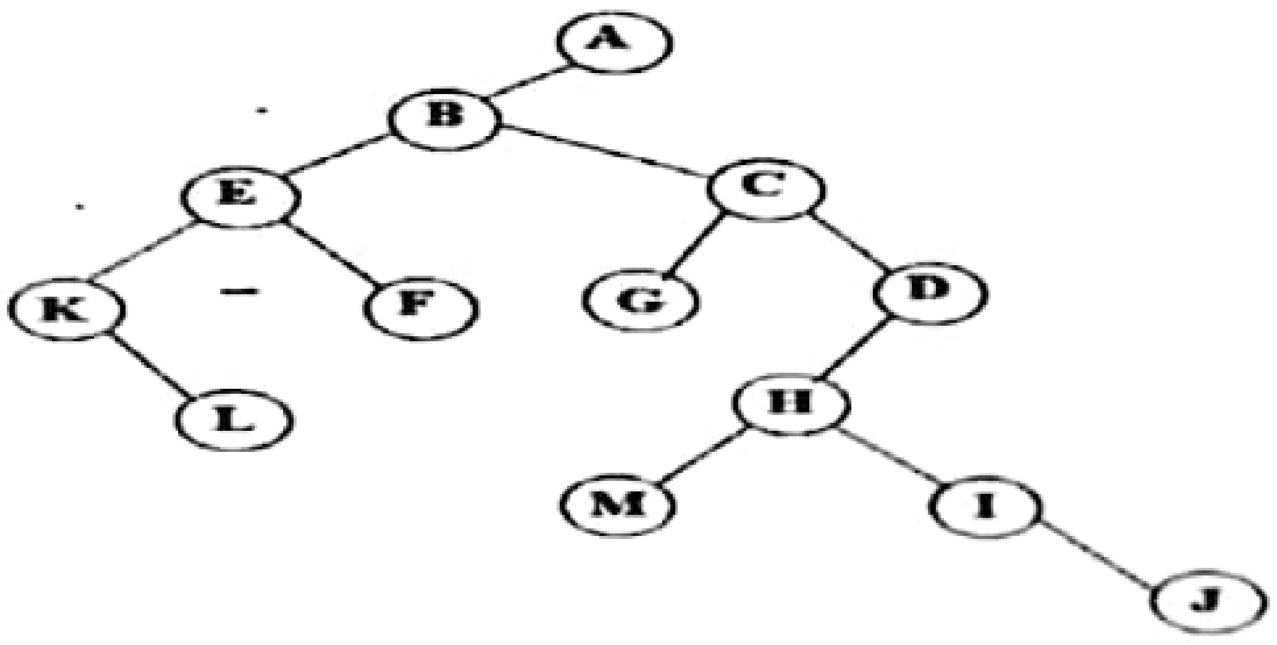
The above tree example can be represented using Left Child - Right Sibling representation as follows...



Representation as a Degree –Two Tree

To obtain degree-two tree representation of a tree, rotate the right- sibling pointers in the left child-right sibling tree clockwise by 45 degrees. In a degree-two representation, the two children of anode are referred as left and right children.

*Figure 5.6: Left child-right child tree representation of a tree (p.191)



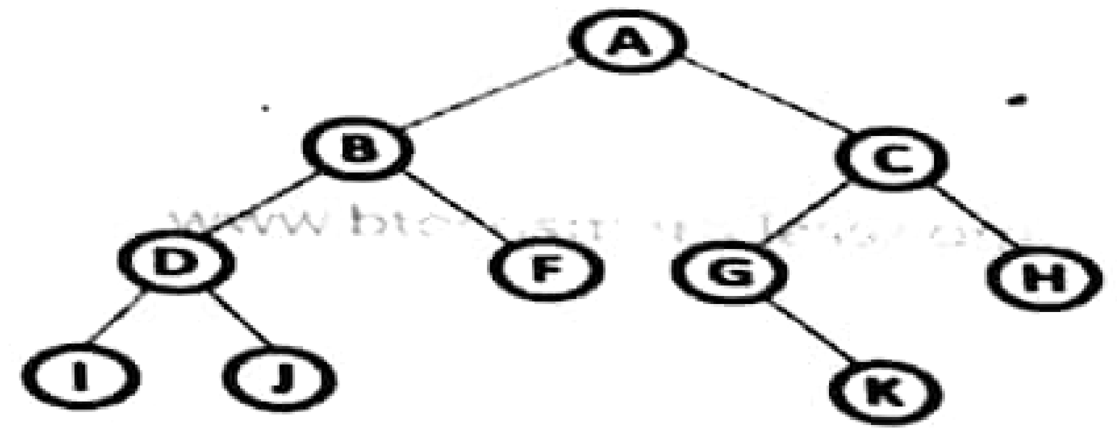
Binary Trees

Introduction

In a normal tree, every node can have any number of children. 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 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 as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children. Example



There are different types of binary trees and they are...

1. Strictly 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. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree

2. Complete Binary Tree

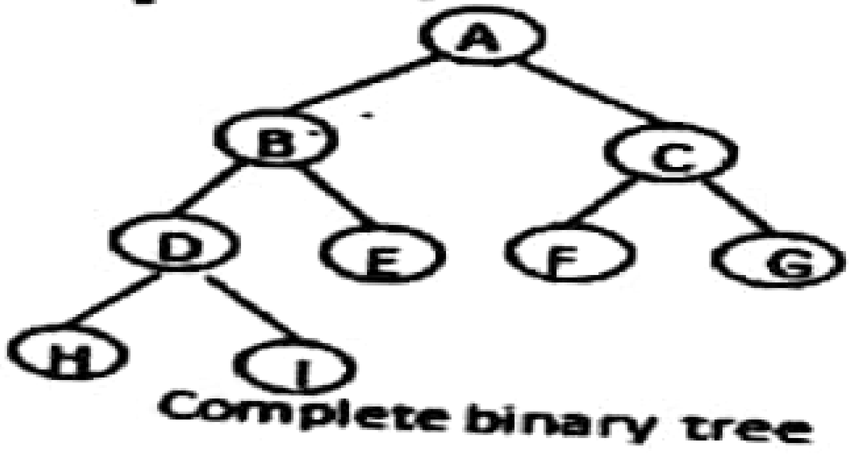
In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2 level number of nodes. 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

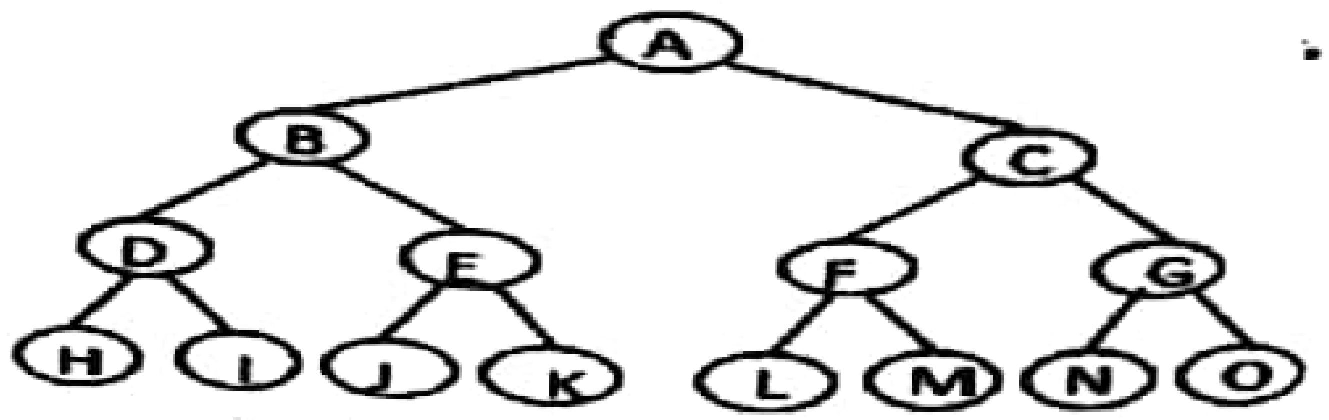
Full BT VS Complete BT

- A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes, k>=0.
- A binary tree with n nodes and depth k is complete *iff* its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.



Complete binary tree

CHAPTER 5



Full binary tree of depth 4

3. Extended Binary Tree

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

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

Abstract Data Type

Definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left subtree and right subtree.

ADT contains specification for the binary tree ADT.

Structure *Binary_Tree*(abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

Functions:

for all *bt*, *bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

Bintree Create()::= creates an empty binary tree

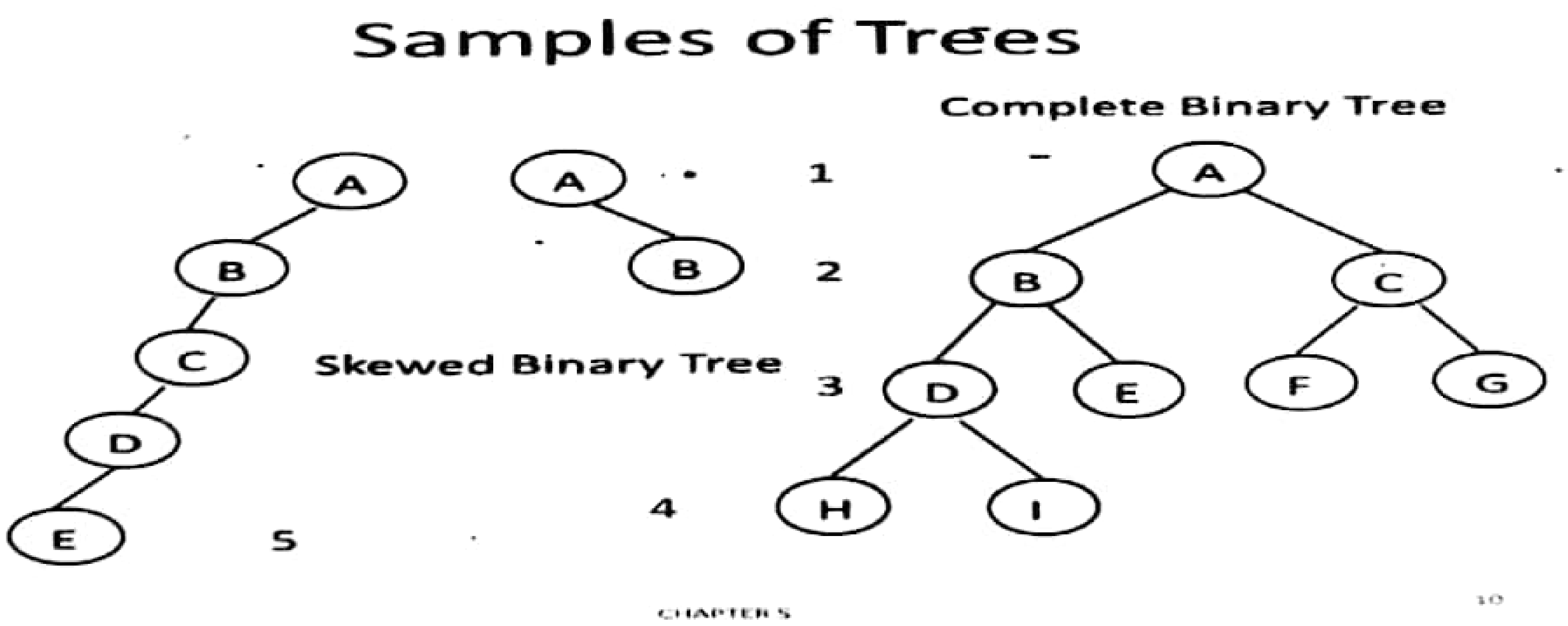
Boolean IsEmpty(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error else return the left subtree of *bt*

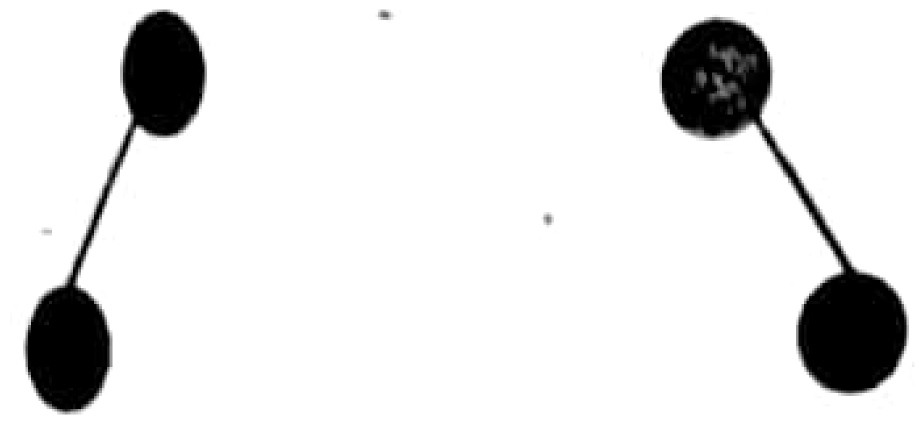
element Data(*bt*)::= if (IsEmpty(*bt*)) return error else return the data in the root node of *bt*

Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error else return the right subtree of *bt*



Differences between A Tree and A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



Above two trees are different when viewed as binary trees. But same when viewed as trees.

Properties of Binary Trees

1.Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1.

Proof By Induction:

Induction Base: The root is the only node on level i=1.Hence ,the maximum number of nodes on level i=1 is $2^{i-1}=2^0=1$.

Induction Hypothesis: Let I be an arbitrary positive integer greater than 1.Assume that maximum number of nodes on level i-1 is 2^{i-2}.

Induction Step: The maximum number of nodes on level i-1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2,the maximum number of nodes on level i is two times the maximum number of nodes on level i-1,or 2^{i-1}.

The maximum number of nodes in a binary tree of depth k is $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$

2.Relation between number of leaf nodes and degree-2 nodes: For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$.

PROOF: Let *n* and *B* denote the total number of nodes and branches in *T*. Let n_0, n_1, n_2 represent the nodes with zero children, single child, and two children respectively.

$$B+1=n \rightarrow B=n_1+2n_2 \Longrightarrow n_1+2n_2+1=n,$$

$$n_1+2n_2+1=n_0+n_1+n_2 \Longrightarrow n_0=n_2+1$$

3. A *full binary tree* of depth *k* is a binary tree of depth *k* having 2 -1 nodes, *k*>=0.

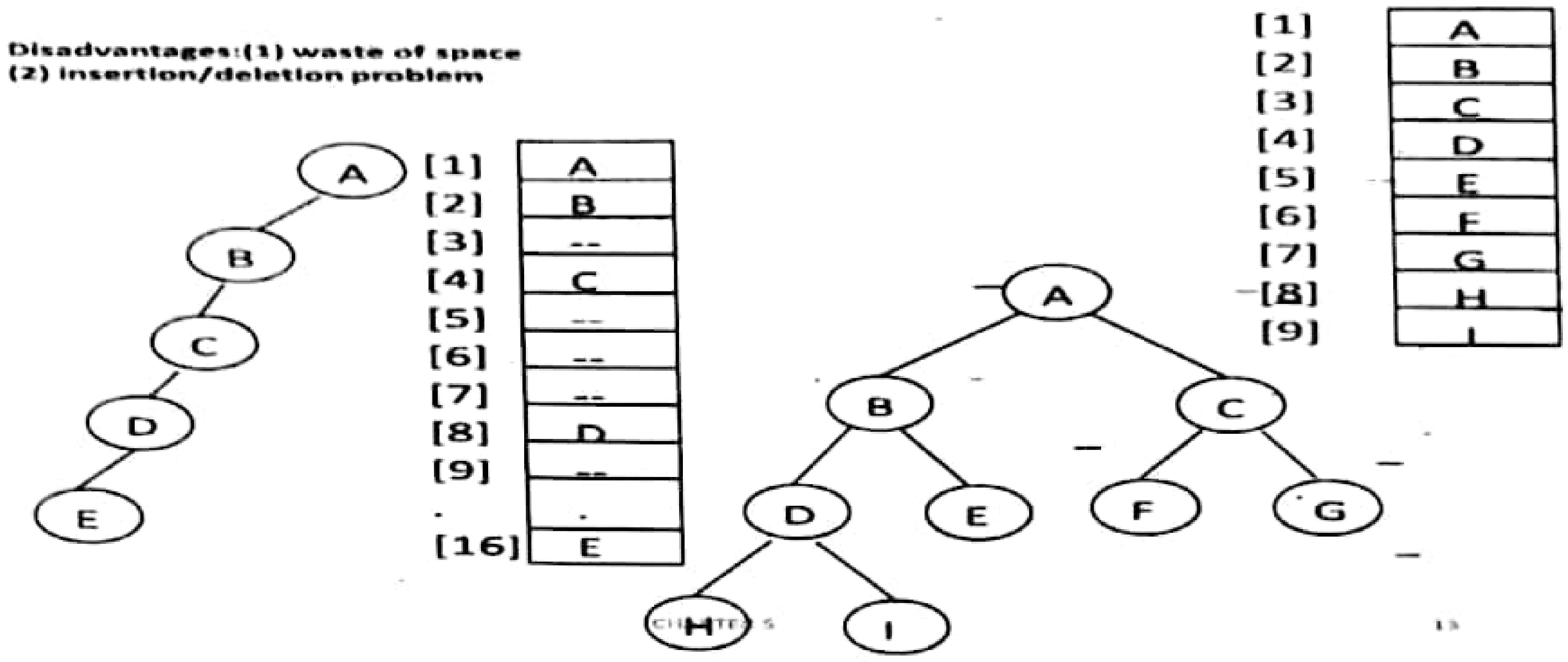
A binary tree with *n* nodes and depth *k* is *complete iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.

Binary Tree Representation

A binary tree data structure is represented using two methods. Those methods are 1)Array Representation 2)Linked List Representation

1)Array Representation: In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of

A complete binary tree with *n* nodes (depth = log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have: a) *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent. b)*left_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child. c) *right_child*(*i*) is at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.

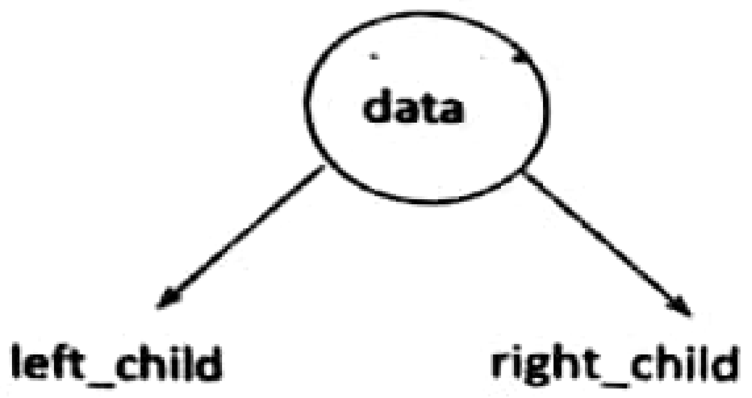


Disadvantages:(1) waste of space
(2) insertion/deletion problem

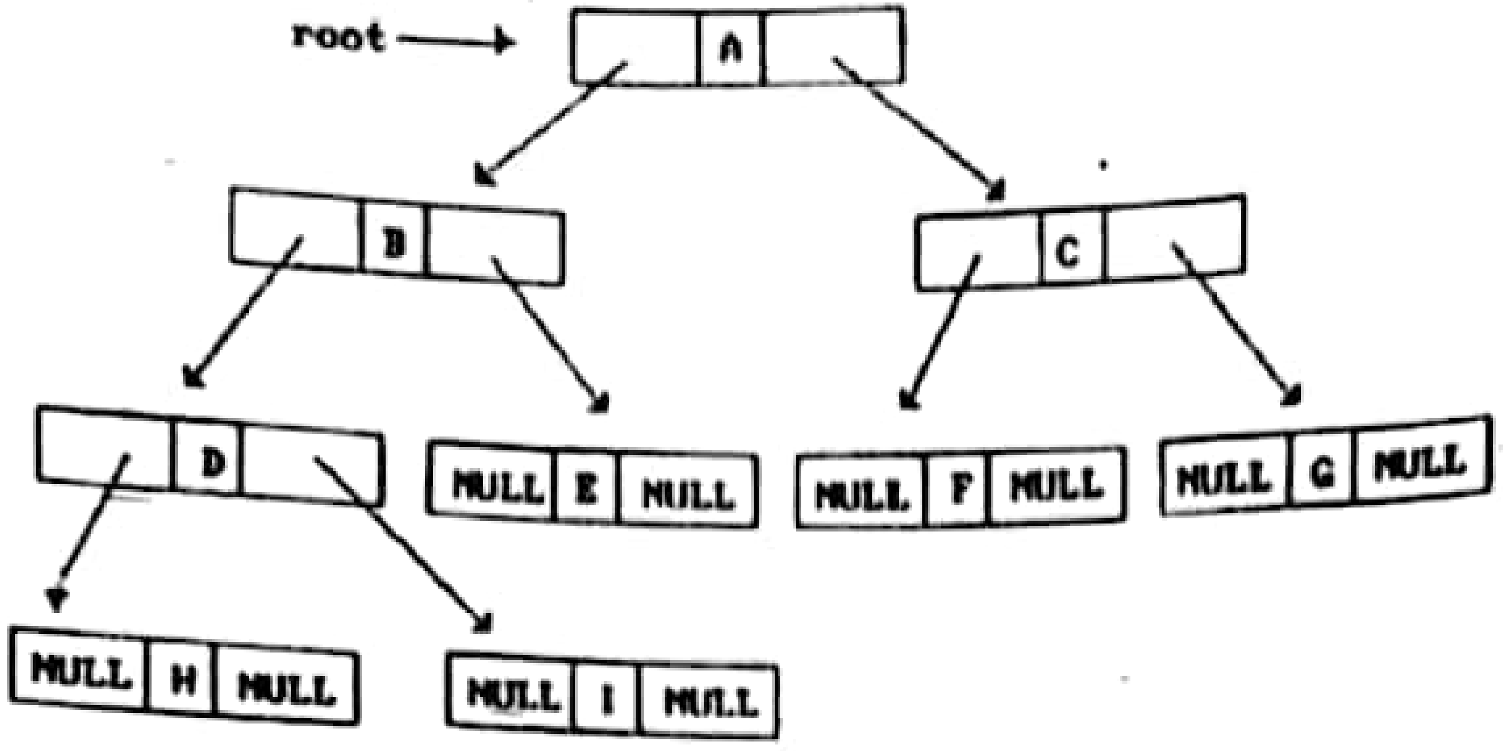
2. Linked Representation

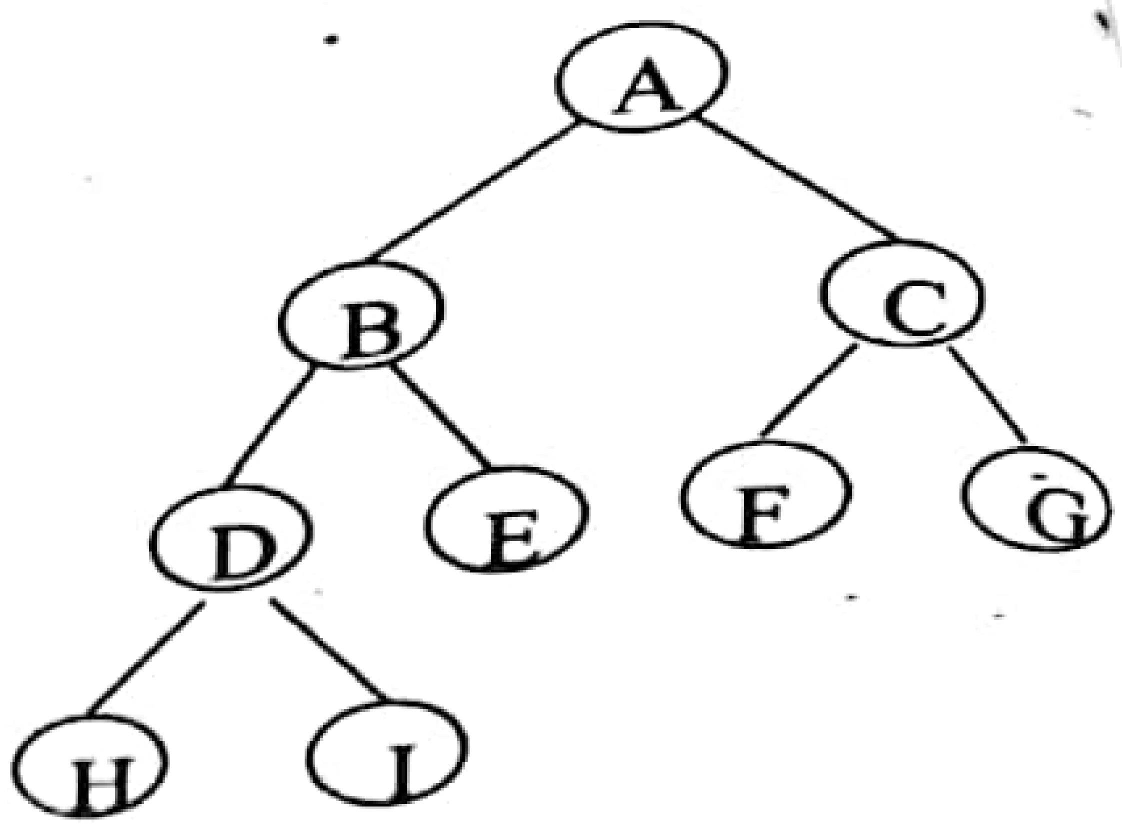
We use linked list to represent a binary tree. In a linked list, every node consists of three fields. First field, for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...

left_child	data	right_child



```
typedef struct node *tree_pointer;

typedef struct node {

 int data;

 tree_pointer left_child, right_child;};
```



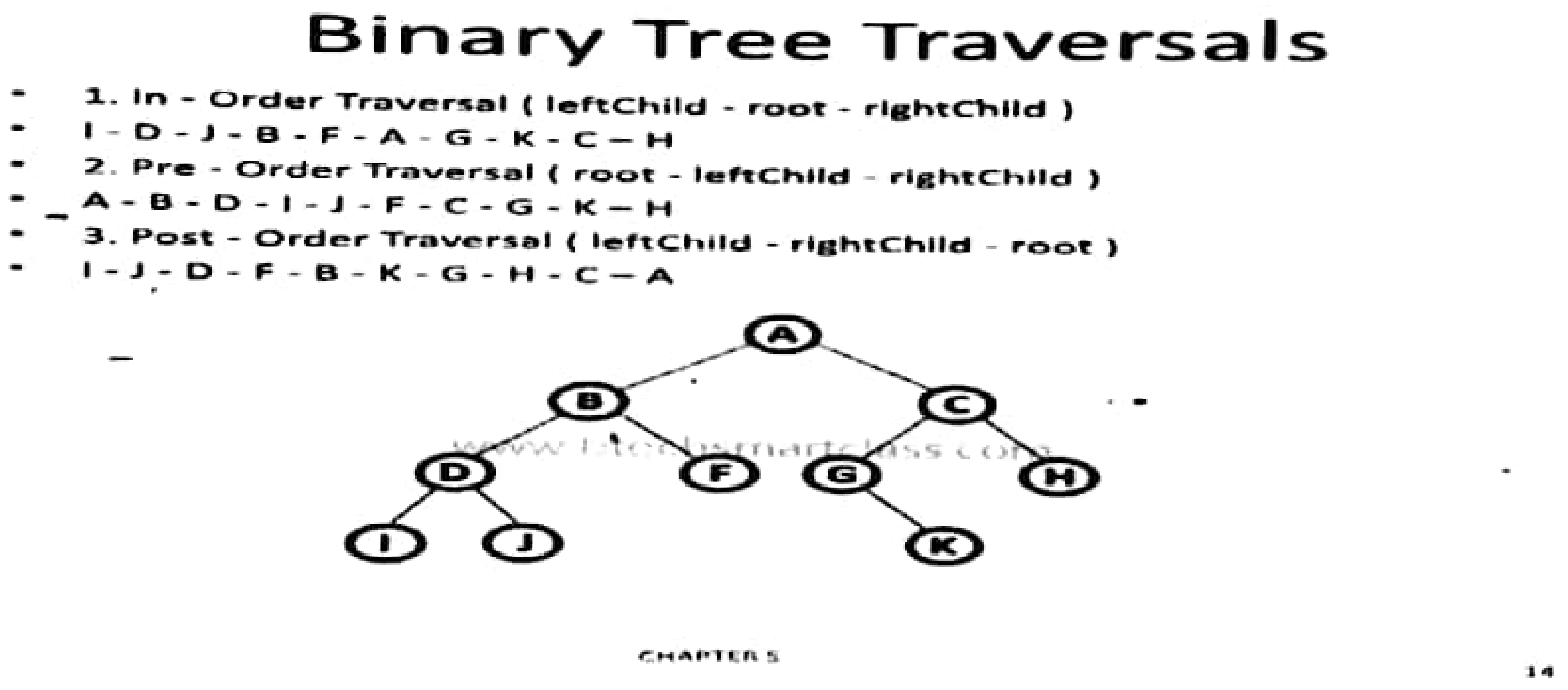


Binary Tree Traversals

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



Binary Tree Traversals

- 1. In - Order Traversal (leftChild - root - rightChild)
- I - D - J - B - F - A - G - K - C – H
- 2. Pre - Order Traversal (root - leftChild - rightChild)
- A - B - D - I - J - F - C - G - K – H
- 3. Post - Order Traversal (leftChild - rightChild - root)
- I - J - D - F - B - K - G - H - C – A

A
B
C
D
F
G
H
I
J
K

CHAPTER 5

14

1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between 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 right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child 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 left most 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 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 right most 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.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C – H

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

```
void inorder(tree_pointer ptr)        /* inorder tree traversal */  Recursive
{
   if (ptr) {
      inorder(ptr->left_child);
      printf("%d", ptr->data);
      indorder(ptr->right_child);
   }
}
```

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

In Pre-Order traversal, the root node is visited before 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 subtrees in the tree.

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 left most 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 right most 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.

Algorithm

Until all nodes are traversed –

Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

```
void preorder(tree_pointer ptr)          /* preorder tree traversal */          Recursive
{
  if (ptr) {
     printf("%d", ptr->data);
     preorder(ptr->left_child);
     preorder(ptr->right_child);
  }
}
```

3. Post - Order Traversal (leftChild - rightChild - 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 its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Algorithm

Until all nodes are traversed –

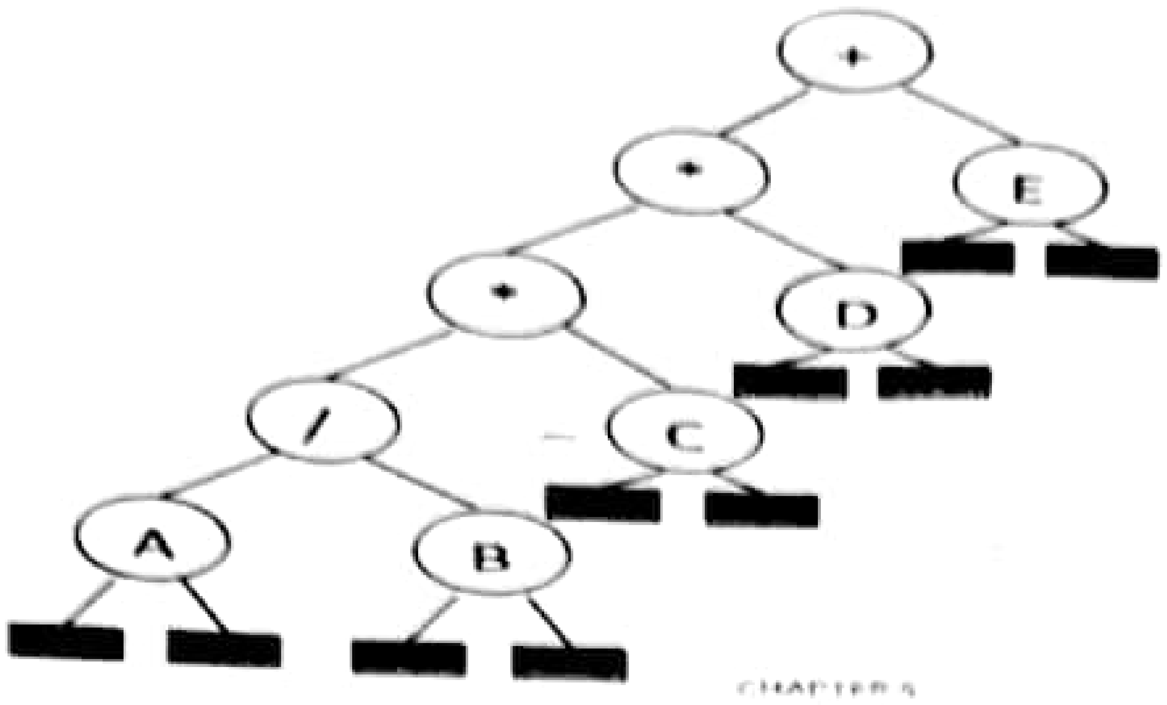
Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

```
void postorder(tree_pointer ptr)          /* postorder tree traversal */          Recursive
{
  if (ptr) {
     postorder(ptr->left_child);
     postorder(ptr->right_child);
     printf("%d", ptr->data);
  }
}
```

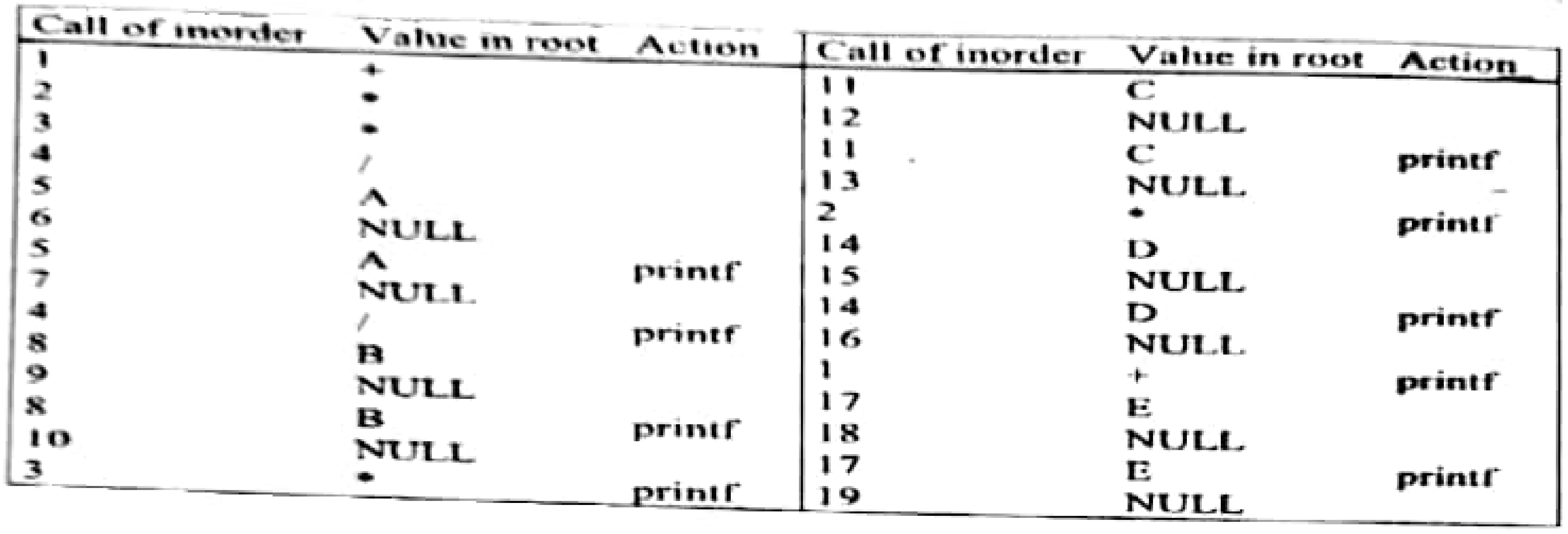
Arithmetic Expression Using BT



inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B

CHAPTER 5 15

Trace Operations of Inorder Traversal



Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

CHAPTER 5 16

Iterative Inorder Traversal (using stack)

```
void iter_inorder(tree_pointer node)
{
 int top= -1;       /* initialize stack */
 tree_pointer stack[MAX_STACK_SIZE];
 for (;;) {
 for (; node; node=node->left_child)
   add(&top, node);      /* add to stack */
 node= delete(&top);      /* delete from stack */
 if (!node) break;           /* empty stack */
 printf("%D", node->data);
 node = node->right_child;
 }
}
```

In iterative inorder traversal, we must create our own stack to add and remove nodes as in recursion.

Analysis: Let n be number of nodes in tree, every node of tree is placed on and removed from the stack exactly once. So time complexity is O(n). The space requirement is equal to the depth of tree which is O(n).

Level Order Traversal (Using Queue) – Traversal without Stack

```
void level_order(tree_pointer ptr)          /* level order tree traversal */
{
 int front = rear = 0;
 tree_pointer queue[MAX_QUEUE_SIZE];
 if (!ptr) return; /* empty queue */
 addq(front, &rear, ptr);
 for (;;) {
  ptr = deleteq(&front, rear);
if (ptr) {
   printf("%d", ptr->data);
   if (ptr->left_child)
    addq(front, &rear, ptr->left_child);
   if (ptr->right_child)
    addq(front, &rear, ptr->right_child);
 }
 else break;
}     }
```

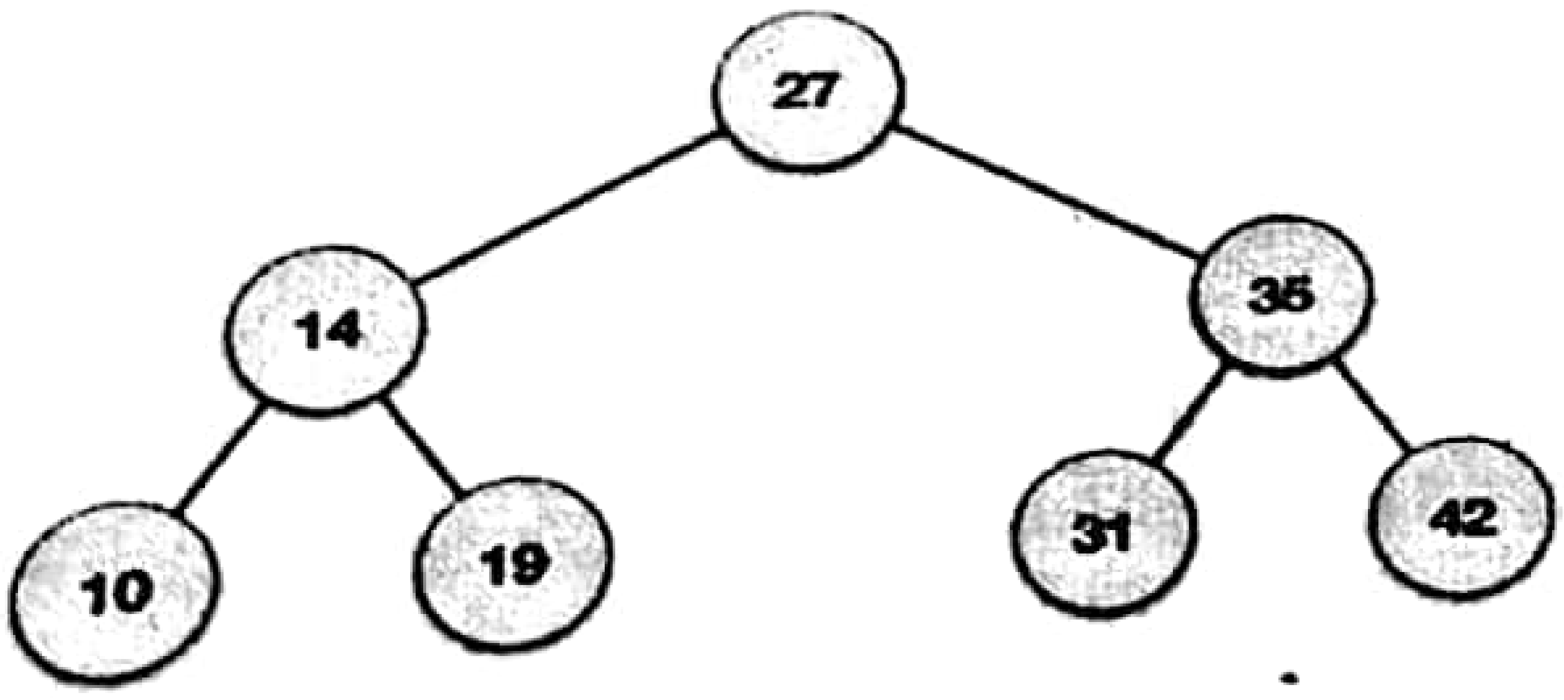
Level order Traversal is implemented with circular queue. In this order, we visit the root first, then root's left child followed by root's right child. We continue in this manner, visiting the nodes at each new level from left most to right most nodes.

We begin by adding root to the queue. The function operates by deleting the node at the front of the queue, printing out the node's data field, and adding the node's left and right children to the queue. The level order traversal for above arithmetic expression is + * E * D / C A B.

Binary Search Trees

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.



We're going to implement tree using node object and connecting them through references.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty,then all nodes follows the below mentioned properties –

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The keys in a nonempty right subtree larger than the key in the root of subtree.
- The left and right subtrees are also binary search trees.

left sub-tree and right sub-tree and can be defined as –

left_subtree (keys) $\leq$ node (key) $\leq$ right_subtree (keys)

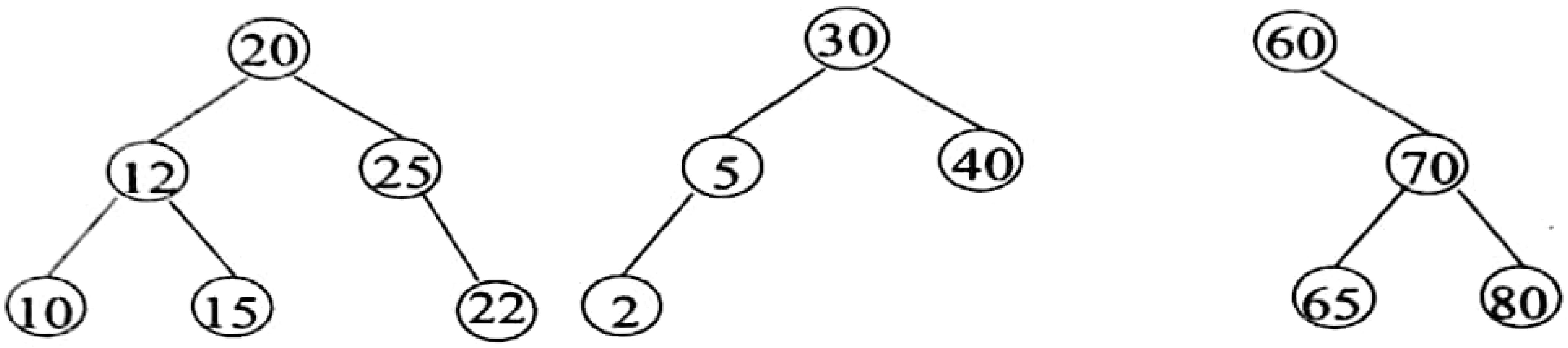


Fig: Example Binary Search Trees

ADT for Dictionary:

BST Basic Operations

The basic operations that can be performed on binary search tree data structure, are following –

- **Search** – search an element in a binary search tree.
- **Insert** – insert an element into a binary search tree / create a tree.
- **Delete** – Delete an element from a binary search tree.
- **Height** -- Height of a binary search tree.

Searching a Binary Search Tree

Let an element k is to search in binary search tree. Start search from root node of the search tree. If root is NULL, search tree contains no nodes and search unsuccessful. Otherwise, compare k with the key in the root. If k equals the root's key, terminate search, if k is less than key value, search element k in left subtree otherwise search element k in right subtree. The function search recursively searches the subtrees.

Algorithm:Recursive search of a Binary Search Tree

```
tree_pointer search(tree_pointer root,  int key)
{
/*   return   a   pointer   to   the   node   that   contains   key.   If   there   is   no   such
node, return NULL */
 if (!root) return NULL;
 if (key == root->data) return root;
 if (key < root->data)
    return search(root->left_child,  key);
 return search(root->right_child,key);
}
```

Algorithm: Iteraive search of a Binary Search Tree

```
tree_pointer search2(tree_pointer tree, int key)
{
 while (tree) {
  if (key == tree->data) return tree;
  if (key < tree->data)
     tree = tree->left_child;
  else tree = tree->right_child;
 }
return NULL;
}
```

Analysis of Recursive search and Iterative Search Algorithms:

If h is the height of the binary search tree, both algorithms perform search in O(h) time. Recursive search requires additional stack space which is O(h).

Inserting into a Binary Search Tree

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left sub tree and insert the data. Otherwise search empty location in right sub tree and insert the data.

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

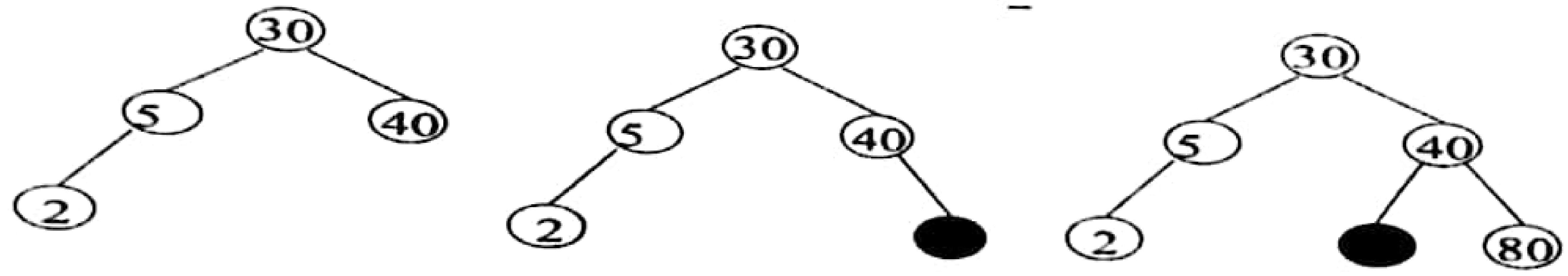
Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach a node (e.i., reach to NULL) where search terminates.

Step 7: After reaching a last node, then insert the newNode as left child if newNode is smaller or equal to that node else insert it as right child.



Insert 80

Insert 35

Algorithm

```
Create newnode
If root is NULL
  then create root node
return
If root exists then
  compare the data with node.data
  while until insertion position is located
    If data is greater than node.data
```

```
         goto right subtree
      else
         goto left subtree
   endwhile
   insert newnode
end If
```

Implementation

The implementation of insert function should look like this –

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;
   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;
   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   }else {
      current = root;
      parent  = NULL;
      while(1) {
         parent = current;
         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
```

```
        return;          }
    }

    //go to right of the tree
    else {
      current = current->rightChild;
      //insert to the right
      if(current == NULL) {
        parent->rightChild = tempNode;
        return;
      }
    }    }
  }
}
```

Deleting a node

Remove operation on binary search tree is more complicated, than insert and search. Basically, in can be divided into two stages:

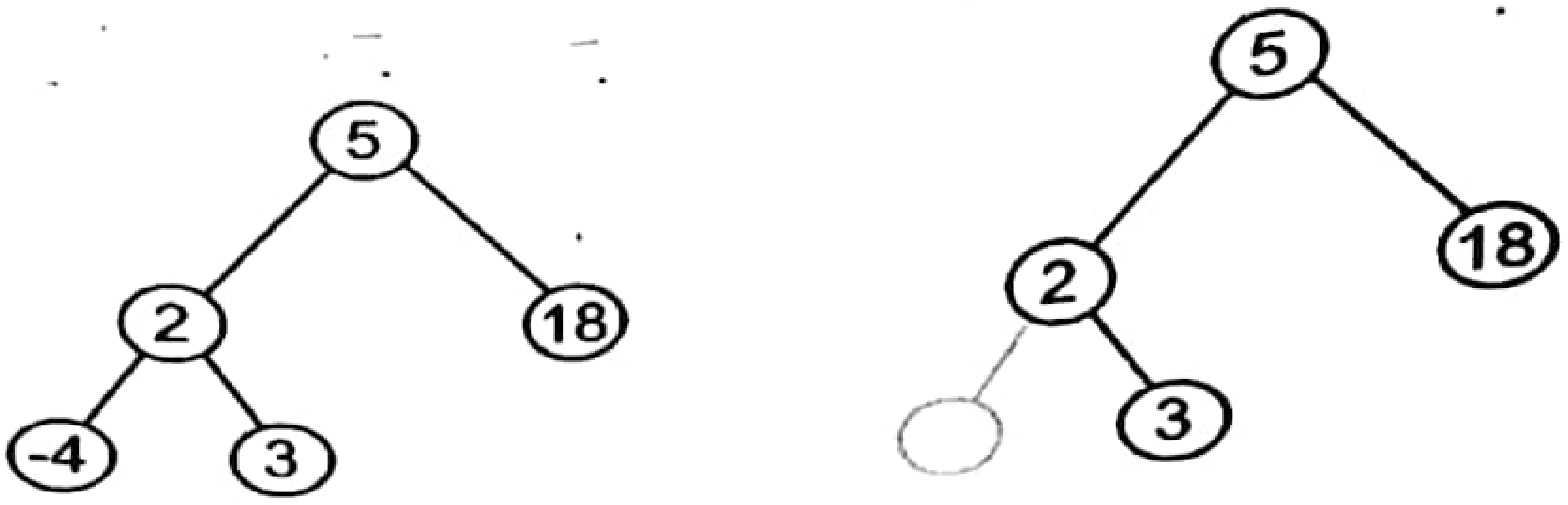
- search for a node to remove
- if the node is found, run remove algorithm.

Remove algorithm in detail

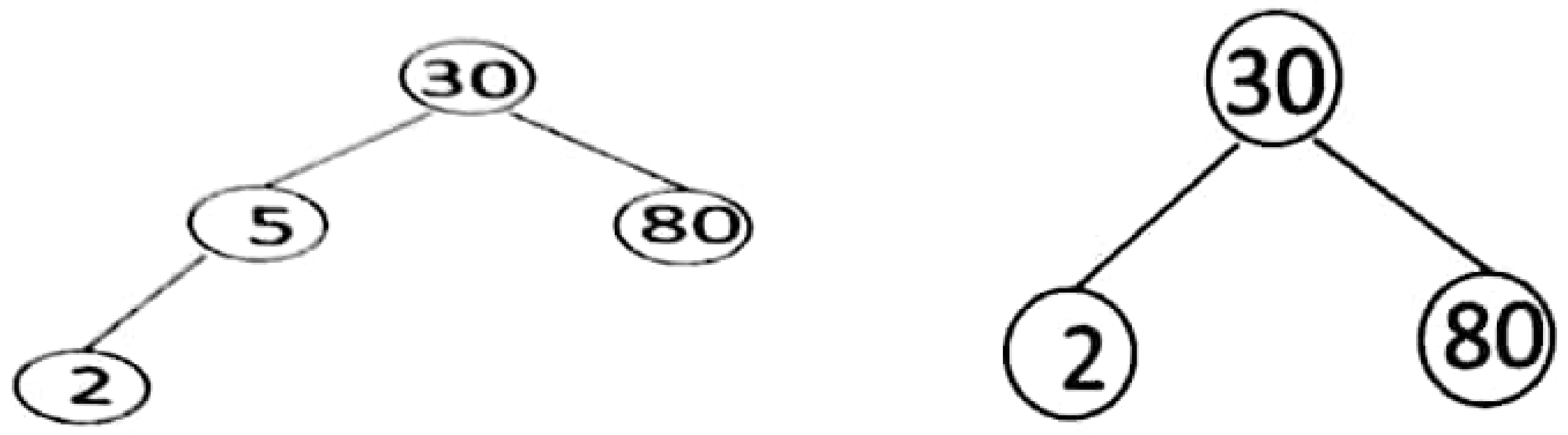
Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

1.Node to be removed has no children. --This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

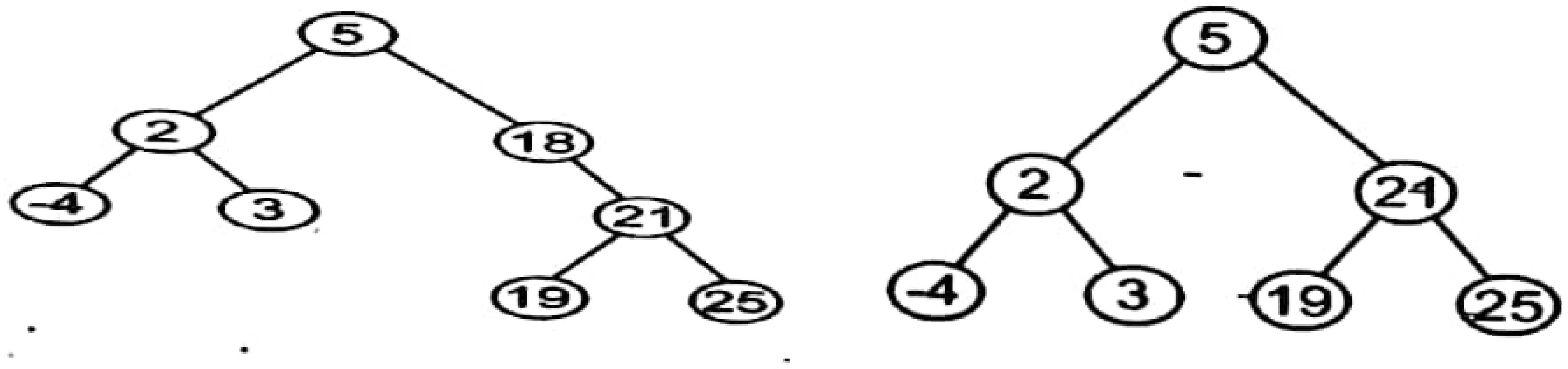
Example. Remove -4 from a BST.



2.Node to be removed has one child. In this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.



3.Node to be removed has two children. --This is the most complex case. The deleted node can be replaced by either largest key in its left subtree or the smallest in its right subtree. Preferably which node has one child.



Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

Step 7: Repeat the same process until node is deleted from the tree.

```
/* deletion in binary search tree */
 void deletion(struct treeNode **node, struct treeNode **parent, int data) {
       struct treeNode *tmpNode, *tmpParent;
       if (*node == NULL)
              return;
       if ((*node)->data == data) {
              /* deleting the leaf node */
              if (!(*node)->left && !(*node)->right) {
                     if (parent) {
                            /* delete leaf node */
```

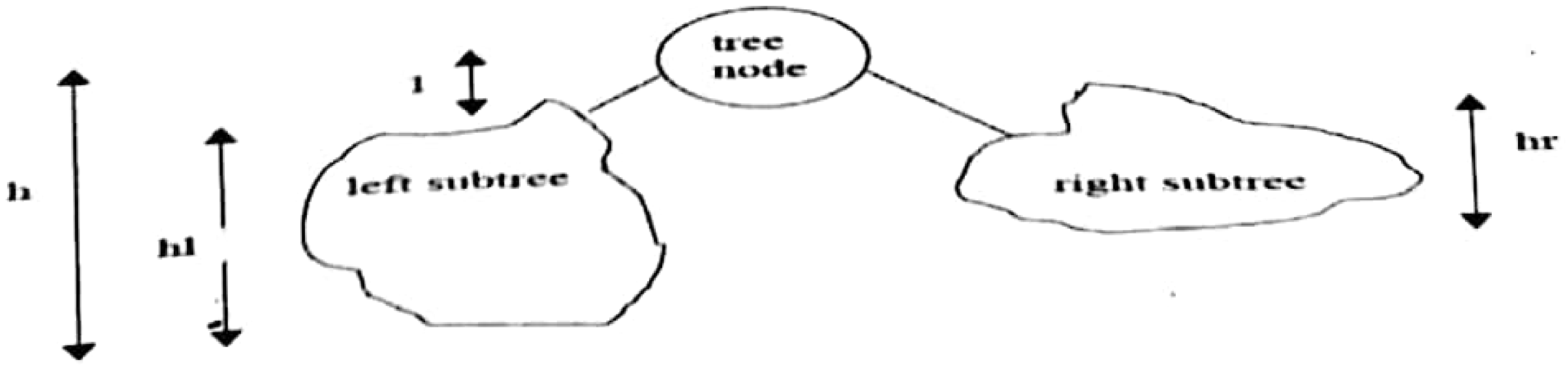
```
                if ((*parent)->left == *node)
                        (*parent)->left = NULL;
                else
                        (*parent)->right = NULL;
                free(*node);
            } else {
                /* delete root node with no children */
                free(*node);
            }
    /* deleting node with one child */
    } else if (!(*node)->right && (*node)->left) {
            /* deleting node with left child alone */
            tmpNode = *node;
            (*parent)->right = (*node)->left;
            free(tmpNode);
            *node = (*parent)->right;
    } else if ((*node)->right && !(*node)->left) {
            /* deleting node with right child alone */
            tmpNode = *node;
            (*parent)->left = (*node)->right;
            free(tmpNode);
            (*node) = (*parent)->left;
    } else if (!(*node)->right->left) {
            /*
             * deleting a node whose right child
             * is the smallest node in the right
             * subtree for the node to be deleted.
             */

            tmpNode = *node;

            (*node)->right->left = (*node)->left;

            (*parent)->left = (*node)->right;
            free(tmpNode);
            *node = (*parent)->left;
    } else {
            /*
             * Deleting a node with two children.
             * First, find the smallest node in
             * the right subtree.  Replace the
             * smallest node with the node to be
             * deleted. Then, do proper connections
             * for the children of replaced node.
             */
            tmpNode = (*node)->right;
            while (tmpNode->left) {
                tmpParent = tmpNode;
                tmpNode = tmpNode->left;
            }
            tmpParent->left = tmpNode->right;
            tmpNode->left = (*node)->left;
            tmpNode->right =(*node)->right;
            free(*node);
            *node = tmpNode;
        }
    } else if (data < (*node)->data) {
        /* traverse towards left subtree */
        deletion(&(*node)->left, node, data);
    } else if (data > (*node)->data) {
        /* traversing towards right subtree */
        deletion(&(*node)->right, node, data);
    }
}
```

Height of a Binary Search Tree:

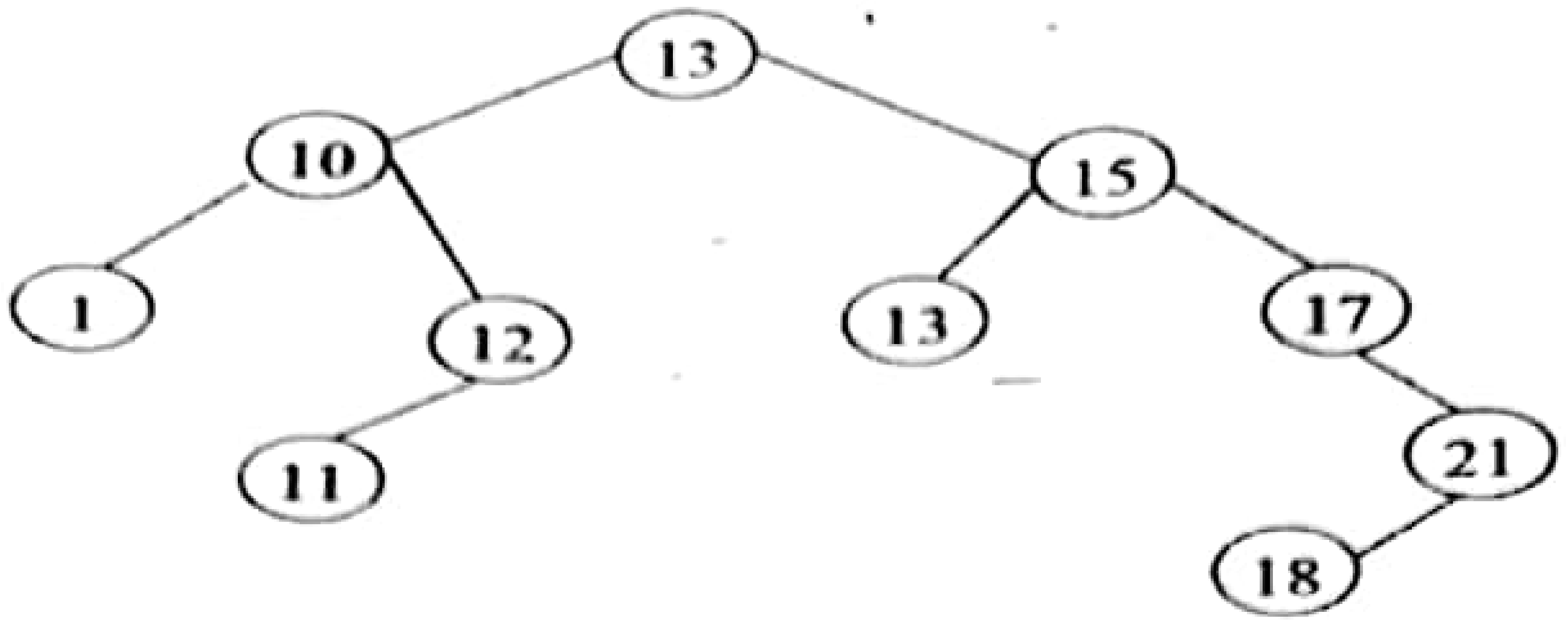
Height of a Binary Tree For a tree with just one node, the root node, the height is defined to be 0, if there are 2 levels of nodes the height is 1 and so on. A null tree (no nodes except the null node) is defined to have a height of –1.

The following height function in pseudocode is defined recursively

```
int height( BinaryTree Node t) {
    if t is a null tree
        return  -1;
    hl = height( left subtree of t);
    hr = height( right subtree of t);
    h = 1 + maximum of hl and hr;

    return h;
}
```



For example, the following tree has a height of 4. Its left subtree has height 2 and its right subtree 3.



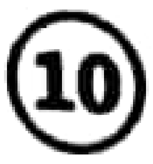
Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

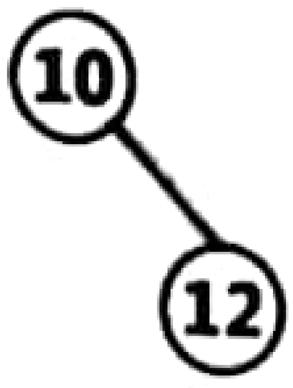
10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...

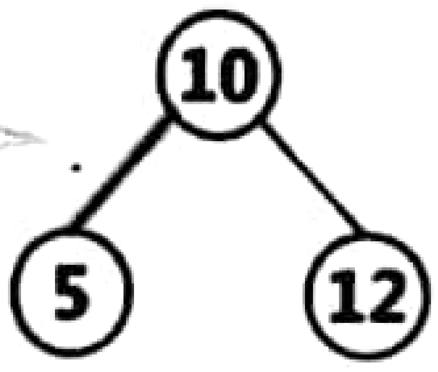
insert (10)



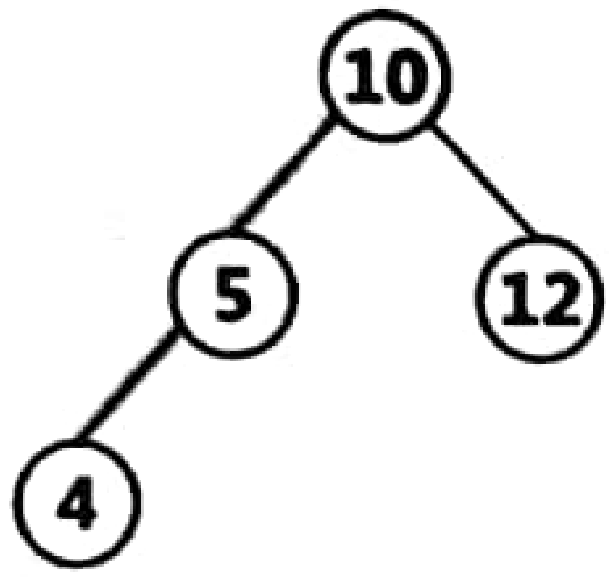
insert (12)



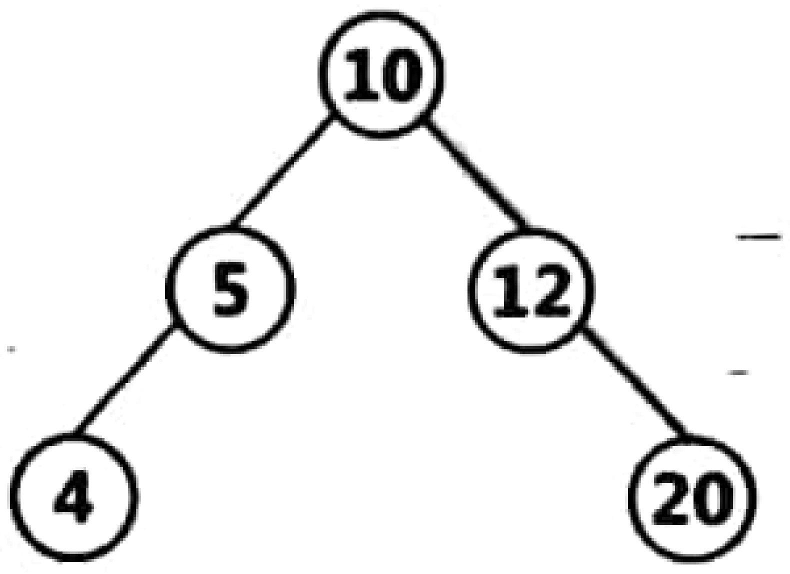
insert (5)



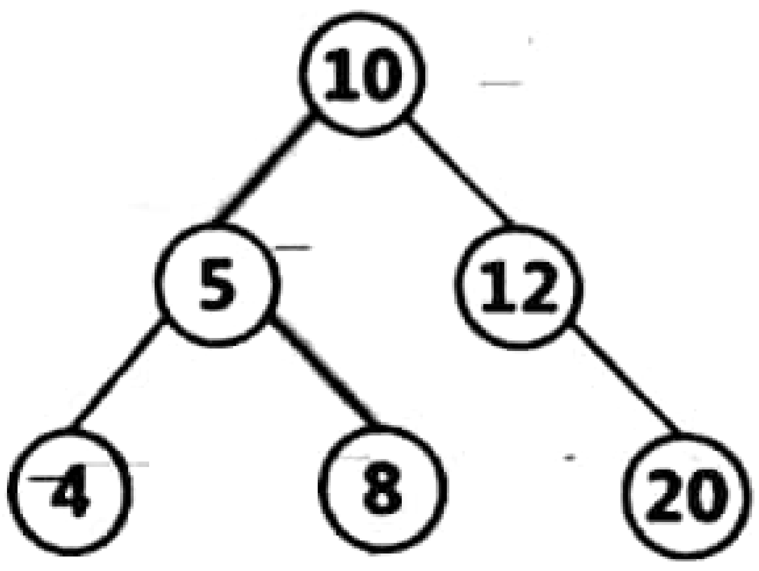
insert (4)



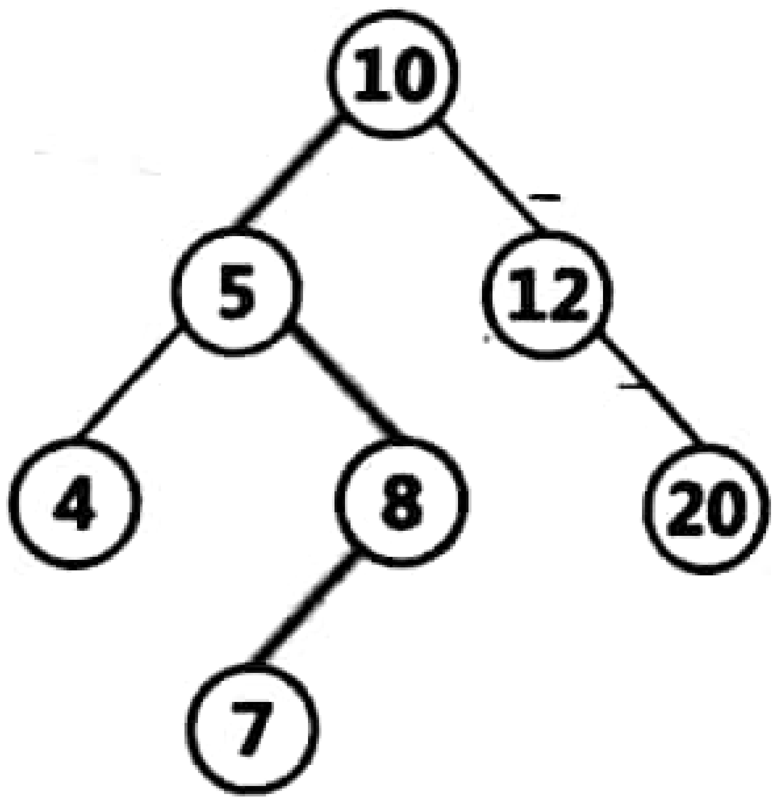
insert (20)



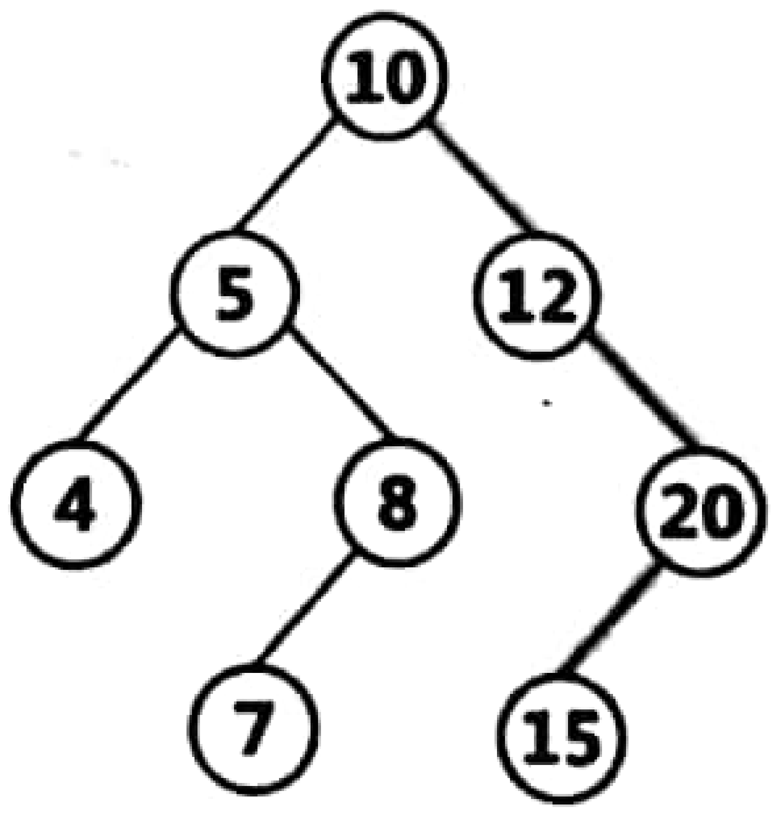
insert (8)



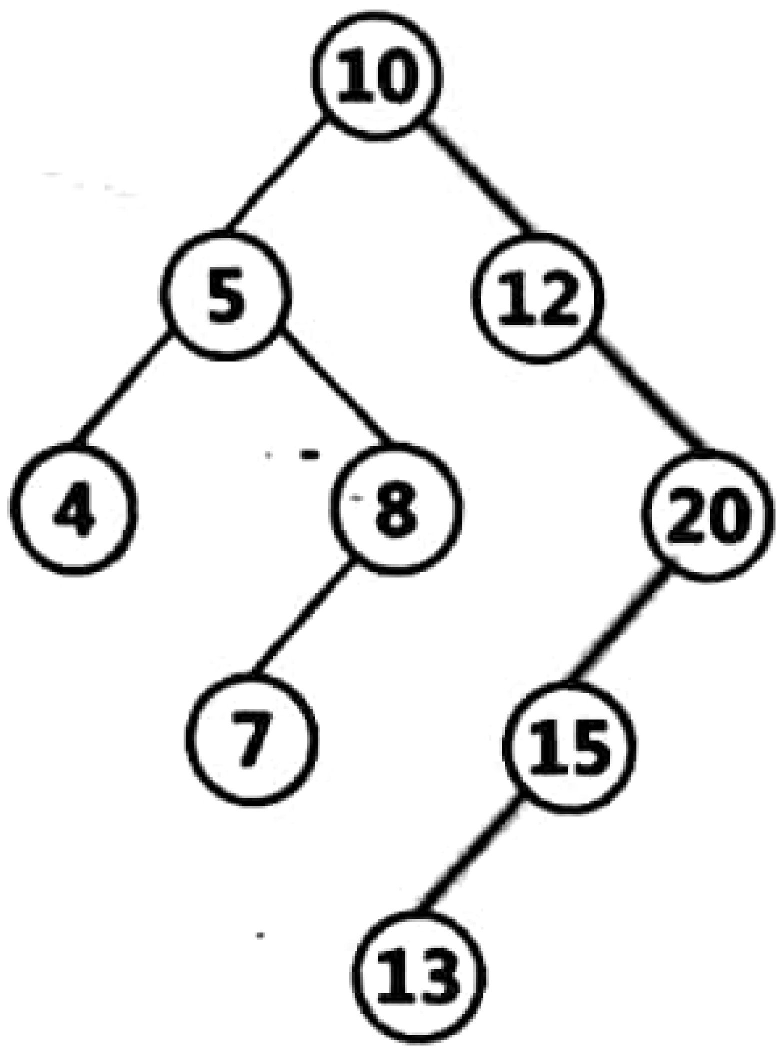
insert (7)



insert (15)



insert (13)



SCS1201_Advanced Data Structures

UNIT – II

THREADED BINARY TREE

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position. In any binary tree linked list representation, there are more number of NULL pointer than actual pointers. Generally, in any binary tree linked list representation, if there are **2N** number of reference fields, then **N+1** number of reference fields are filled with NULL (**N+1 are NULL out of 2N**). This NULL pointer does not play any role except indicating there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called **threads**.

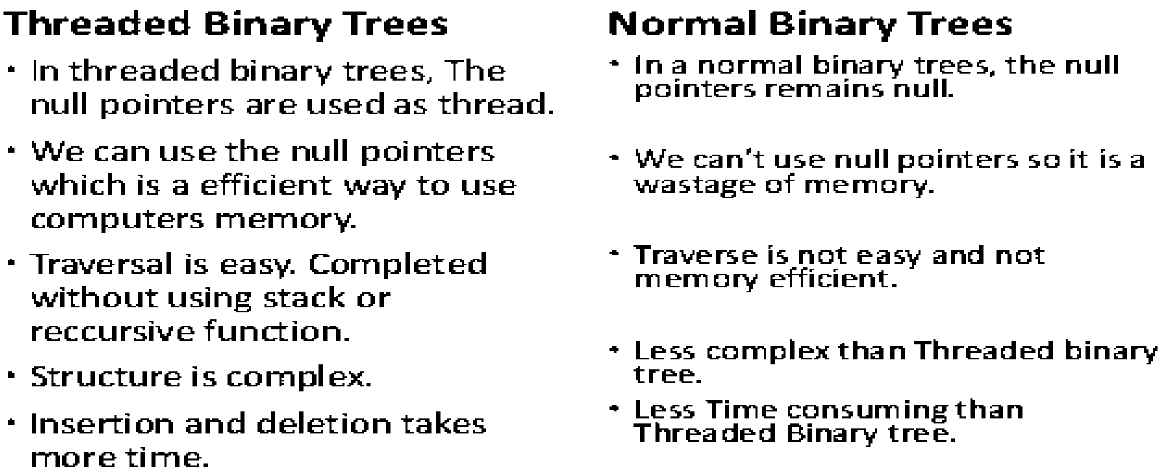
A threaded binary tree defined as follows:

"A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

Why do we need Threaded Binary Tree?

Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals. Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.

Comparison between a normal binary tree and threaded binary tree

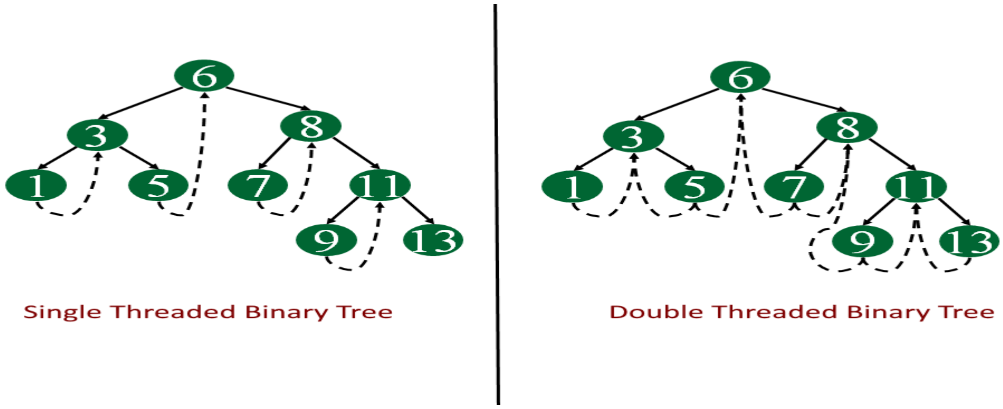


Threaded Binary Trees	Normal Binary Trees
In threaded binary trees, The null pointers are used as thread.	In a normal binary trees, the null pointers remains null.
We can use the null pointers which is a efficient way to use computers memory.	We can't use null pointers so it is a wastage of memory.
Traversal is easy. Completed without using stack or reccursive function.	Traverse is not easy and not memory efficient.
Structure is complex.	Less complex than Threaded binary tree.
Insertion and deletion takes more time.	Less Time consuming than Threaded Binary tree.

Types of threaded binary trees:

Single Threaded: each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

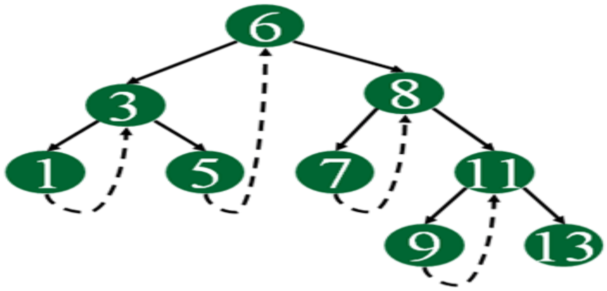
Double threaded: each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor AND all left null pointers will point to inorder predecessor.



Single Threaded Binary Tree

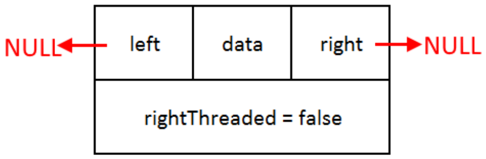
Double Threaded Binary Tree

Single Threaded: each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.



Single Threaded Binary Tree

Implementation:
Let’s see how the Node structure will look like



```
class Node{
   Node left;
   Node right;
   int data;
```

```
    boolean rightThread;
    public Node(int data){
        this.data = data;
        rightThread = false;
    }
}
```

In normal BST node we have left and right references and data but in threaded binary tree we have boolean another field called “rightThreaded”. This field will tell whether node’s right pointer is pointing to its inorder successor, but how, we will see it further.

Operations:

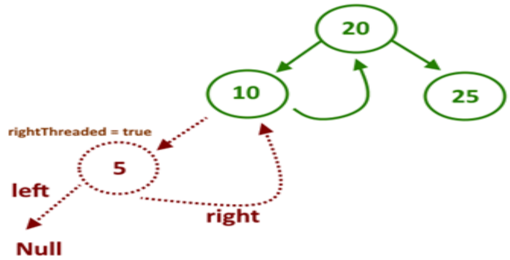
Insert node into tree

Print or traverse the tree.

Insert():

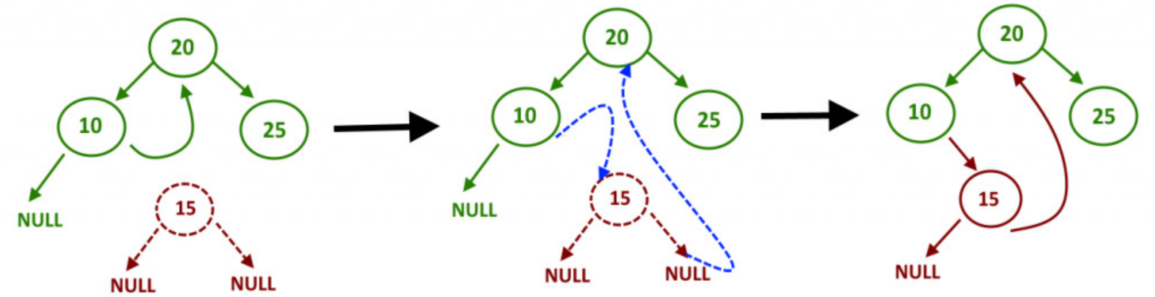
The insert operation will be quite similar to Insert operation in Binary search tree with few modifications.To insert a node our first task is to find the place to insert the node.

- Take current = root .
- Start from the current and compare root.data with n.
- Always keep track of parent node while moving left or right.
- if current.data is greater than n that means we go to the left of the root, if after moving to left, the current = null then we have found the place where we will insert the new node. Add the new node to the left of parent node and make the right pointer points to parent node and rightThread = true for new node.



Inserting node 5

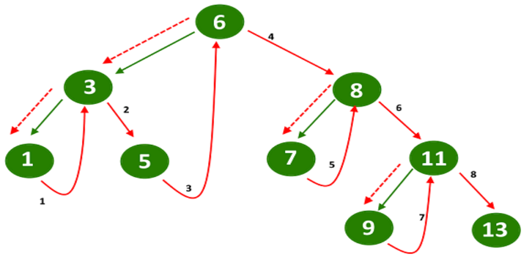
- if current.data is smaller than n that means we need to go to the right of the root, while going into the right sub tree, check rightThread for current node, means right thread is provided and points to the in order successor, if rightThread = false then and current reaches to null, just insert the new node else if rightThread = true then we need to detach the right pointer (store the reference, new node right reference will be point to it) of current node and make it point to the new node and make the right reference point to stored reference.



Inserting Node 15 into threaded binary tree

Traverse():
Traversing the threaded binary tree will be quite easy, no need of any recursion or any stack for storing the node. Just go to the left most node and start traversing the tree using right pointer and whenever rightThread = false again go to the left most node in right sub-tree.

Traversal of Single threaded binary tree



Output : 1 3 5 6 7 8 9 11 13

Follow the red arrow, dotted arrow when moving to left most node from the current node and solid arrow when using the right pointer to move it to it's inorder successor.

```
Node leftMost(Node n) {
   Node ans = n;
   if (ans == null) {
       return null;
   }
   while (ans.left != null) {
      ans = ans.left;
   }
   return ans;
}
void inOrder(Node n) {
   Node cur = leftmost(n);
   while (cur != null) {
      print(cur);
      if (cur.rightThread) {
         cur = cur.right;
      } else {
         cur = leftmost(cur.right);
      }
   }
}
```

HEIGHT BALANCED TREES (AVL TREES)

The Height balanced trees were developed by researchers Adelson-Velskii and Landis. Hence these trees are also called AVL trees. Height balancing attempts to maintain the balance factor of the nodes within limit.

Height of the tree: Height of a tree is the number of nodes visited in traversing a branch that leads to a leaf node at the deepest level of the tree.

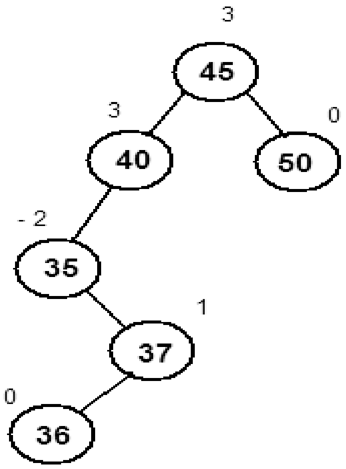
Balance factor: The balance factor of a node is defined to be the difference between the height of the node's left subtree and the height of the node's right subtree.

Consider the following tree. The left height of the tree is 5, because there are 5 nodes (45, 40, 35, 37 and 36) visited in traversing the branch that leads to a leaf node at the deepest level of this tree.

Balance factor = height of left subtree – height of the right subtree

In the following tree the balance factor for each and every node is calculated and shown. For example, the balance factor of node 35 is (0 – 2) = - 2.

The tree which is shown below is a binary search tree. The purpose of going for a binary search tree is to make the searching efficient. But when the elements are added to the binary search tree in such a way that one side of the tree becomes heavier, then the searching becomes inefficient. The very purpose of going for a binary search tree is not served. Hence we try to adjust this unbalanced tree to have nodes equally distributed on both sides. This is achieved by rotating the tree using standard algorithms called the AVL rotations. After applying AVL rotation, the tree becomes balanced and is called the AVL tree or the height balanced tree.

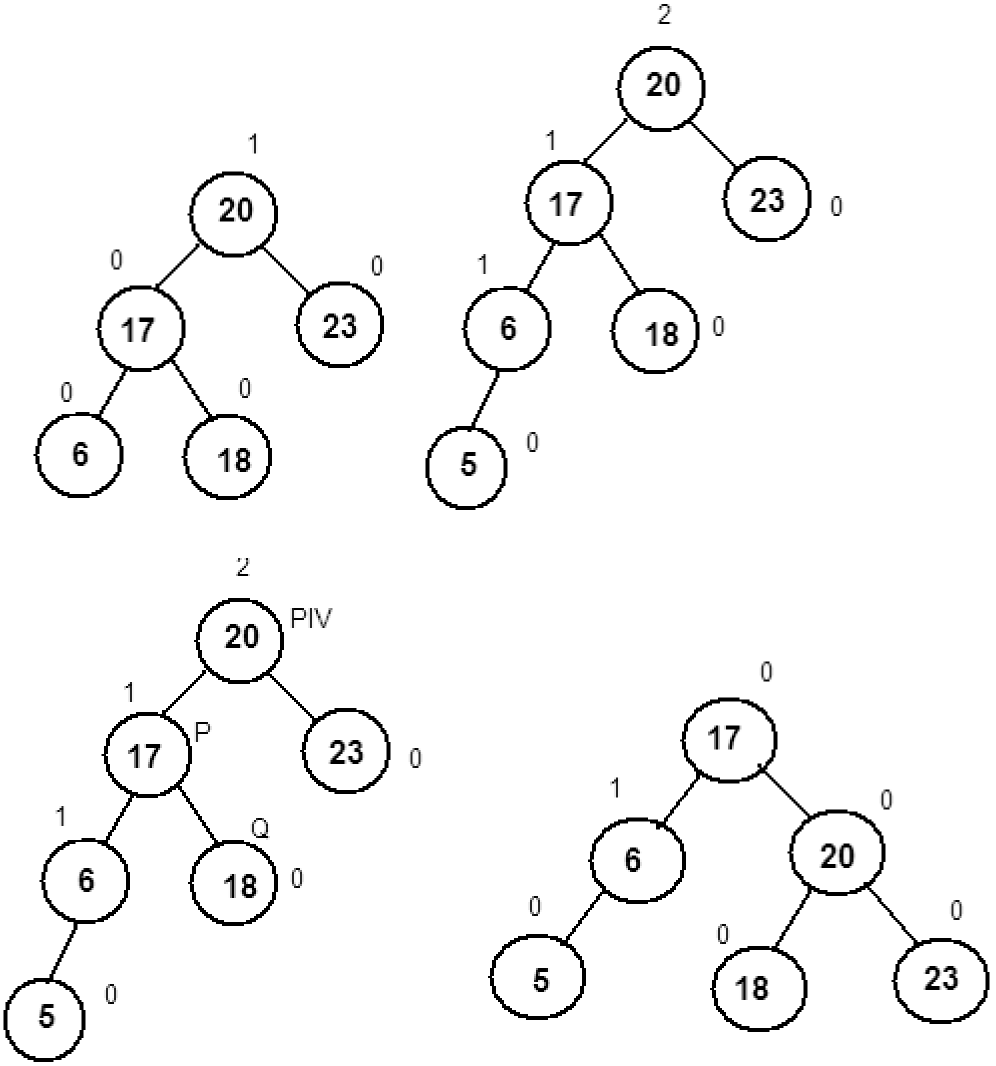


The tree is said to be balanced if each node consists of a balance factor either -1 or 0 or 1. If even one node has a balance factor deviated from these values, then the tree is said to be unbalanced. There are four types of rotations. They are:

1. *Left-of-Left rotation.*
2. *Right-of-Right rotation.*
3. *Right-of-Left rotation.*
4. *Left-of-Right rotation.*

Left-of-Left Rotation

Consider the following tree. Initially the tree is balanced. Now a new node 5 is added. This addition of the new node makes the tree unbalanced as the root node has a balance factor 2. Since this is the node which is disturbing the balance, it is called the pivot node for our rotation. It is observed that the new node was added as the left child to the left subtree of the pivot node. The pointers P and Q are created and made to point to the proper nodes as described by the algorithm. Then the next two steps rotate the tree. The last two steps in the algorithm calculates the new balance factors for the nodes and is seen that the tree has become a balanced tree.

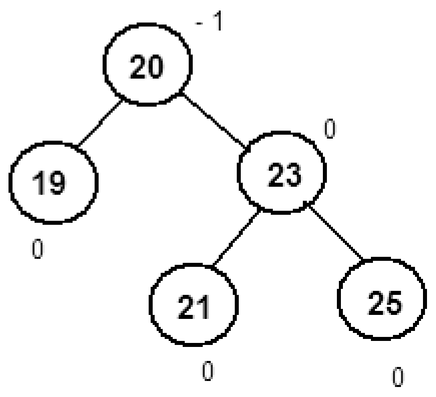


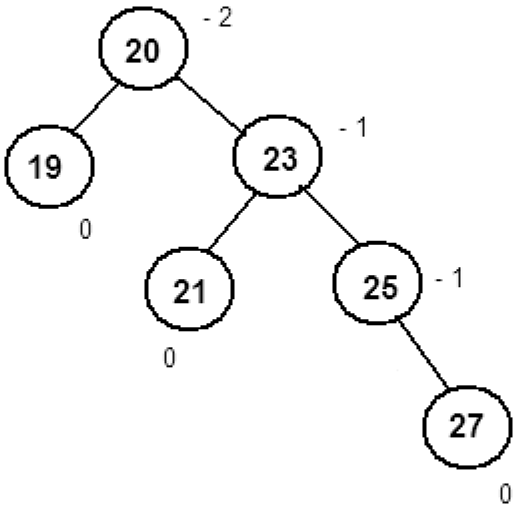
Algorithm

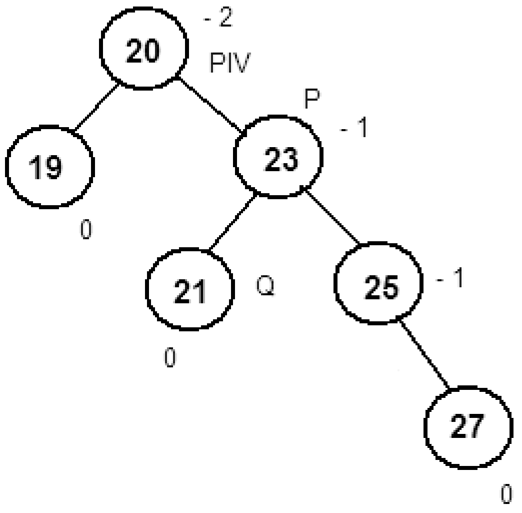
```
LEFT-OF-LEFT(pivot)
P = left(pivot)
Q = right(P)
Root = P
Right(P) = pivot
Left(pivot) = Q
Bal(pivot) = 0
Bal(right(pivot)) = 0
End LEFT-OF-LEFT
```

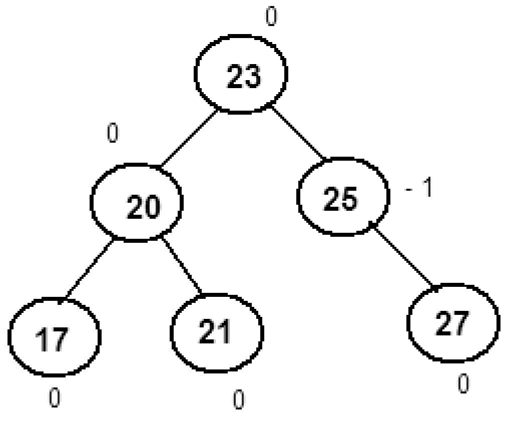
Right-of- Right Rotation

In this case, the pivot element is fixed as before. The new node is found to be added as the right child to the right subtree of the pivot element. The first two steps in the algorithm sets the pointer P and Q to the correct positions. The next two steps rotate the tree to balance it. The last two steps calculate the new balance factor of the nodes.







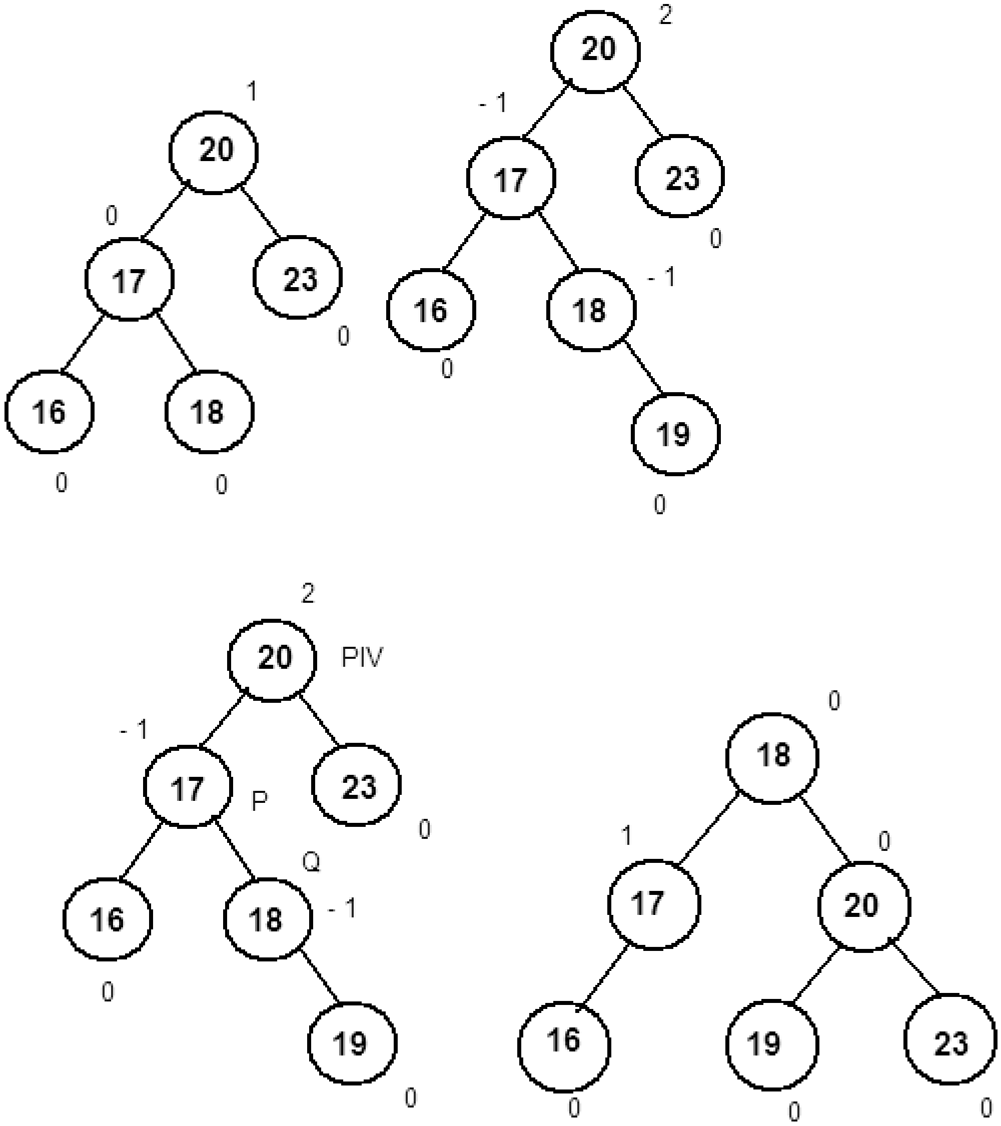


Algorithm

```
RIGHT-OF-RIGHT(pivot)
P = right(pivot)
Q = left(P)
Root = P
Left(P) = pivot
Right(pivot) = Q
Bal(pivot) = 0
Bal(left(pivot)) = 0
End RIGHT-OF-RIGHT
```

Right-of-Left Rotation

In this following tree, a new node 19 is added. This is added as the right child to the left subtree of the pivot node. The node 20 fixed as the pivot node, as it disturbs the balance of the tree. In the first two steps the pointers P and Q are positioned. In the next four steps, tree is rotated. In the remaining steps, the new balance factors are calculated.

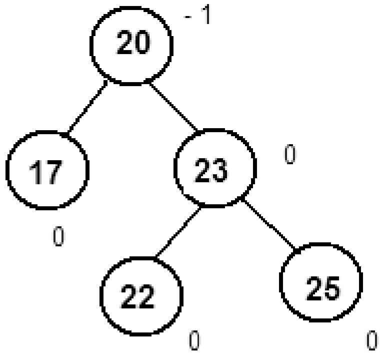


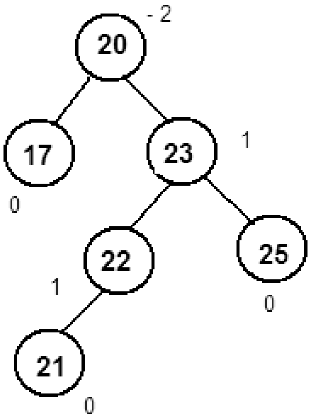
Algorithm

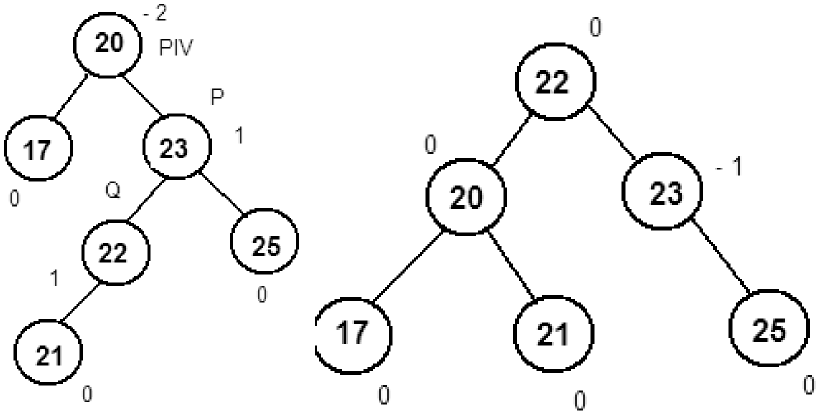
```
RIGHT-OF-LEFT(pivot)
P = left(pivot)
Q = right(P)
Root = Q
Left(Q) = P
Right(Q) = Pivot
Left(pivot) = right(Q)
Right(P) = left(Q)
If Bal(pivot) = 0
    Bal(left(pivot)) = 0
    Bal(right(pivot)) = 0
Else
    If Bal(pivot) = 1
        Bal(pivot) = 0
        Bal(left(pivot)) = 0
        Bal(right(pivot)) = -1
    Else
        Bal(pivot) = 0
        Bal(left(pivot)) = 1
        Bal(right(pivot)) = 0
    End if
End if

End RIGHT-OF-LEFT
```

Left-of-Right

In the following tree, a new node 21 is added. The tree becomes unbalanced and the node 20 is the node which has a deviated balance factor and hence fixed as the pivot node. In the first two steps of the algorithm, the pointers P and Q are positioned. In the next 4 steps the tree is rotated to make it balanced. The remaining steps calculate the new balance factors for the nodes in the tree.







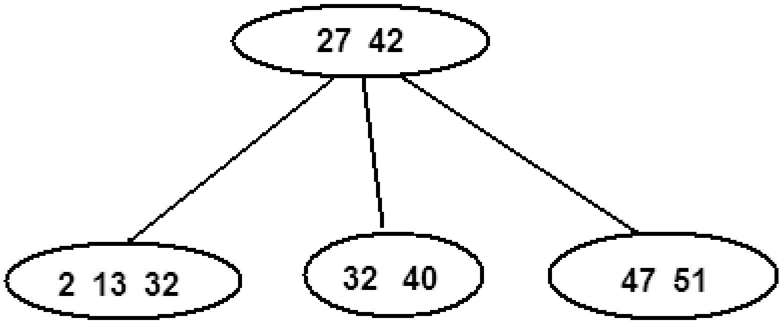
Algorithm

```
LEFT-OF-RIGHT(pivot)

P = right(pivot)
Q = left(P)
Root= Q
Right(Q) = P
Left(Q) = Pivot
Right(pivot) = left(Q)
Left(P) = right(Q)
If Bal(pivot) = 0
        Bal(right(pivot)) = 0
        Bal(left(pivot)) = 0
Else
        If Bal(pivot) = 1
                Bal(pivot) = 0
                Bal(right(pivot)) = 0
                Bal(left(pivot)) = -1
        Else
                Bal(pivot) = 0
                Bal(right(pivot)) = 1
                Bal(left(pivot)) = 0
        End if
End if
End LEFT-OF-RIGHT
```

B – TREES

Multiway search tree (m-way search tree): *Multiway search tree of order n is a tree in which any node may contain maximum n-1 values and can have maximum n children.*

Consider the following tree. Every node in the tree has one or more than one values stored in it. The tree shown is of order 3. Hence this tree can have maximum 3 children and each node can have maximum 2 values. Hence it is an m-way search tree.



B – Tree: *B – tree is a m-way search tree of order n that satisfies the following conditions.*

(i) All non-leaf nodes (except root node) have at least n/2 children and maximum n children.

(ii) The non-leaf root node may have at least 2 children and maximum n children.

(iii) B-Tree can exist with only one node. i.e. the root node containing no child.

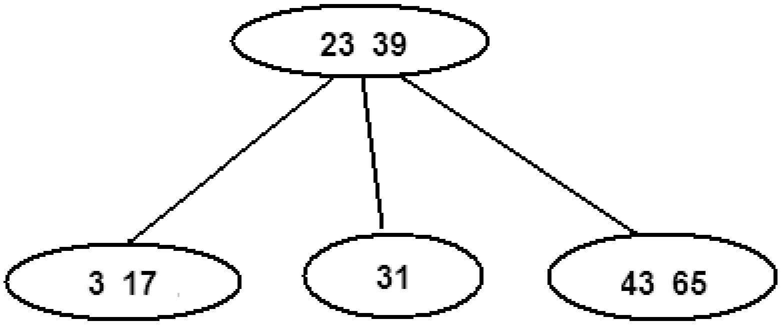
(iv) If a node has n children then it must have n-1 values.

(v) All the values that appear on the left most child of a node are smaller than the first value of that node. All values that appears on the right most child of a node are greater that the last values of that node.

(vi) If x and y are any two i^{th} and $(i+1)^{th}$ values of a node, where x < y, then all the values appearing on the $(i+1)^{th}$ sub-tree of that node are greater than x and less than y.

(vii) All the leaf nodes should appear on the same level. All the nodes except root node should have minimum n/2 values.

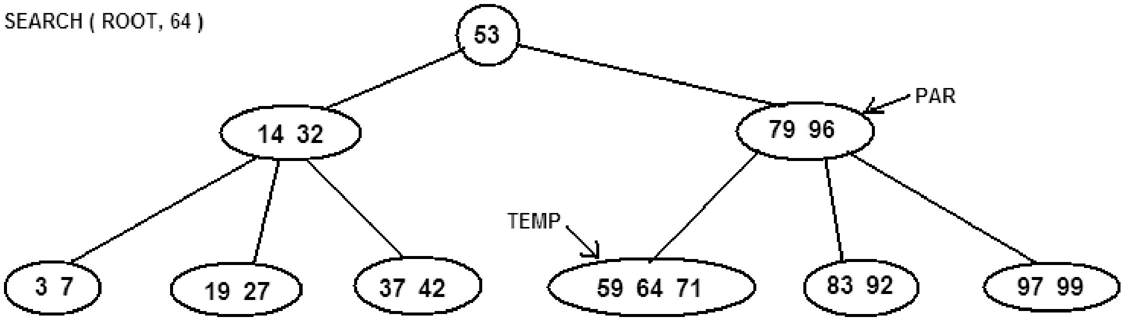
Consider the following tree. Clearly, it is a m-way search tree of order 3. Let us check whether the above conditions are satisfied. It can be seen that root node has 3 children and therefore has only 2 values stored in it. Also it is seen that the elements in the first child (3, 17) are lesser than the value of the first element (23) of the root node. The value of the elements in the second child (31) is greater than the value of the first element of the root node (23) and less than the value of the second element (39) in the root node. The value of the elements in the rightmost child (43, 65) is greater than the value of the rightmost element in the root node. All the three leaf nodes are at the same level (level 2). Hence all the conditions specified above is found to be satisfied by the given m-way search tree. Therefore it is a B-Tee.



Search Operation in a B-Tree

Let us say the number to be searched is k = 64. A temporary pointer temp is made to initially point to the root node. The value k = 64 is now compared with each element in the node pointed by temp. If the value is found then the address of the node where it is found is returned using the temp pointer. If the value k is greater than i^{th} element of the node, then the temp is moved to the $i+1^{th}$ node and the search process is repeated. If the k value is lesser than the first value in the node, then the temp is moved to the first child. If the k value is greater than the last value of the node, then temp is moved to the rightmost child of the node and the search process is repeated.

After the particular node where the value is found is located (now pointed by temp), then a variable LOC is initialized to 0, indicating the position of the value to be searched within that node. The value k is compared with each and every element of the node. When the value of the k is found within the node, then the search comes to an end position where it is found is stored in LOC. If not found the value of LOC is zero indicating that the value is not found.



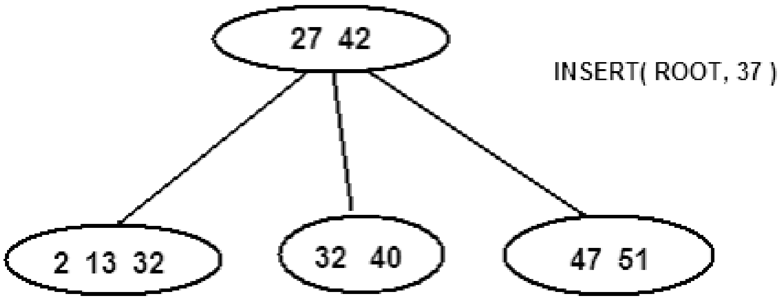
Algorithm

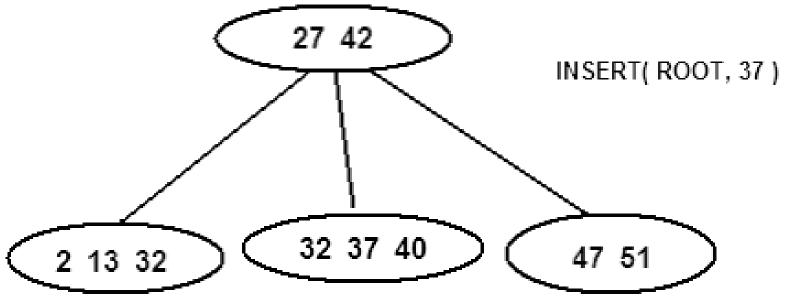
```
SEARCH( ROOT, k )

Temp = ROOT, i = 1, pos = 0
While i ≤ count(temp) and child[i](temp) ≠ NULL
        If k = info(temp[i])
                Pos = i
                Return temp
        Else
                If k < info(temp[i])
                        SEARCH(child[i](temp), k)
                Else
                        If i = count(temp)
                                Par = temp
                                Temp = child[i+1](temp)
                        Else
                                i = i + 1
                        End if
                End if
        End if
End While

While i ≤ count(temp)
    If k = info(temp[i])
        Pos = i
        Return temp
    End if
End while

End SEARCH
```

Insert Operation in a B-Tree

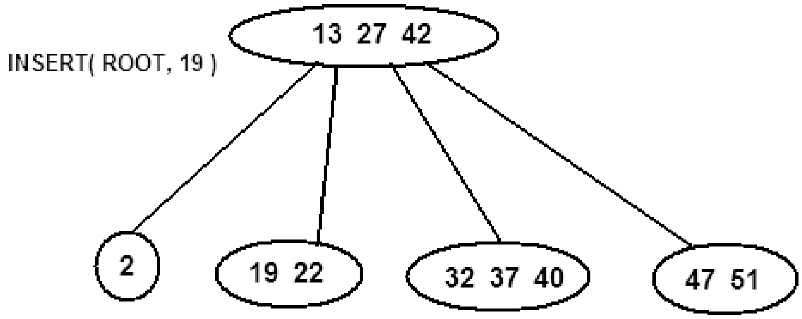
One of the conditions in the B-Tree is that, the maximum number of values that can be present in the node of a tree is n – 1, where n is the order of the tree. Hence, it should be taken care that, even after insertion, this condition is satisfied. There are two cases: In the first case, the element is inserted into a node which already had less than n- 1 values, and the in the second case, the element is inserted into a node which already had exactly n-1 values. The first case is a simple one. The insertion into the node does not violate any condition of the tree. But in the second case, if the insertion is done, then after insertion, the number values exceeds the limit in that node.

Let us take the first case. In both the cases, the insertion is done by searching for that element in the tree which will give the node where it is to be inserted. While searching, if the value is already found, then no insertion is done as B-Tree is used for storing the key values and keys do not have duplicates. Now the value given is inserted into the node. Consider the figure which shows how value 37 is inserted into correct place.





In the second case, insertion is done as explained above. But now, it is found that, the number of values in the node after insertion exceeds the maximum limit. Consider the same tree shown above. Let us insert a value 19 into it. After insertion of the value 19, the number of values (2, 13, 19, 22) in that node has become 4. But it is a B-Tree of order 4 in which there should be only maximum 3 values per node. Hence the node is split into two nodes, the first node containing the numbers starting from the first value to the value just before the middle value (first node: 2). The second node will contain the numbers starting just after the mid value till the last value (second node: 19, 22). The mid value 13 is pushed into the parent. Now the adjusted B-Tree appears as shown.



Algorithm

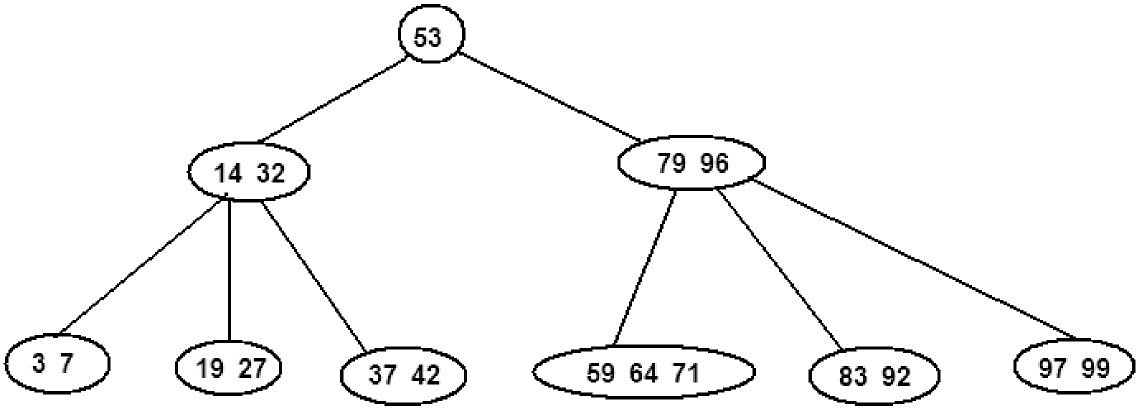
```
INSERT( ROOT, k )

Temp = SEARCH( ROOT, k )
If count(temp) < n-1
        Ins(temp, k)
        Return
Else
        Repeat for i = n/2 +1 to n-1
                Info(R[i-n/2]) = info(temp[i])
                Count(R) = count(R) + 1
        End repeat
        Count(temp) = n/2 – 1
        Ins(par, info(temp[m/2])
End if
INSERT( ROOT, k )

End INSERT
```

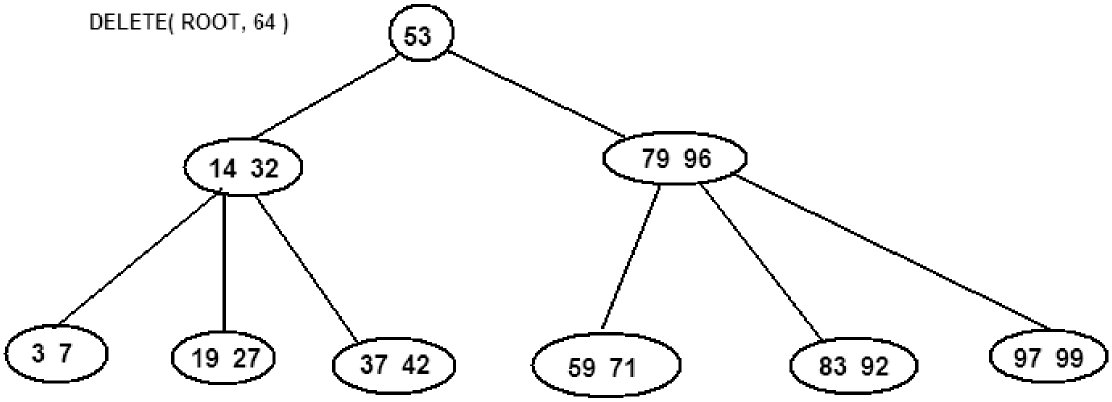
Delete Operation in B-Tree

When the delete operation is performed, we should take care that even after deletion, the node has minimum n/2 value in it, where n is the order of the tree.

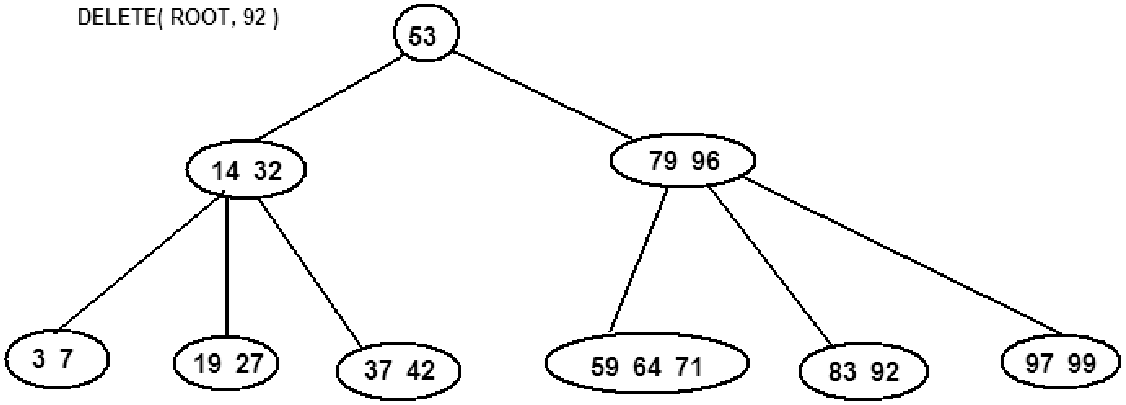
There are three cases:

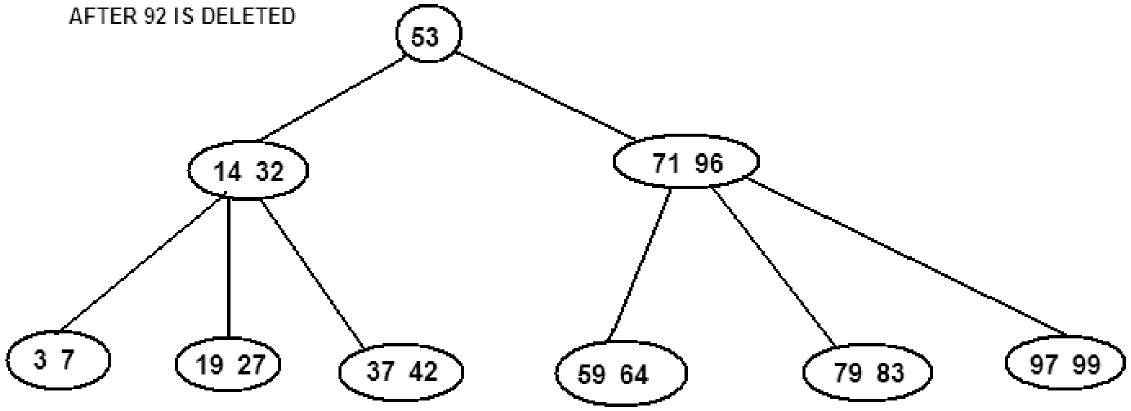
Case 1: The node from which the value is deleted has minimum n/2 values even after deletion. Let us consider the following B-Tree of order 5. A value 64 is to be deleted. Even after the deletion of the value 64, the node has minimum n/2 values (i.e., 2 values). Hence the rules of the B-Tree are not violated.



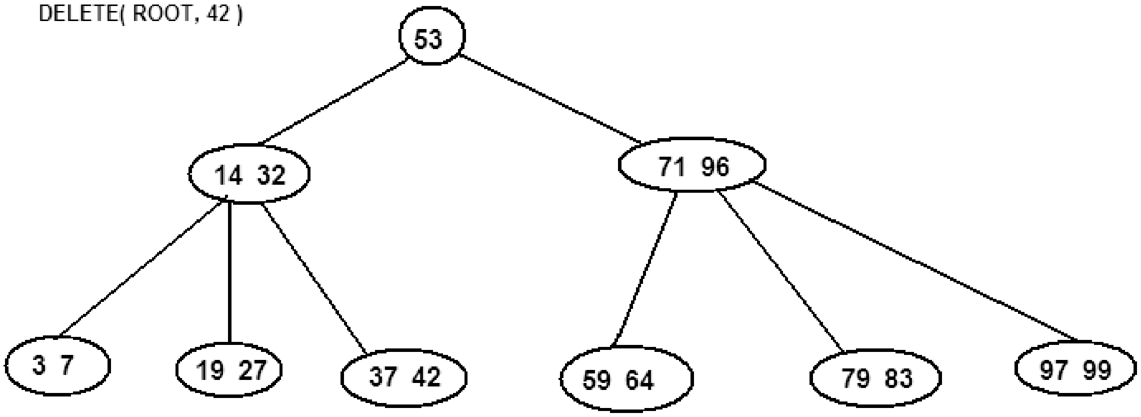


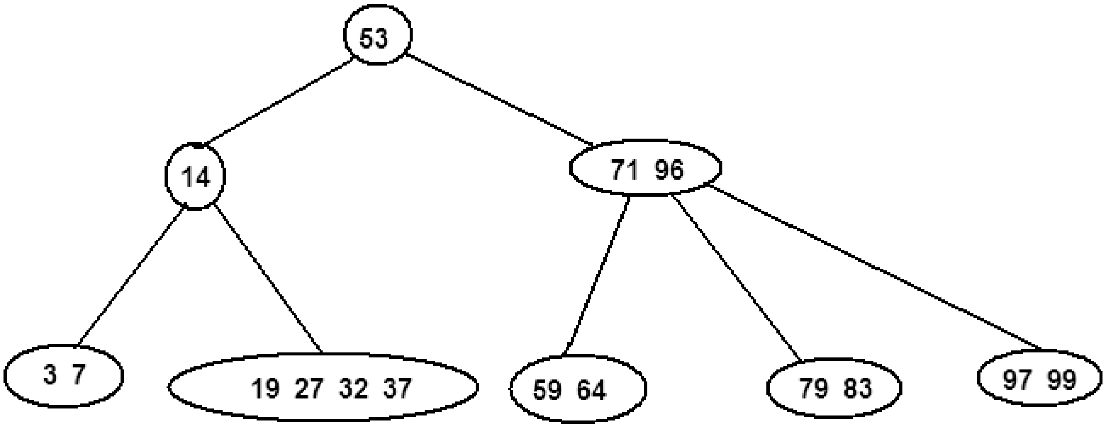
Case 2: In the second case, after the deletion the node has less than minimum n/2 values. Let us say we delete 92 from the tree. After 92 is deleted, the node has only one value 83. But a node adjacent to it consist 3 values (i.e., there are extra values in the adjacent node). Then the last value in that node 71 is pushed to its parent and the first value in the parent namely 79 is pushed into the node which has values less than minimum limit. Now the node has obtained the minimum required values.

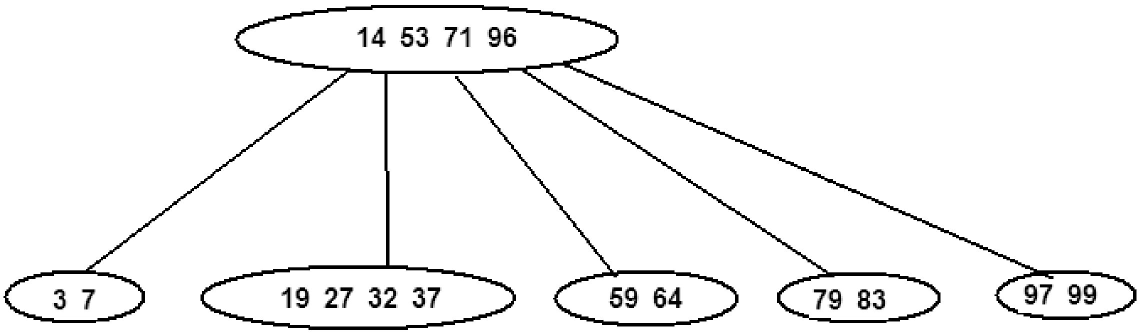




Case 3: In the previous case, there was an adjacent node with extra elements and hence the adjustment was made easily. But if all the nodes have exactly the minimum required, and if now a value is deleted from a node in this, then no value can be borrowed from any of the adjacent nodes. Hence as before a value from the parent is pushed into the node (in this case 32 is pushed down). Then the nodes are merged together. But we see that the parent node has insufficient number of values. Hence same process of merging takes place recursively till the entire tree is adjusted.







Algorithm

```
DELETE( ROOT, K )

Temp = SEARCH( ROOT, k ), DELETED  = 0, i = 1
While i < = count(temp)
    If (k = info(temp[i])
        DELETED = 1
        Delete temp[i]
    End if
End while

If DELETED = 0
    Print "Item not found"
    Return
Else
    If count(temp) < n / 2
        i = 1
        While i <= count(par)
            If count(child[i](par)) > n/2
                s = child[i](par)
                break
            Else
                i = i + 1
            End if
        End while
        If info(temp[1]) > info(s[count(s)])
            Ins(temp, info(par[1]))
            Ins(par, info(s[count(s)]))
        Else
            Ins(temp, info(par[count(par)]))
            Ins(par, info(s[1]))
        End if
    End if
End if
End DELETE
```

Splay trees

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) time.

Tree Splaying

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.

Advantages:

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly.

Tree rotations

To bring the recently accessed node closer to the tree *root*, a splay tree uses tree rotations. There are six types of tree rotations, three of which are symmetric to the other three. These are as follows:

- Left and right rotations
- Zig-zig left-left and zig-zig right-right rotations
- Zig-zag left-right and zig-zag right-left rotations

The first type of rotations, either left or right, is always a *terminal rotation*. In other words, the splaying is complete when we finish a left or right rotation.

Deciding which rotation to use

The decision to choose one of the above rotations depends on three things:

- Does the node we are trying to rotate have a grand-parent?
- Is the node left or right child of the parent?
- Is the parent left or right child of the grand-parent?

If the node does not have a grand-parent, we carry out a left rotation if it is the right child of the parent; otherwise, we carry out a right rotation.

If the node has a grand-parent, we have four cases to choose from:

If node is left of parent and parent is left of grand-parent, we do a *zig-zig right-right* rotation.

If node is left of parent but parent is right of grand-parent, we do a *zig-zag right-left* rotation.

If node is right of parent and parent is right of grand-parent, we do a *zig-zig left-left* rotation.

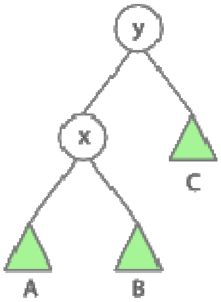
Finally, if node is right of parent but parent is left or grand-parent, we do a *zig-zag left-right* rotation.

The actual rotations are described in the following sections.

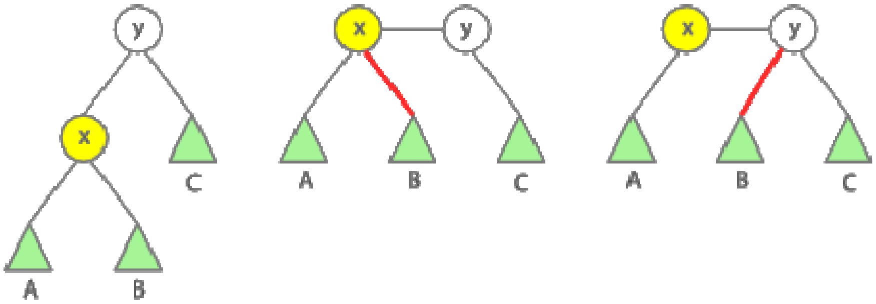
Left and right rotations

The following shows the intermediate steps in understanding a right rotation. The left rotation is symmetric to this.

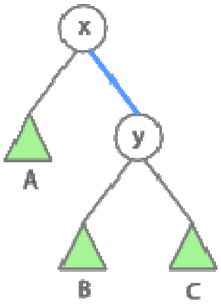
Splay x (rotate)



Right rotate x



Switch parent-child

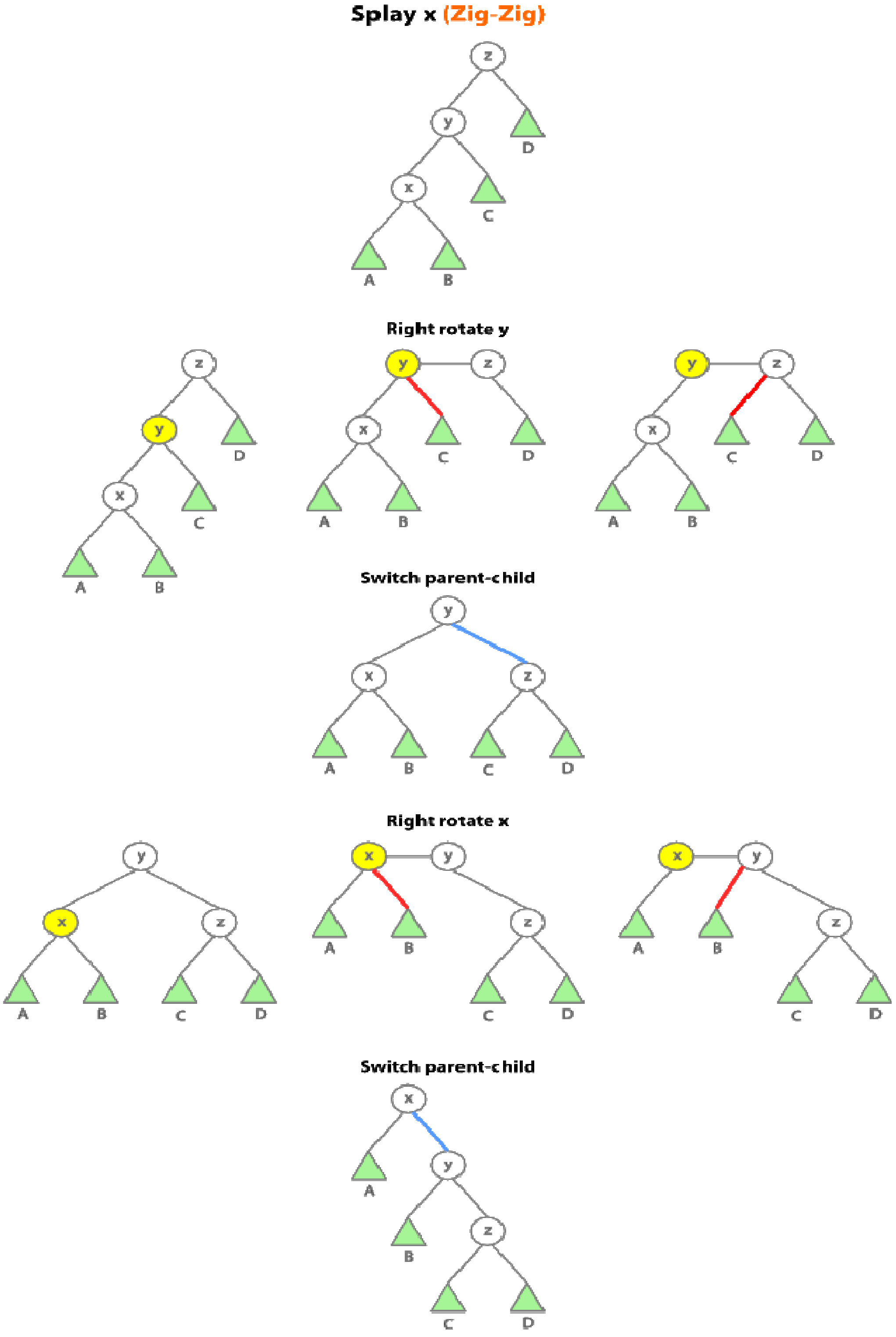


As we can see, each of the left or right rotations requires **five** pointer updates:

```
if (current == parent->left) {
   /* right rotate */
   parent->left = current->right;
   if (current->right)
      current->right->parent = parent;
   parent->parent = current;
   current->right = parent;
} else {
   /* left rotate */
   parent->right = current->left;
   if (current->left)
      current->left->parent = parent;
   parent->parent = current;
   current->left = parent;
}
current->parent = 0;
```

Zig-zig right-right and left-left rotations

The following shows the intermediate steps in understanding a zig-zig right-right rotation. The zig-zig left-left rotation is symmetric to this. Note in the following that with zig-zig rotations, we first do a right or left rotation on the **parent**, before doing a right or left rotation on the **node**.



As we can see, zig-zig right-right rotation requires **nine** pointer updates.

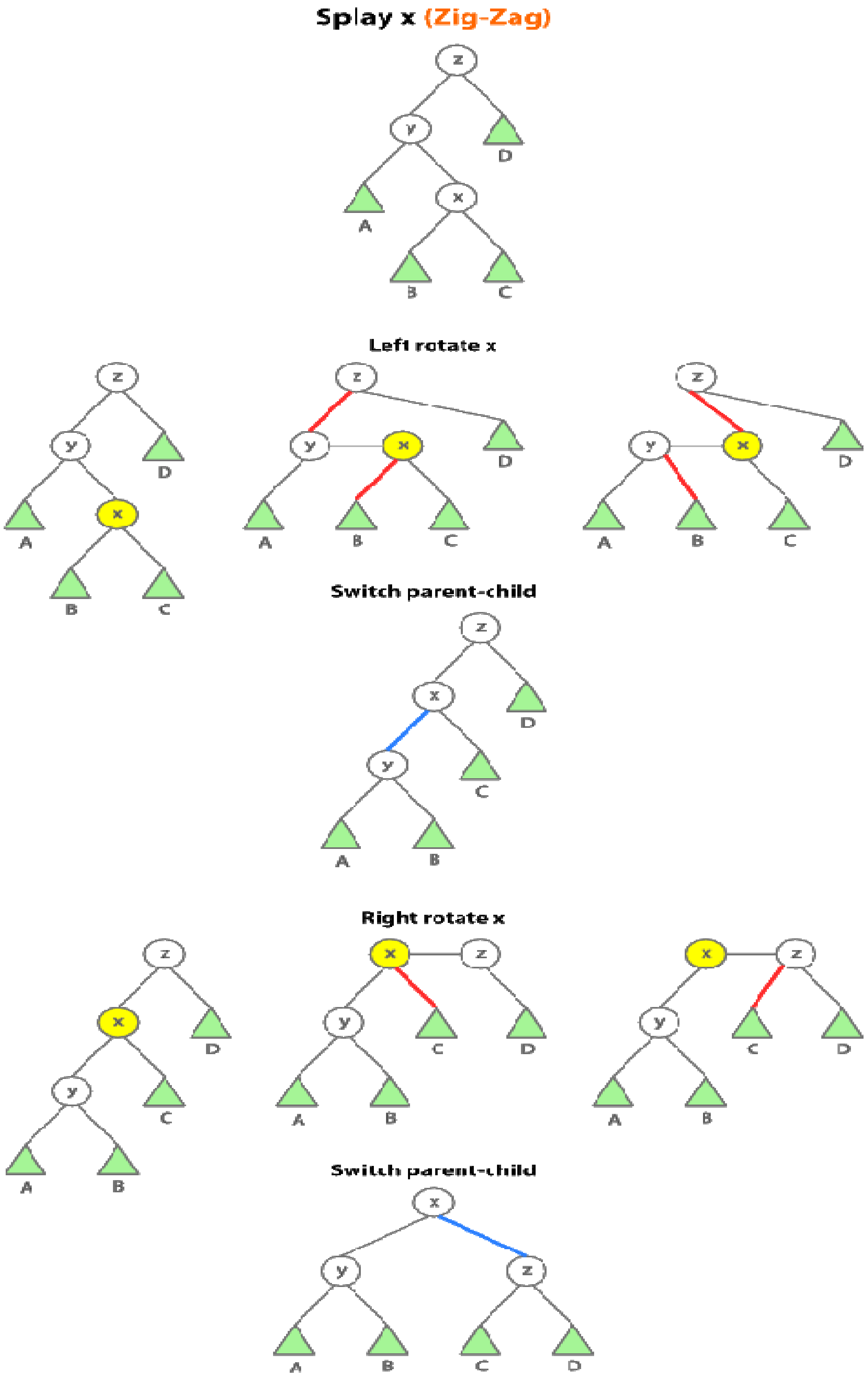
```
/* zig-zig right-right rotations */
if (current->right)
   current->right->parent = parent;
if (parent->right)
   parent->right->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = parent;
parent->parent = current;
grandParent->left = parent->right;
parent->right = grandParent;
parent->left = current->right;
current->right = parent;
```

The same number of pointer updates for zig-zig left-left rotation.

```
/* zig-zig left-left rotations */
if (current->left)
   current->left->parent = parent;
if (parent->left)
   parent->left->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = parent;
parent->parent = current;
grandParent->right = parent->left;
parent->left = grandParent;
parent->right = current->left;
current->left = parent;
```

Zig-zag left-right and right-left rotations

The following shows the intermediate steps in understanding a zig-zag left-right rotation. The zig-zag right-left rotation is symmetric to this. Note in the following that with zig-zag rotations, we do both rotations on the node, in contrast to zig-zig rotations.



As we can see, zig-zag left-right rotation requires **nine** pointer updates.
/* zig-zag right-left rotations */
if (current->left)

```
    current->left->parent = grandParent;
if (current->right)
    current->right->parent = parent;
current->parent = grandParent->parent;
grandParent->parent = current;
parent->parent = current;
grandParent->right = current->left;
parent->left = current->right;
current->right = parent;
current->left = grandParent;
```

The same number of pointer updates for zig-zag right-left rotation.

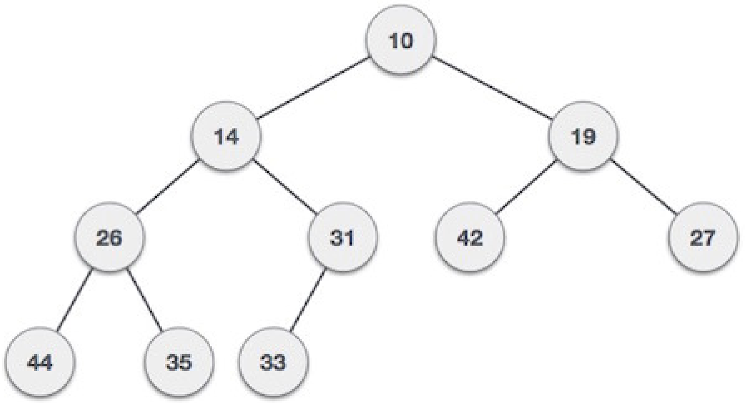
```
/* zig-zag left-right rotations */
if (current->left)
    current->left->parent = parent;
if (current->right)
    current->right->parent = grandParent;
current->parent = grandParent->parent;
grandParent->parent = current;
parent->parent = current;
grandParent->left = current->right;
parent->right = current->left;
current->left = parent;
current->right = grandParent;
```

Heap Trees

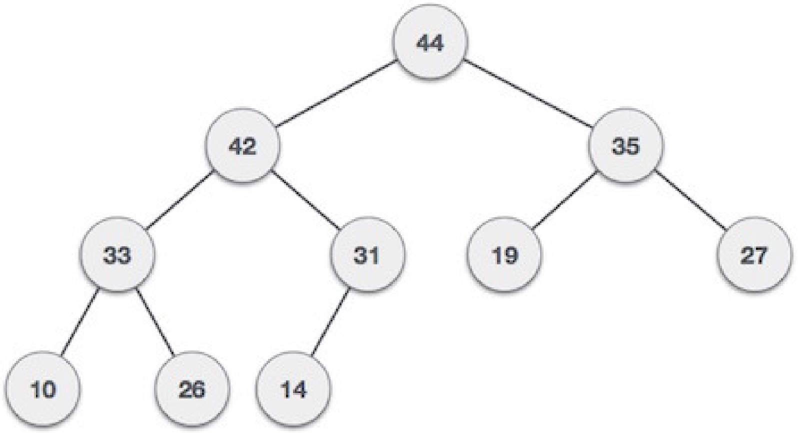
Heap is a special case of balanced binary tree data structure where root-node value is compared with its children and arranged accordingly. Heap trees are of two types- Max heap and Min heap.

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap – where the value of root node is less than or equal to either of its children.



Max-Heap – where the value of root node is greater than or equal to either of its children.



If a given node is in position I then the position of the left child and the right child can be calculated using **Left (L) = 2I** and **Right (R) = 2I + 1.** To check whether the right child exists or not, use the condition **R ≤ N.** If true, Right child exists otherwise not.The last node of the tree is **N/2.** After this position tree has only leaves.

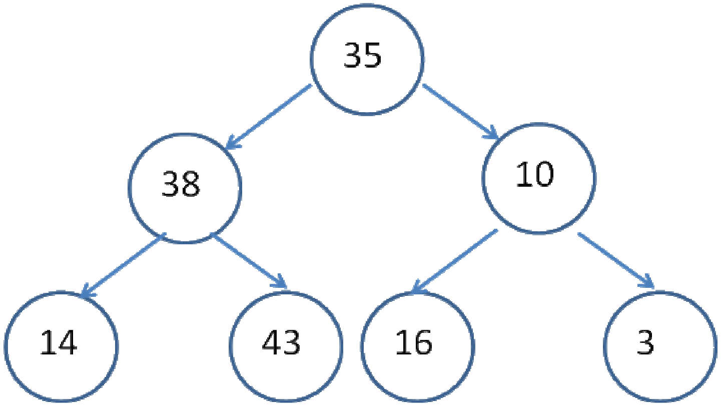
```
Procedure HEAPIFY(A,N)
// A is the list of elements
//N is the number of elements
For ( I = N/2 to 1)
    WALKDOWN (A,I,N)
END FOR
End Procedure
Procedure WALKDOWN(A, I,N)
//A is the list of unsorted elements
//N is the number of elements in the array
//I is the position of the node where the walkdown procedure is to be applied.

While I ≤ N/2
   L ← 2I, R ← 2I + 1

If A[L] > A[I] Then
          M ← L
Else
```

```
        M ← I
End If

If A[R] > A[M] and R ≤ N Then
         M ← R
End If
If M ≠ I Then
         A[I] ↔ A[M]
        I ←M

Else
        Return
End If
End While
End WALKDOWN
```

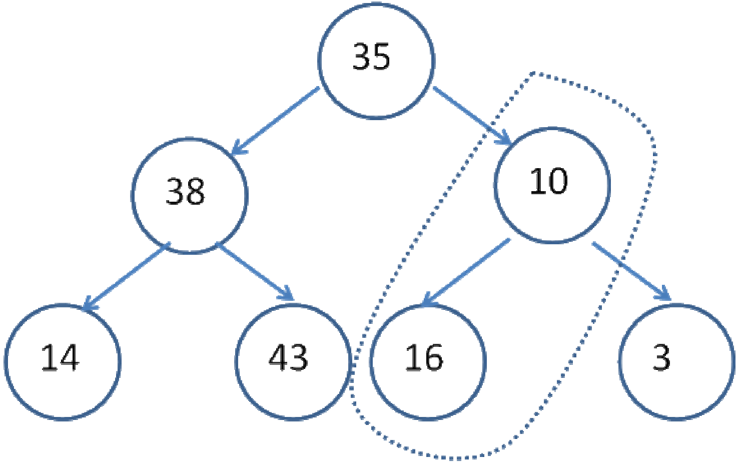
Example:
Given a list A with 8 elements:

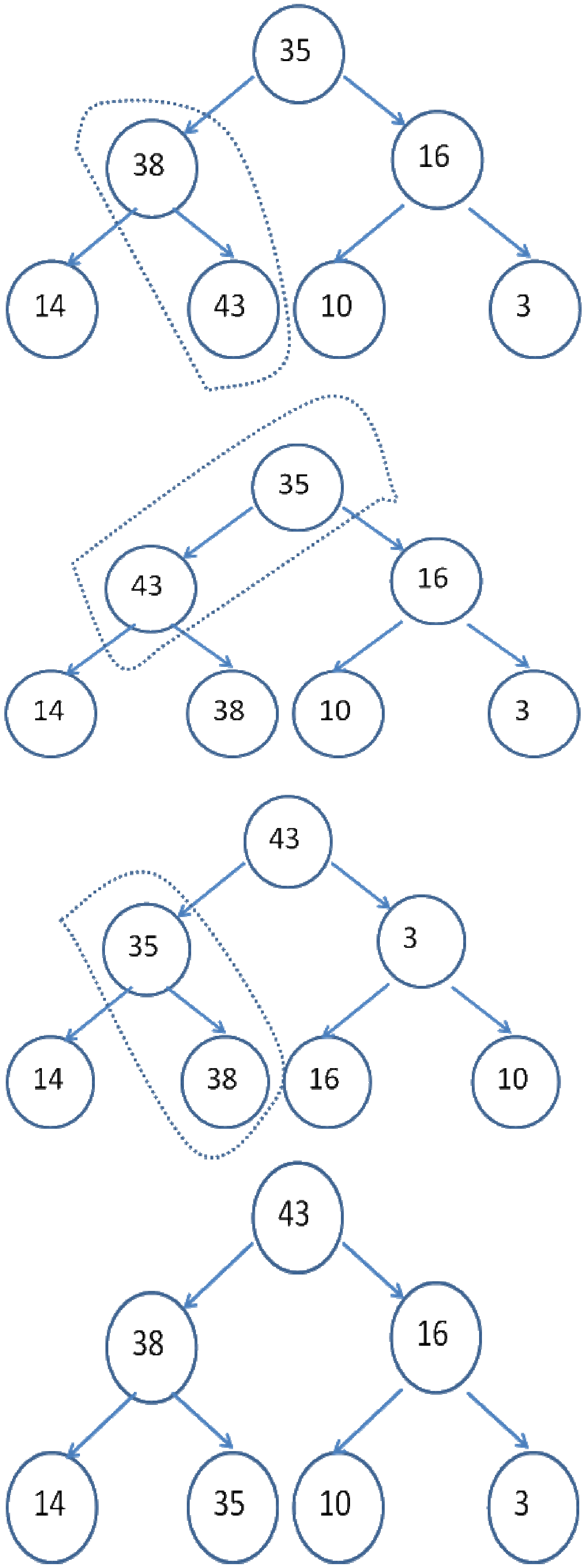
35	**38**	**10**	**14**	**43**	**16**	**3**

The given list is first converted into a binary tree as shown.



Then they are heapified , starting from the lowest parent.





The obtained tree is a Max heap tree.

TRIES

-Also called digital tree or radix tree or prefix tree.

-Tries are an excellent data structure for strings.

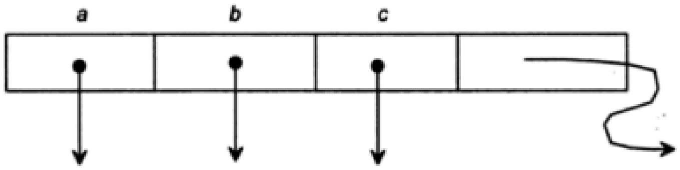
-Tries is a tree data structure used for storing collections of strings.

-Tries came from the word retrieval.

-Nodes store associative keys (strings) and values.

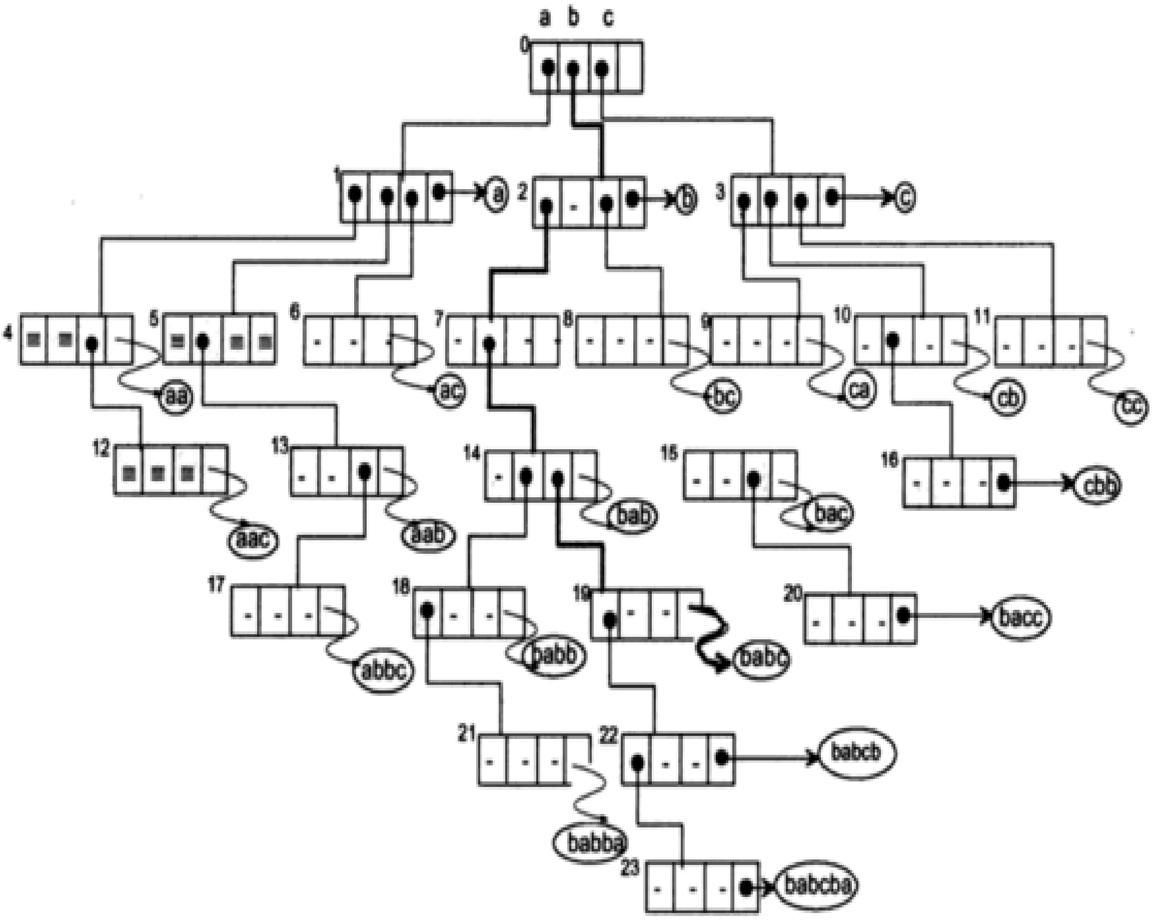
Tries Structure

Let us consider the case of a tries tree of order 3.Let the key value in this tries is constituted from three letters namely a, b and c. Each node has the following structure:



```
// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};
```

Here 3 link fields' points to three nodes in the next level and the last field are called the information field. The information field has the value either TRUE or FALSE. If the value is TRUE then traversing from the root node to this node yields some information. A tries of order 3 is given below:



For example, let us assume the key value ‘bab’. Starting from the root node, and branching based on each letter, traversal will be 0-2-7-14 and in node 14, the information field TRUE implies that ‘bab’ is a word.

NOTE:

- Tries indexing is suitable for maintaining variable sized key values.
- Actual key value is never stored but key values are implied through links.
- If English alphabets are used, then a trie of order 26 can maintain whole English dictionary.

Operations on Trie

Searching

Searching for a key begins at the root node, compare the characters and move down. The search can terminate due to end of string or lack of key in tries. In the former case, if the *value* field of last node is non-zero then the key exists in tries. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

PSEUDOCODE. The search algorithm involves the following steps:

1. For each character in the string, see if there is a child node with that character as the content.

2. If that character does not exist, return false.
3. If that character exist, repeat step 1.
4. Do the above steps until the end of string is reached.
5. When end of string is reached and if the marker (NotLeaf) of the current Node is set to false, return true, else return false.

Insertion

Inserting a key into trie is simple approach. Every character of input key is inserted as an individual trie node. Note that the children are an array of pointers to next level trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth.

PSEUDOCODE: Any insertion would ideally be following the below algorithm:

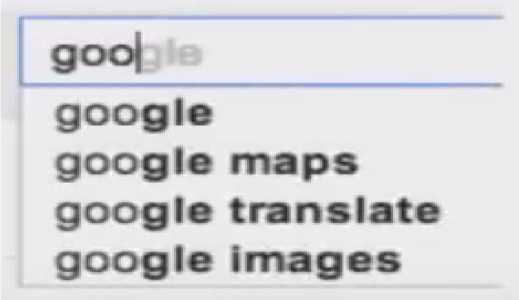
1. Find the place of the item by following bits.
2. If there is nothing, just insert the item there as a leaf node.
3. If there is something on the leaf node, it becomes a new internal node. Build a new sub tree to that inner node depending how the item to be inserted and the item that was in the leaf node differs.
4. Create new leaf nodes where you store the item that was to be inserted and the item that was originally in the leaf node.

Deletion

Deletion procedure is same as searching and insertion with some modification. To delete a key from a trie, trace down the path corresponding to the key to be deleted, and when we reach the appropriate node, set the TAG field of this node as FALSE. If all the field entries of this node are NULL, then return this node to the pool of free storage. To do so, maintain a stack of PATH to store all the pointers of nodes on the path from the root to the last node reached.

Application of Tries

- Retrieval operation of lexicographic words in a dictionary.
- Word processing packages to support the spelling check.
- Useful for storing a predictive text for auto complete.



GRAPHS

UNIT IV: Graph Theory Terminology, Graph Representations, Graph operations- Graph Traversals (BFS & DFS), Connected components, Spanning Trees, Biconnected Components, Minimum Spanning Trees- Krushkal's Algorithm , Prim's Algorithm, Shortest paths, Transitive closure, All pairs Shortest path-Marshall's Algorithm.

BASIC CONCEPTS

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

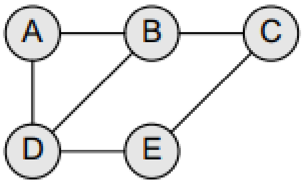
WHY GRAPHS ARE USEFUL

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- Family trees: in which the member nodes have an edge from parent to each of their children.
- Transportation networks: in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

Definition

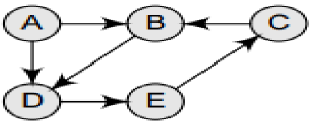
A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.



Undirected graph

A graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}. Note that there are five vertices or nodes and six edges in the graph.

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.



In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at nodeB (terminalnode).
Directed Graph

Graph Terminology

Adjacent nodes or neighbours

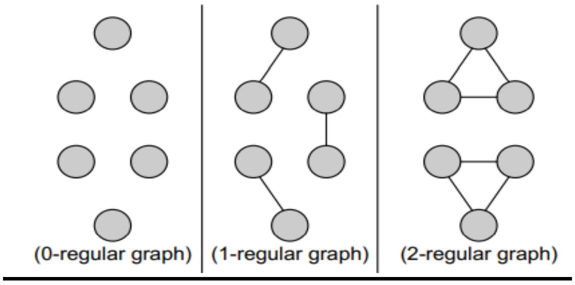
For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbours.

Degree of a node

Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

Regular graph

It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k–regular graph or a regular graph of degree k.



Path

A path P written as P = {v0 , v1 , v2 , ..., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0 , v = vn and vi–1 is adjacent to vi for i = 1, 2, 3, ..., n.

Closed path

A path P is known as a closed path if the edge has the same end-points. That is, if v0 = vn .

Simple path

A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn . If v0 = vn , then the path is called a closed simple path.

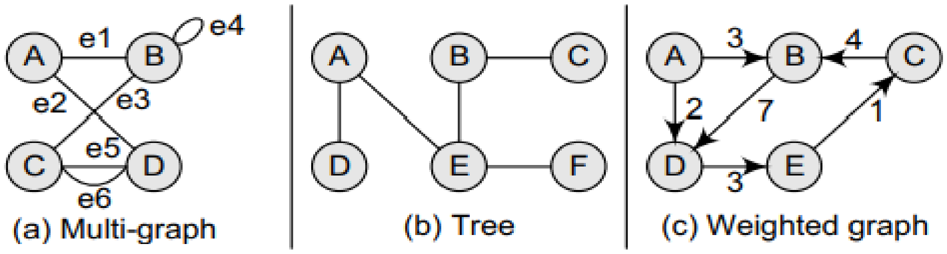
Cycle

A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

Connected graph

A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph

Complete graph A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G



(a) Multi-graph (b) Tree (c) Weighted graph

Labelled graph or weighted graph

A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.

Multiple edges Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Loop An edge that has identical end-points is called a loop. That is, e = (u, u).

Multi-graph A graph with multiple edges and/or loops is called a multi-graph.

Size of a graph The size of a graph is the total number of edges in i

Directed Graphs

A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e.Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u. ∑ Nodes u and v are adjacent to each other

Terminology of a Directed Graph

Out-degree of a node The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u.

In-degree of a node The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.

Degree of a node The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).

Isolated vertex A vertex with degree zero. Such a vertex is not an end-point of any edge.

P**endant vertex** (also known as leaf vertex) A vertex with degree one.

Cut vertex A vertex which when deleted would disconnect the remaining graph.

Source A node u is known as a source if it has a positive out-degree but a zero in-degree.

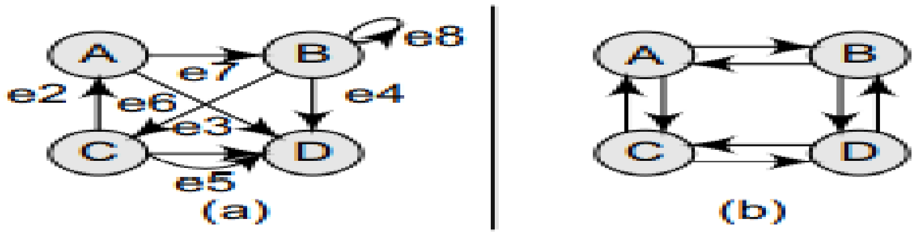
Sink A node u is known as a sink if it has a positive in-degree but a zero out-degree.

Reachability A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v. For example, if you consider the directed graph given in Fig. 13.5(a), you will observe that node D is reachable from node A.

Strongly connected directed graph A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Weakly connected digraph A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Parallel/Multiple edges Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G. In below diagram e3 and e5 are multiple edges connecting nodes C and D.



(a) Directed acyclic graph and (b) strongly connected directed acyclic graph

Simple directed graph A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with an exception that it cannot have more than one loop at a given node.

REPRESENTATION OF GRAPHS

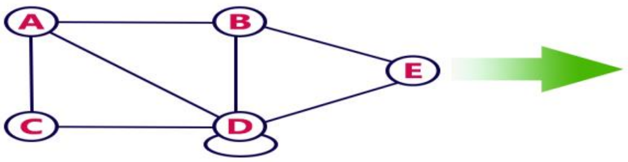
There are three common ways of storing graphs in the computer's memory.

1. **Adjacency Matrix**
2. **Adjacency List**

Adjacency Matrix Representation

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...

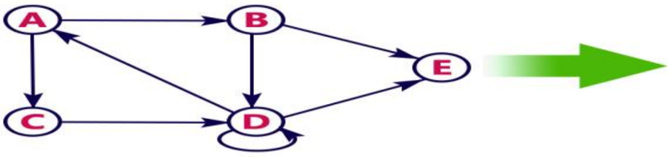


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

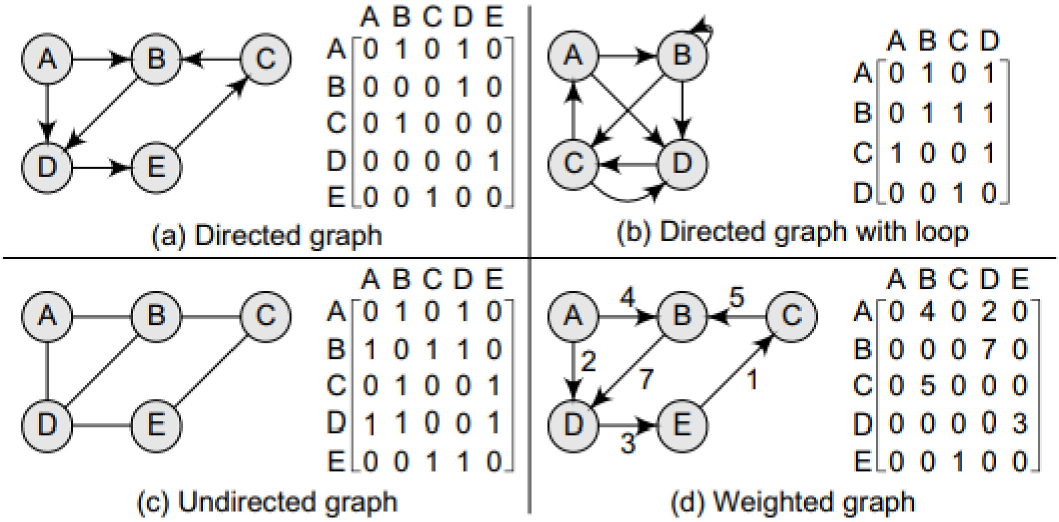
Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

$$a_{ij} = \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j\text{, that is there is an edge } (v_i, v_j)\text{]A} \\ 0 & \text{[otherwise]} \end{cases}$$

Directed graph representation...



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(a) Directed graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

(b) Directed graph with loop

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

(c) Undirected graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 2 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 5 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 3 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(d) Weighted graph

Graphs and their corresponding adjacency matrices

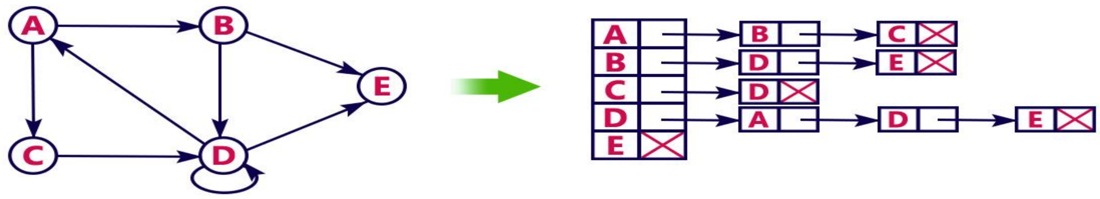
From the above examples, we can draw the following conclusions:

1. For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
2. The adjacency matrix of an undirected graph is symmetric.
3. The memory use of an adjacency matrix is O(n2), where n is the number of nodes in the graph.
4. Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
5. The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

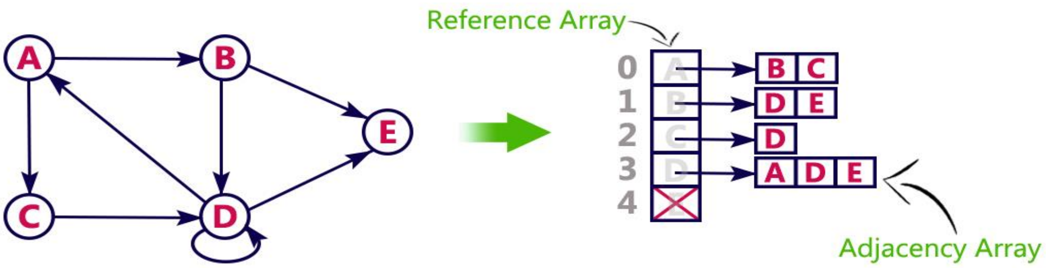
Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

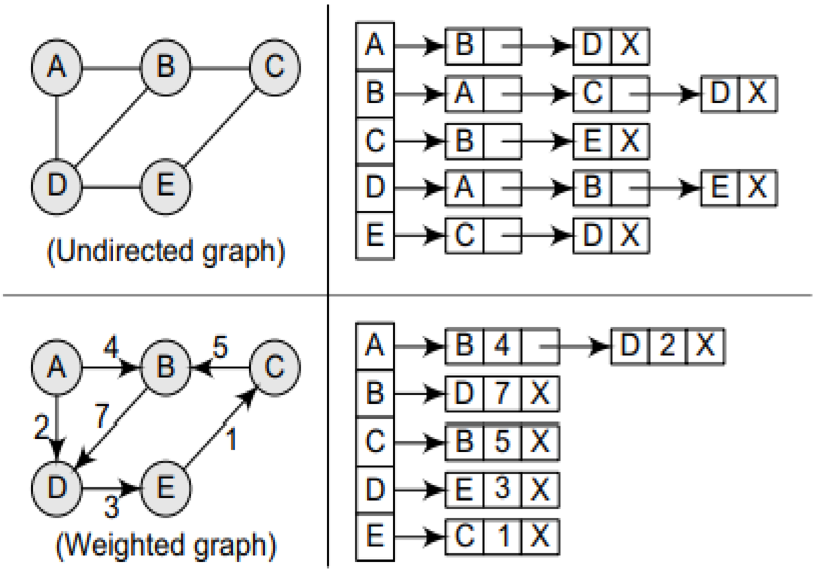
For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..


Reference Array

Adjacency Array



Adjacency list for an undirected graph and a weighted graph

The key advantages of using an adjacency list are:

1 It is easy to follow and clearly shows the adjacent nodes of a particular node.

2. It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.

3. Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

4. For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G.

5. For an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u.

Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in 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. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighboring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

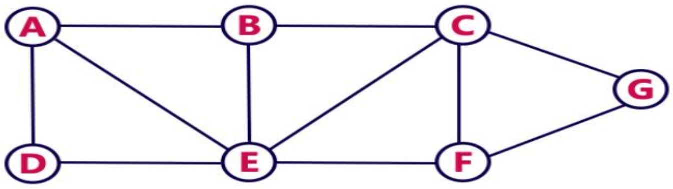
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

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 insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete 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 from the graph

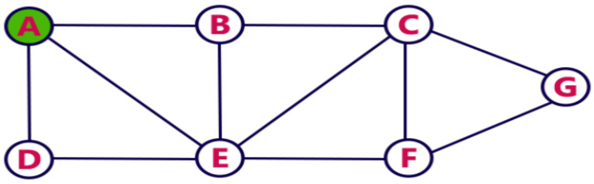
Example

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

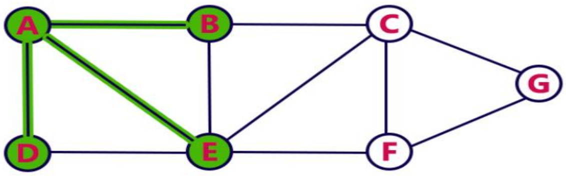


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

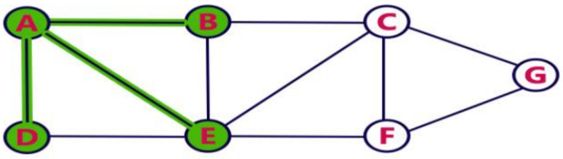


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

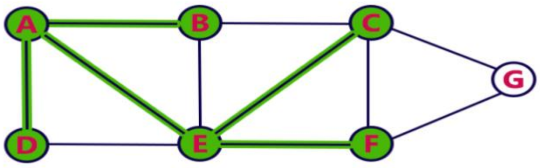


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

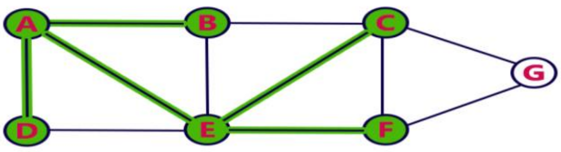


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

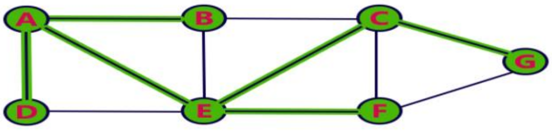


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

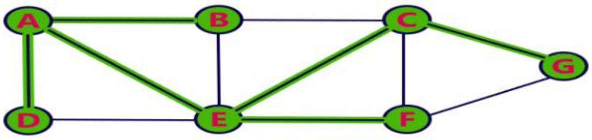


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

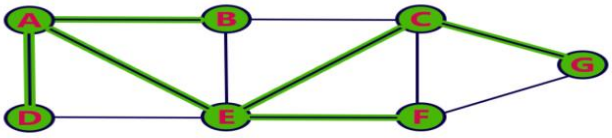


Queue

						G

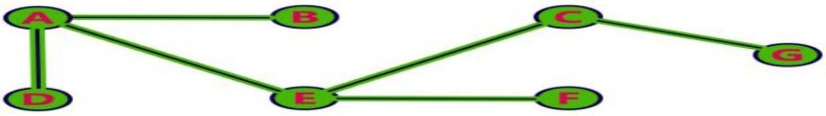
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



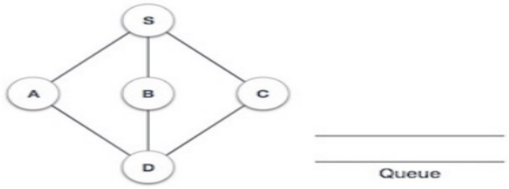
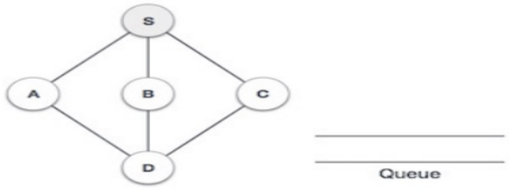
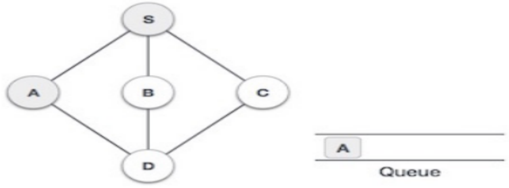
Queue

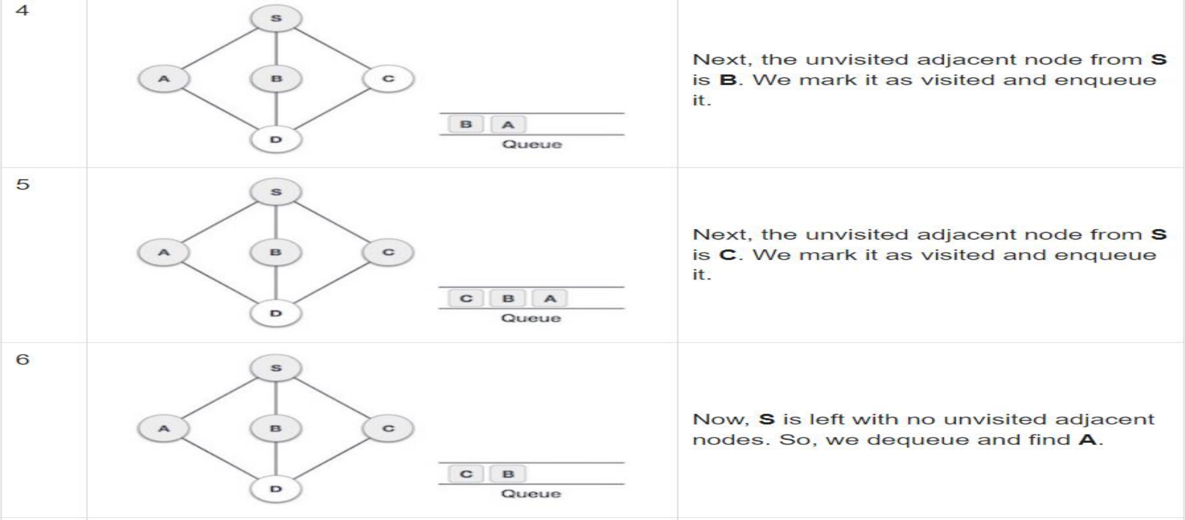
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



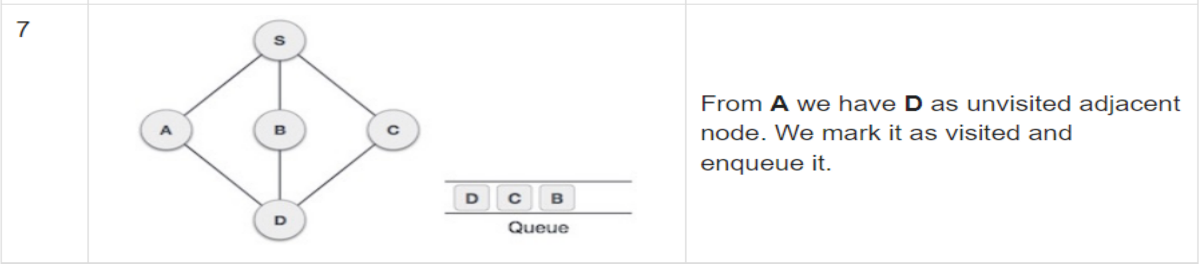
Example 2:

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.



4	B A	Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5	C B A	Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6	C B	Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

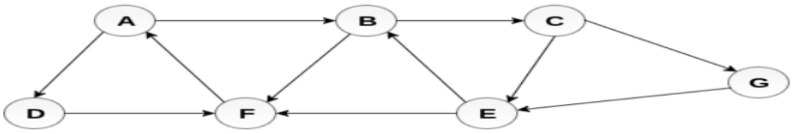


7	D C B	From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

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

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at 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 from **QUEUE1**.

Lets start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.

QUEUE1 = {A}
QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

QUEUE1 = {B, D}
QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

QUEUE1 = {D, C, F}
QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

QUEUE1 = {C, F}
QUEUE2 = { A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

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

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

QUEUE1 = {E, G}

QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.
QUEUE1 = {G}
QUEUE2 = {A, B, D, C, F, E}

Applications of BFS Algorithm

Some of the real-life applications where a BFS algorithm implementation can be highly effective.

- **Un-weighted Graphs:** BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.
- **P2P Networks:** BFS can be implemented to locate all the nearest or neighboring nodes in a peer to peer network. This will find the required data faster.
- **Web Crawlers:** Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.
- **Navigation Systems:** BFS can help find all the neighboring locations from the main or source location.
- **Network Broadcasting:** A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

DFS (Depth First Search)

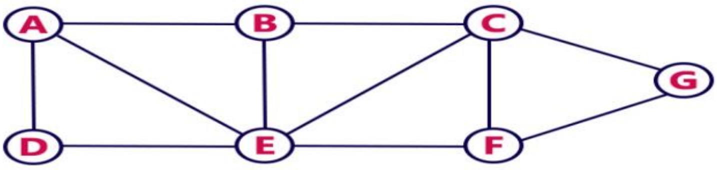
DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal
we use the following steps to implement DFS traversal...

- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we reached the current vertex.

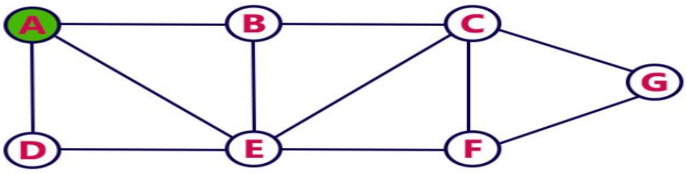
Example

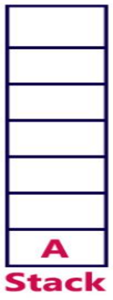
Consider the following example graph to perform DFS traversal



Step 1:

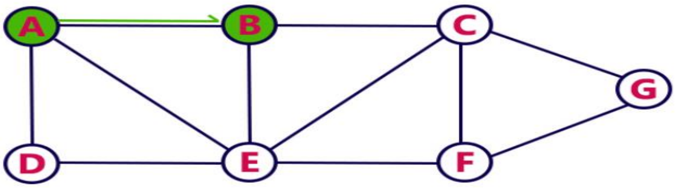
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

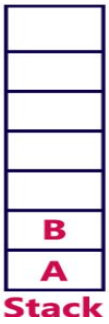




Step 2:

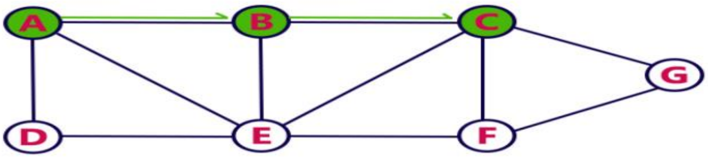
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

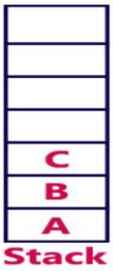




Step 3:

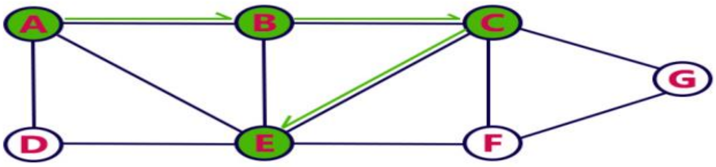
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

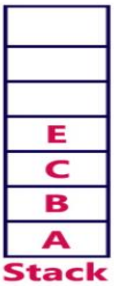




Step 4:

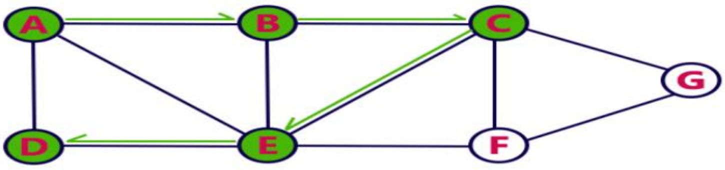
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

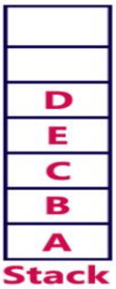




Step 5:

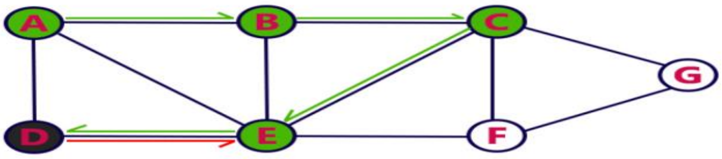
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

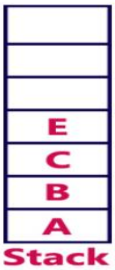




Step 6:

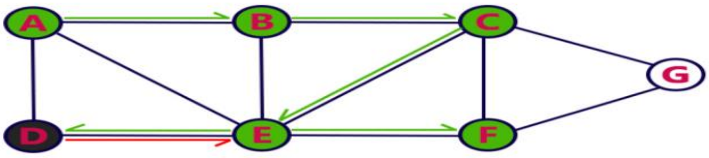
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

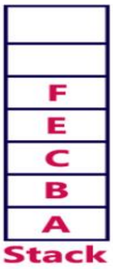




Step 7:

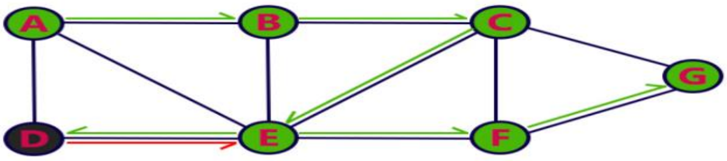
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

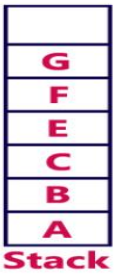




Step 8:

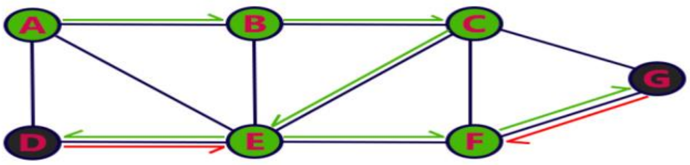
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

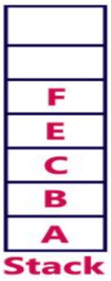




Step 9:

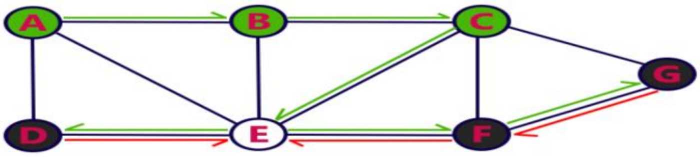
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

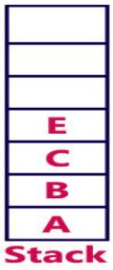




Step 10:

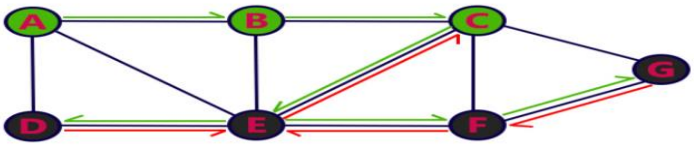
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

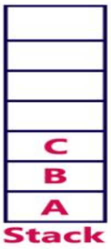




Step 11:

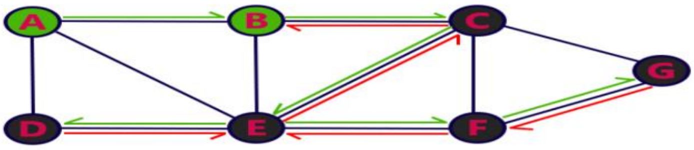
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

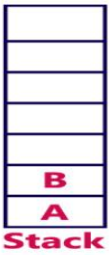




Step 12:

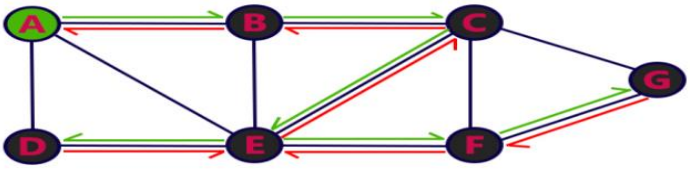
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

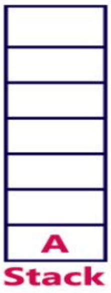




Step 13:

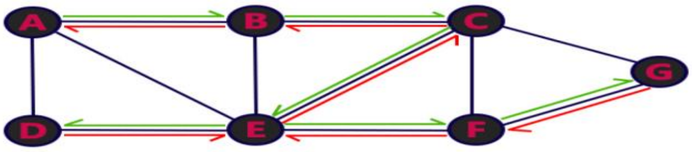
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

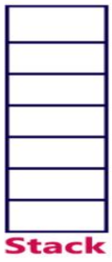




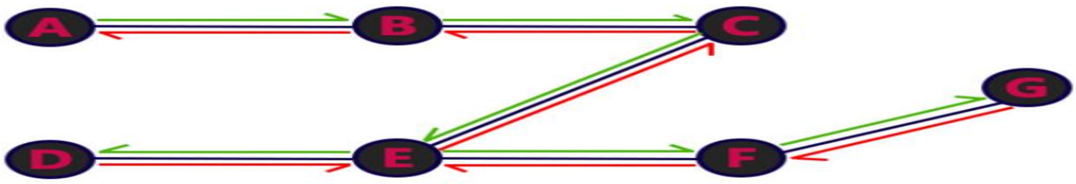
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Applications of Depth-First Search Algorithm

Depth-first search is useful for:

1. Finding a path between two specified nodes, u and v, of an unweighted graph.
2. Finding a path between two specified nodes, u and v, of a weighted graph.
3. Finding whether a graph is connected or not.
4. Computing the spanning tree of a connected graph.

SHORTEST PATH ALGORITHMS

Three different algorithms to calculate the shortest path between the vertices of a graph G. These algorithms include:

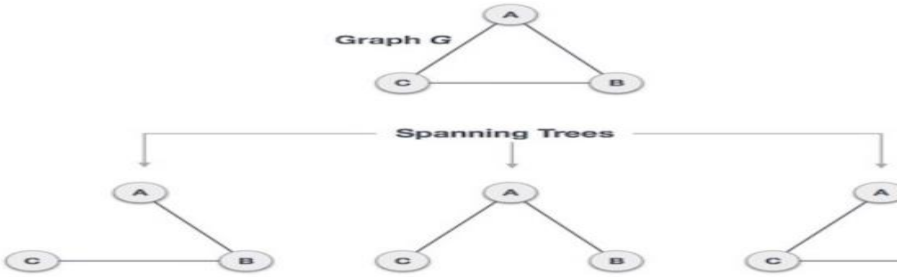
1. Minimum spanning tree
2. Dijkstra's algorithm
3. Warshall's algorithm

While the first two use an adjacency list to find the shortest path, Warshall's algorithm uses an adjacency matrix to do the same.

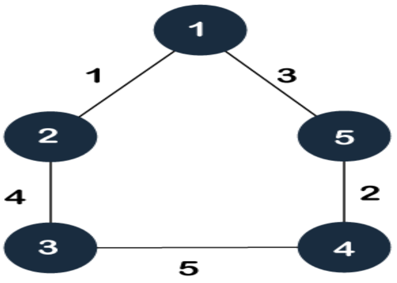
Minimum Spanning Trees

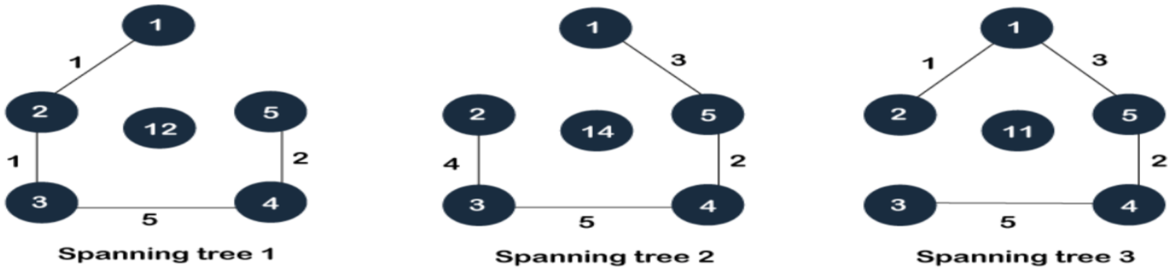
A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum **n^{n-2}** number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence **3^{3-2} = 3** spanning trees are possible.

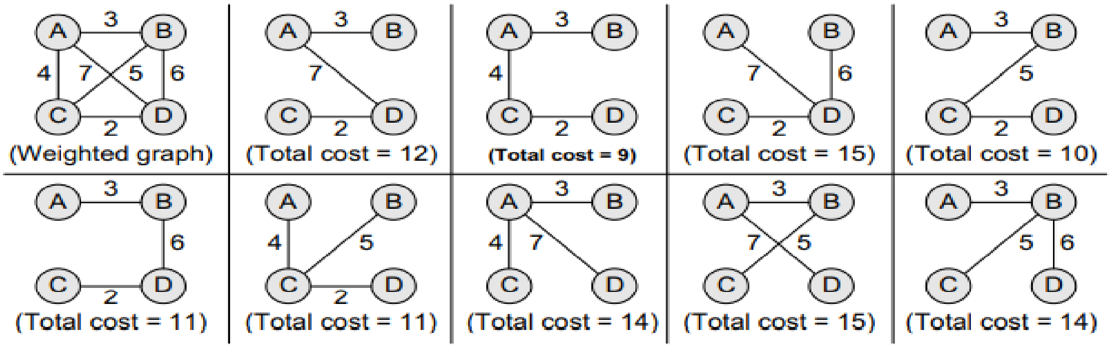




Spanning tree 1

Spanning tree 2

Spanning tree 3



(Weighted graph)

(Total cost = 12)

(Total cost = 9)

(Total cost = 15)

(Total cost = 10)

(Total cost = 11)

(Total cost = 11)

(Total cost = 14)

(Total cost = 15)

(Total cost = 14)

Weighted graph and its spanning trees

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Application of Spanning Tree:

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees is −

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

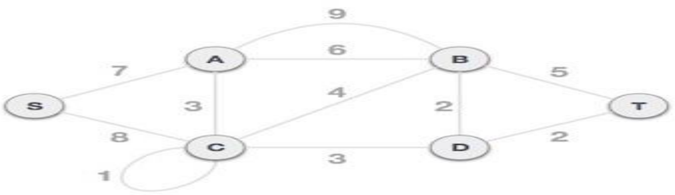
- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

Kruskal’s Minimum Spanning Tree Algorithm

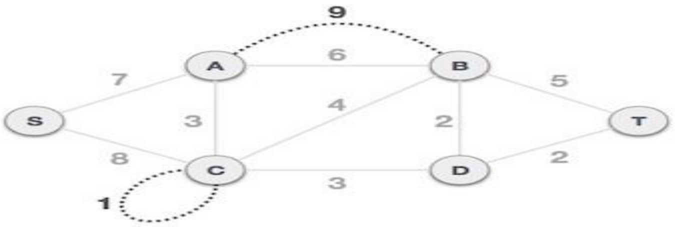
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

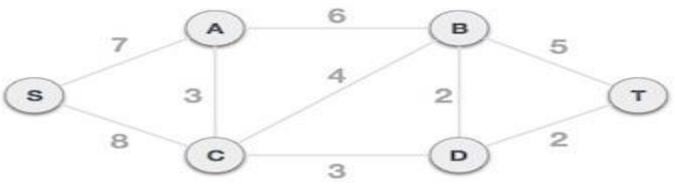


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.

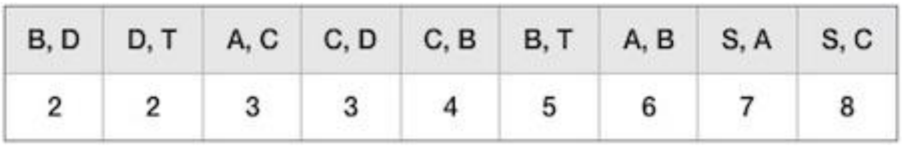


In case of parallel edges, keep the one which has the least cost associated and remove all others.



Step 2 - Arrange all edges in their increasing order of weight

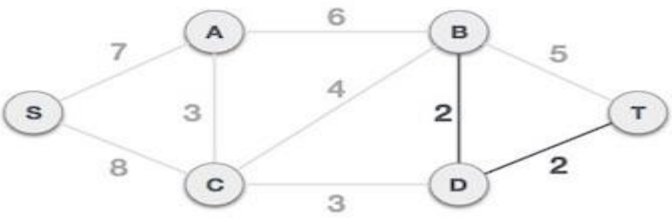
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).



B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

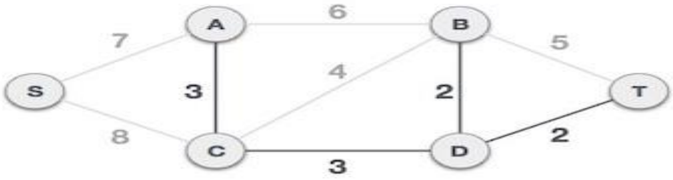
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

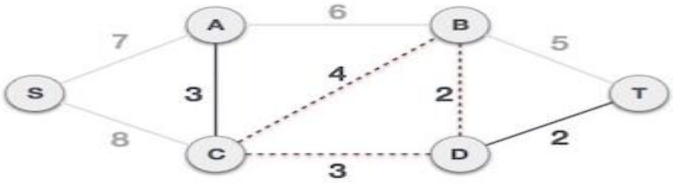


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

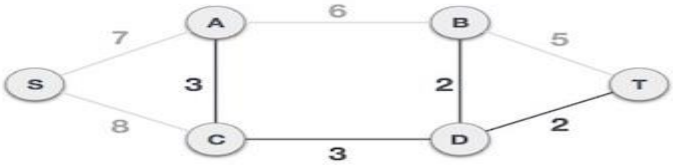
Next cost is 3, and associated edges are A,C and C,D. We add them again −



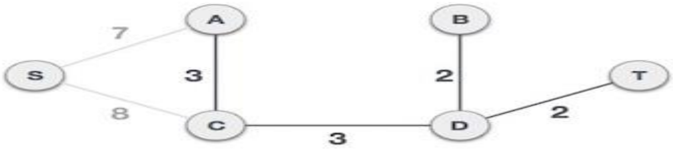
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



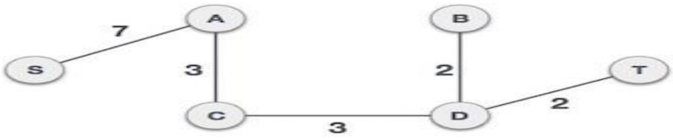
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

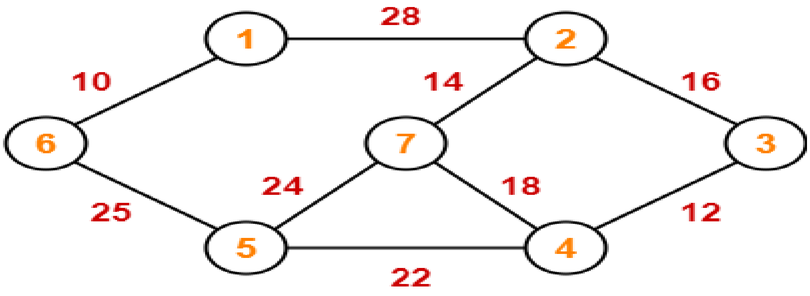


Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree

Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-

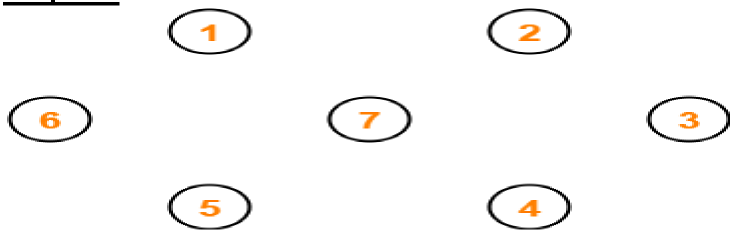


Solution-

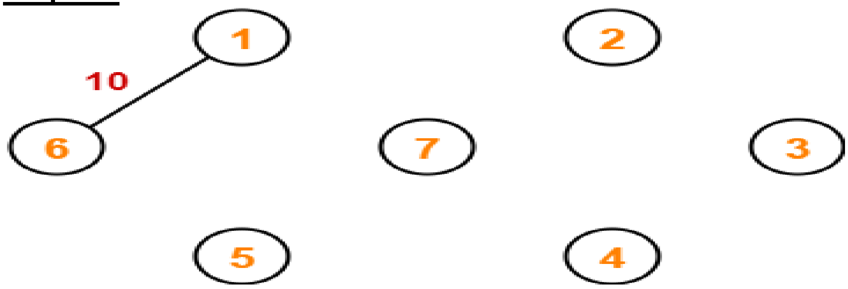
To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

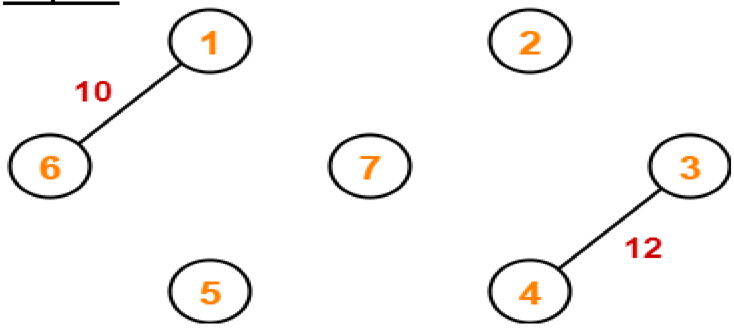
Step-01:



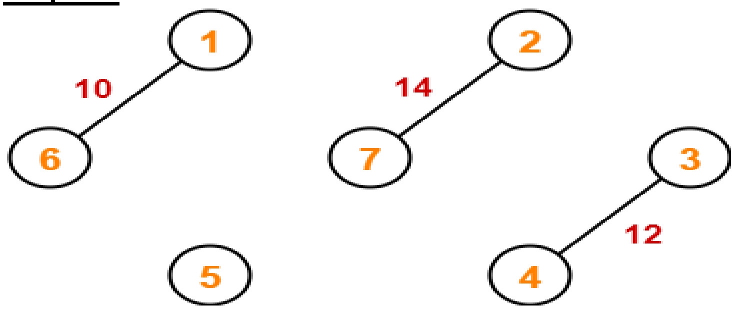
Step-02:



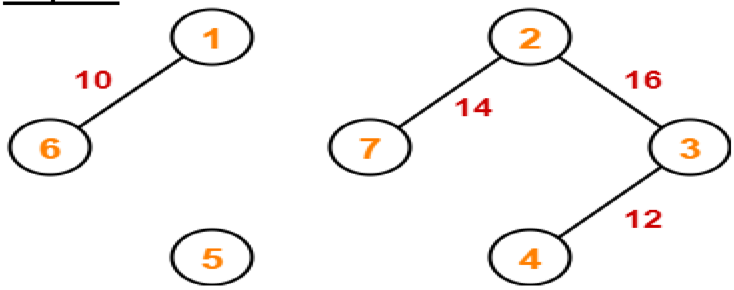
Step-03:



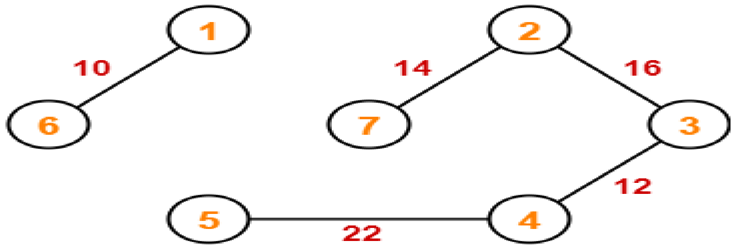
Step-04:



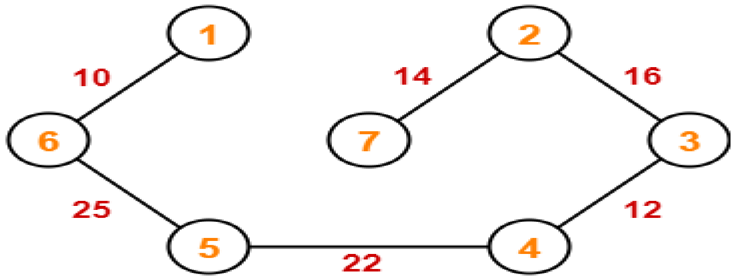
Step-05:



Step-06:



Step-07:



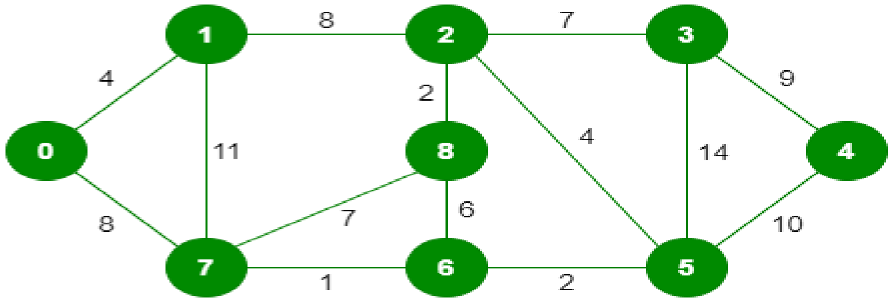
Since all the vertices have been connected / included in the MST, so we stop.

Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units



Example

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

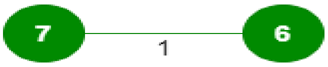
After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3

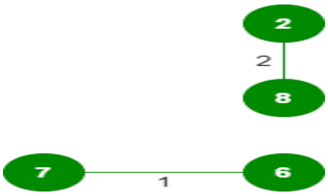
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from the sorted list of edges

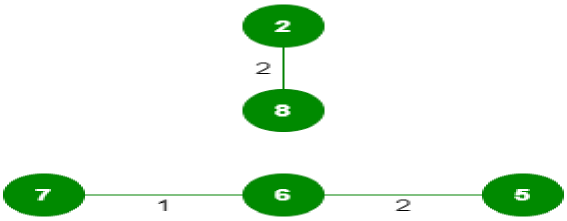
1. *Pick edge 7-6:* No cycle is formed, include it.



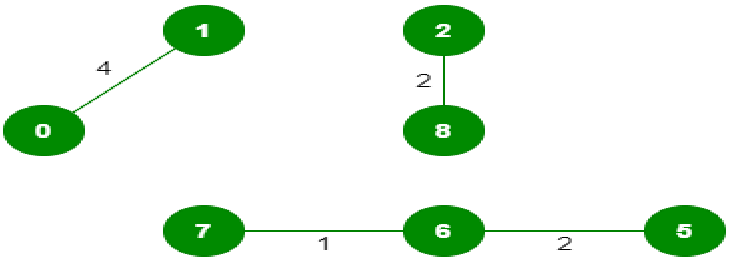
2.Pick edge 8-2: No cycle is formed, include it.



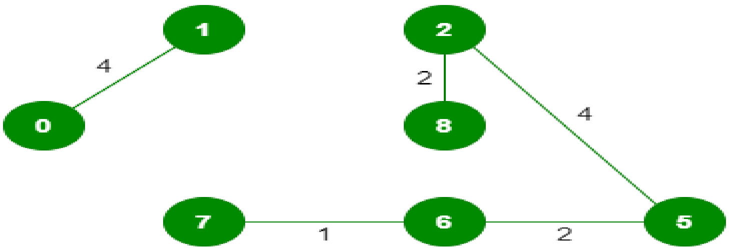
3.Pick edge 6-5: No cycle is formed, include it.



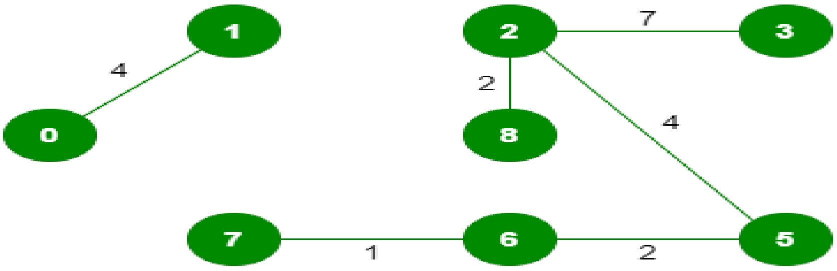
4. *Pick edge 0-1:* No cycle is formed, include it.



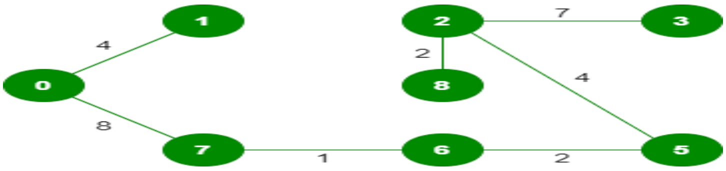
5.Pick edge 2-5: No cycle is formed, include it.



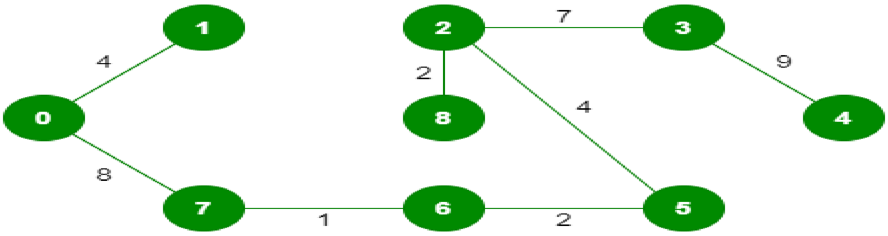
6.*Pick edge 8-6:* Since including this edge results in the cycle, discard it.
7. *Pick edge 2-3:* No cycle is formed, include it.



8. *Pick edge 7-8:* Since including this edge results in the cycle, discard it.
9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in the cycle, discard it.
11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

Prim's Algorithm-

- Prim's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

Prim's Algorithm Implementation-

The implementation of Prim's Algorithm is explained in the following steps-

Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step-02:

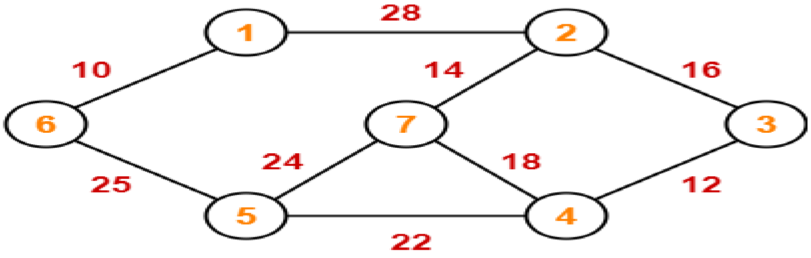
- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

Problem-01:

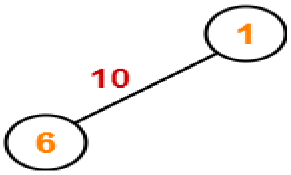
Construct the minimum spanning tree (MST) for the given graph using Prim’s Algorithm-



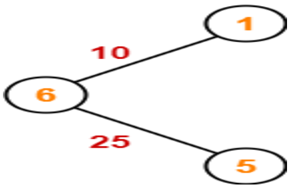
Solution-

The above discussed steps are followed to find the minimum cost spanning tree using Prim’s Algorithm-

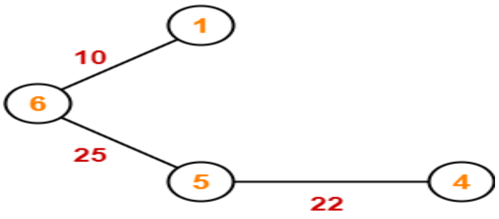
Step-01:



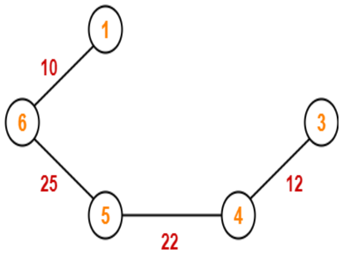
Step-02:



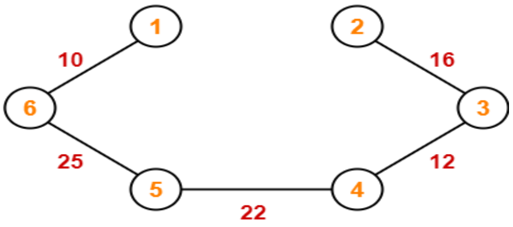
Step-03:



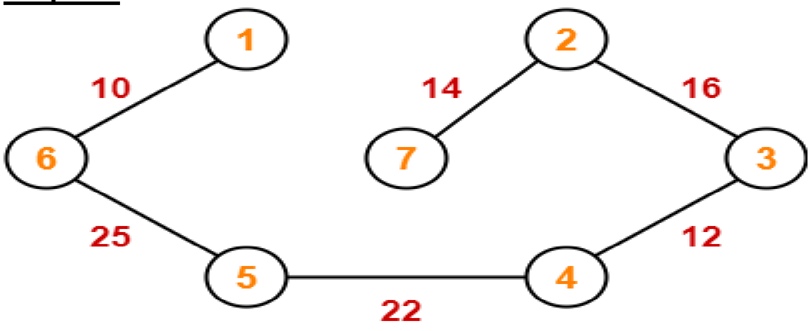
Step-04:



Step-05:



Step-06:



- Since all the vertices have been included in the MST, so we stop.

  Now, Cost of Minimum Spanning Tree

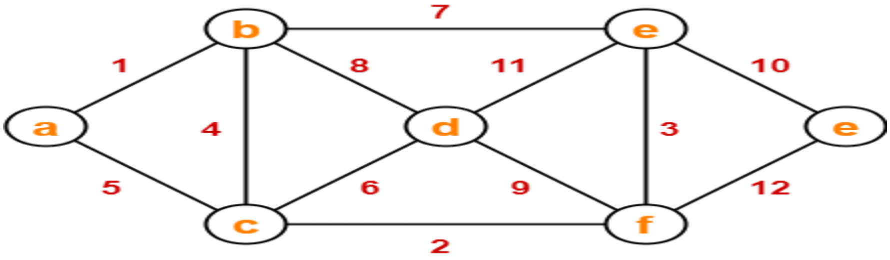
  = Sum of all edge weights

  = 10 + 25 + 22 + 12 + 16 + 14

  = 99 units

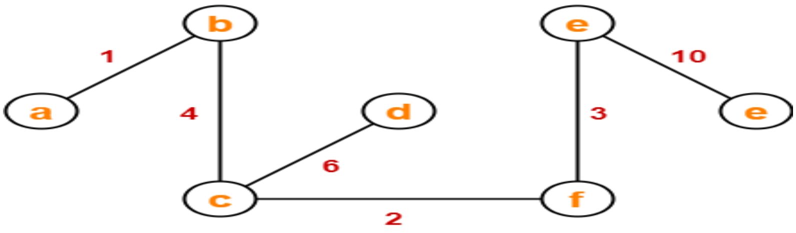
Example 2

Using Prim's Algorithm, find the cost of minimum spanning tree (MST) of the given graph-



Solution-

The minimum spanning tree obtained by the application of Prim's Algorithm on the given graph is as shown below-



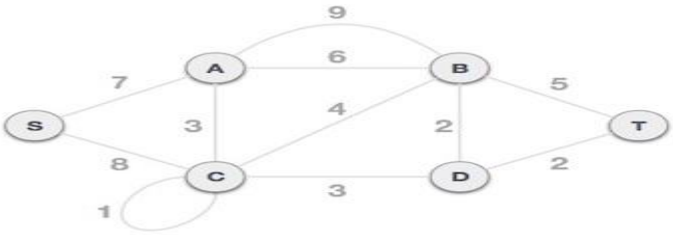
Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

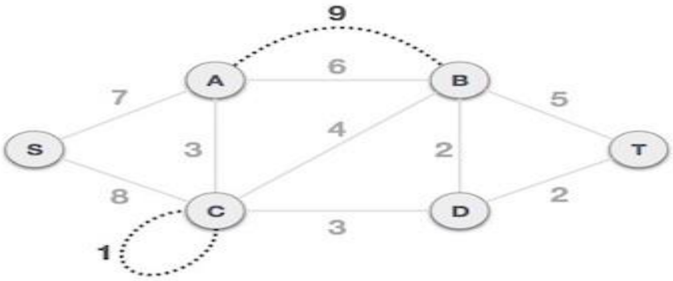
= 1 + 4 + 2 + 6 + 3 + 10

= 26 units

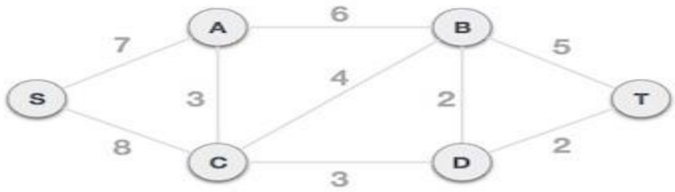
Example:



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

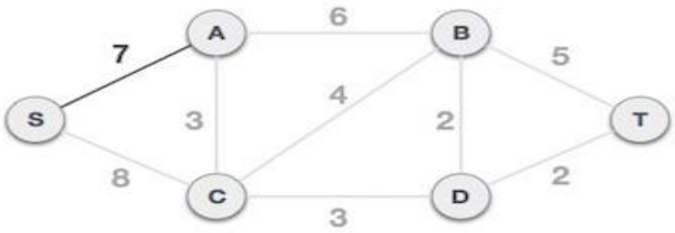


Step 2 - Choose any arbitrary node as root node

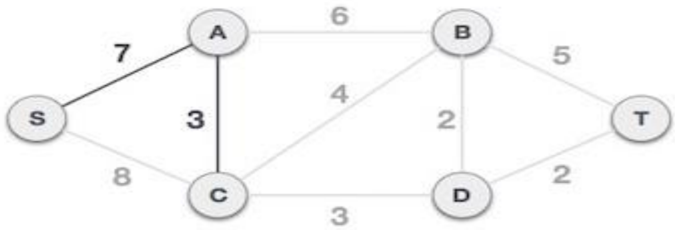
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

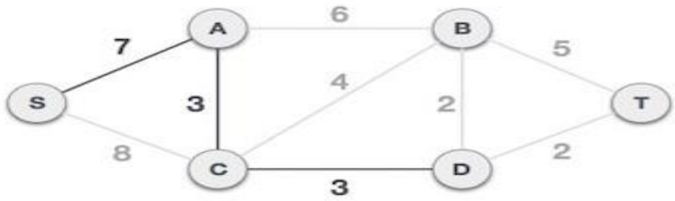
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



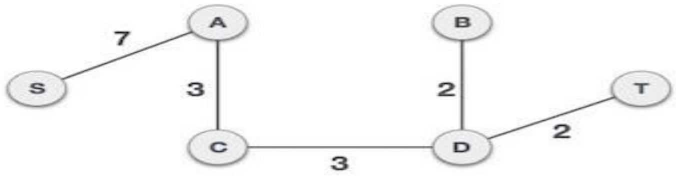
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



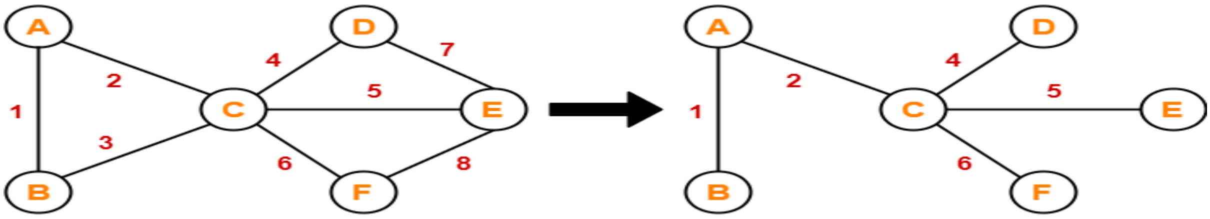
We may find that the output spanning tree of the same graph using two different algorithms is same.

Comparison between Prim's and Krushkals

1. If all the edge weights are distinct, then both the algorithms are guaranteed to find the same MST.

Example-

Consider the following example-



Given Graph

Minimum Spanning Tree (MST)
(Cost = 18 units)

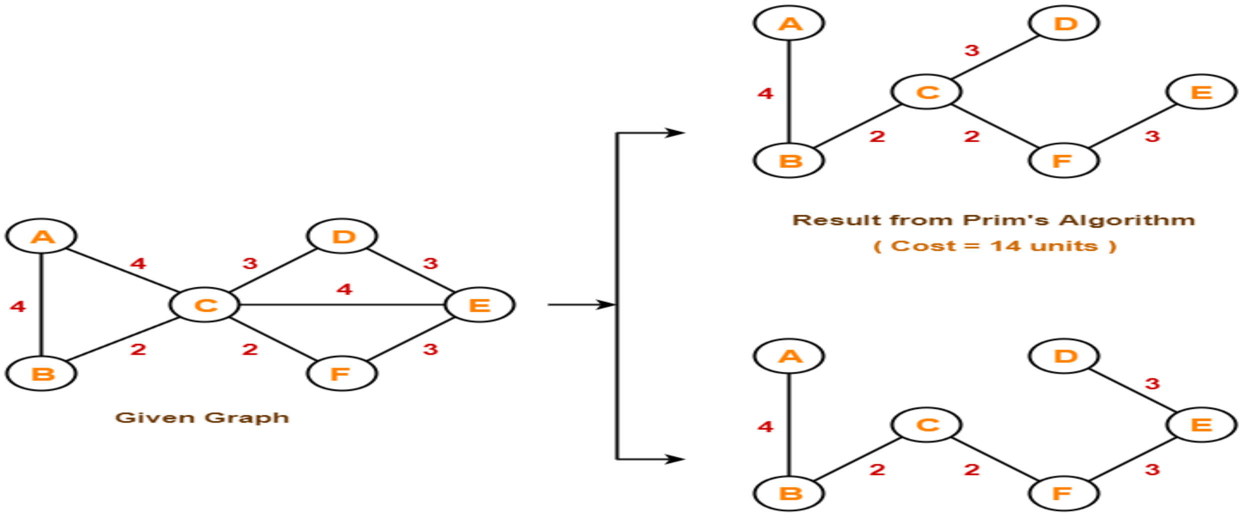
Here, both the algorithms on the above given graph produces the same MST as shown.

2. If all the edge weights are not distinct, then both the algorithms may not always produce the same MST.

- However, cost of both the MST$_s$ would always be same in both the cases.

Example-

Consider the following example-



Given Graph

Result from Prim's Algorithm
(Cost = 14 units)

Result from Kruskal's Algorithm
(Cost = 14 units)

3. Kruskal’s Algorithm is preferred when-

- The graph is sparse.
- There are less number of edges in the graph like E = O(V)
- The edges are already sorted or can be sorted in linear time.

Prim’s Algorithm is preferred when-

- The graph is dense.
- There are large number of edges in the graph like E = O(V2).

4.Difference between Prim’s Algorithm and Kruskal’s Algorithm-

Prim’s Algorithm	**Kruskal’s Algorithm**
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim’s Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal’s Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim’s Algorithm is faster for dense graphs.	Kruskal’s Algorithm is faster for sparse graphs.

Definition

This algorithm was created and published by Dr. Edsger W. Dijkstra, a brilliant Dutch computer scientist and software engineer.

The Dijkstra’s algorithm finds the shortest path from a particular node, called the source node to every other node in a connected graph. It produces a shortest path tree with the source node as the root. It is profoundly used in computer networks to generate optimal routes with the aim of minimizing routing costs.

Basics of Dijkstra's Algorithm

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Dijkstra's Algorithm

Input − A graph representing the network; and a source node, s

Output − A shortest path tree, spt[], with s as the root node.

1. Select the source node also called the initial node

2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.

3. Label the initial node with , and insert it into N.

4. Repeat Steps 5 to 7 until the destination node is inNor there are no more labelled nodes in N.

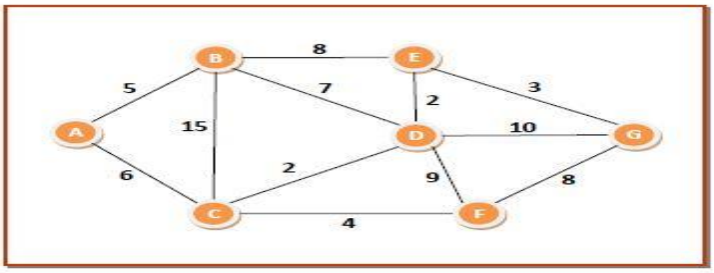
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.

6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.

(b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)

7. Pick a node not in N that has the smallest label assigned to it and add it to N.

Example



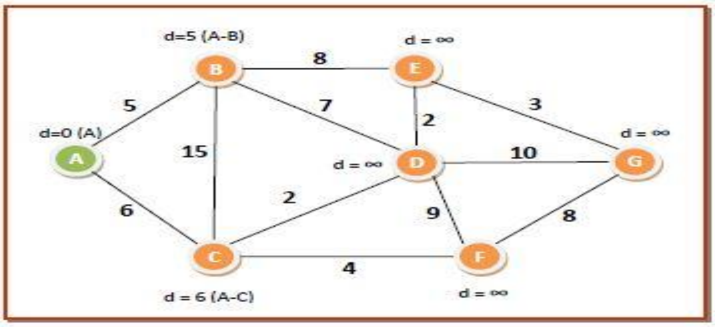
The initializations will be as follows −

- dist[7]={0,∞,∞,∞,∞,∞,∞}
- Q={A,B,C,D,E,F,G}
- S*S*= ∅

Pass 1 − We choose node **A** from **Q** since it has the lowest **dist[]** value of ***0*** and put it in S. The neighbouring nodes of A are B and C. We update dist[] values corresponding to B and C according to the algorithm. So the values of the data structures become −

- dist[7]={0,5,6,∞,∞,∞,∞}
- Q={B,C,D,E,F,G}
- S={A}

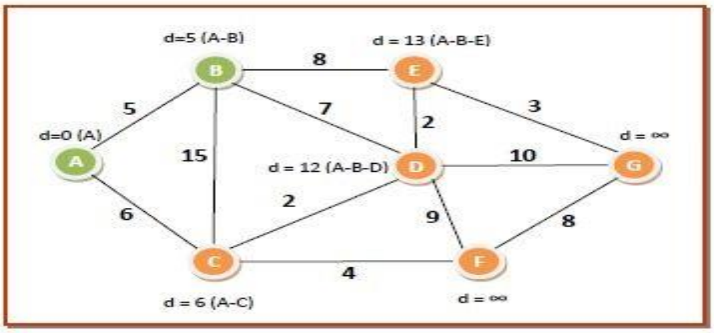
The distances and shortest paths after this pass are shown in the following graph. The green node denotes the node already added to S −



Pass 2 − We choose node **B** from **Q** since it has the lowest **dist[]** value of **5** and put it in **S.** The neighbouring nodes of B are **C, D** and **E**. We update dist[] values corresponding to **C, D** and E according to the algorithm. So the values of the data structures become −

- dist[7]={0,5,6,12,13,∞,∞}
- Q={C,D,E,F,G}
- S={A,B}

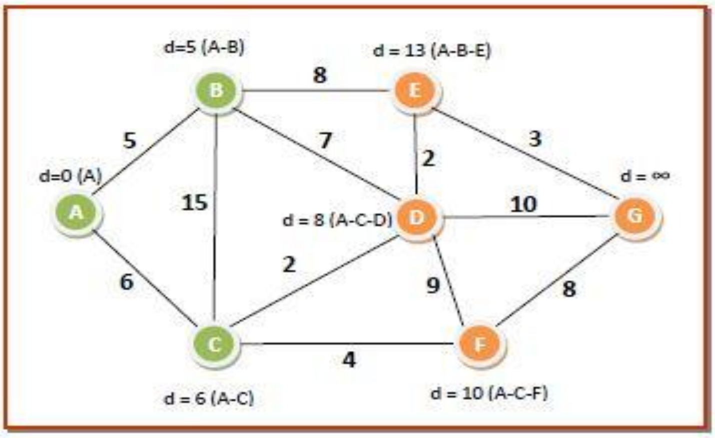
The distances and shortest paths after this pass are −



Pass 3 − We choose node **C** from **Q** since it has the lowest dist[] value of 6 and put it in S. The neighbouring nodes of C are D and F. We update dist[] values corresponding to D and F. So the values of the data structures become −

- dist[7]={0,5,6,8,13,10,∞}
- Q={D,E,F,G}
- S={A,B,C}

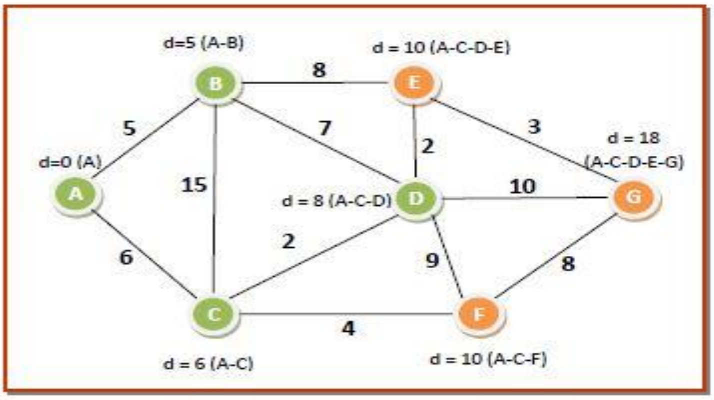
The distances and shortest paths after this pass are –



Pass 4 − We choose node **D** from **Q** since it has the lowest **dist[]** value of 8 and put it in S. The neighbouring nodes of D are E, F and G. We update **dist[]** values corresponding to **E, F** and **G**. So the values of the data structures become −

- dist[7]={0,5,6,8,10,10,18}
- Q={E,F,G}
- S={A,B,C,D}

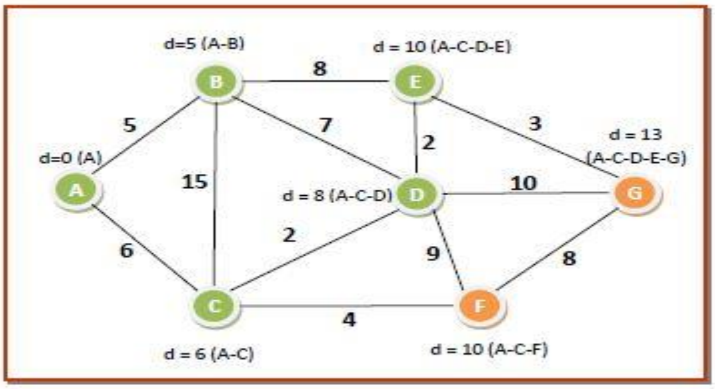
The distances and shortest paths after this pass are –



Pass 5 − We can choose either node E or node **F** from **Q** since both of them have the lowest **dist[]** value of ***10***. We select any one of them, say **E**, and put it in **S**. The neighbouring nodes of **D** is **G**. We update **dist[]** values corresponding to G. So the values of the data structures become −

- dist[7]={0,5,6,8,10,10,13}
- Q={F,G}
- S={A,B,C,D,E}

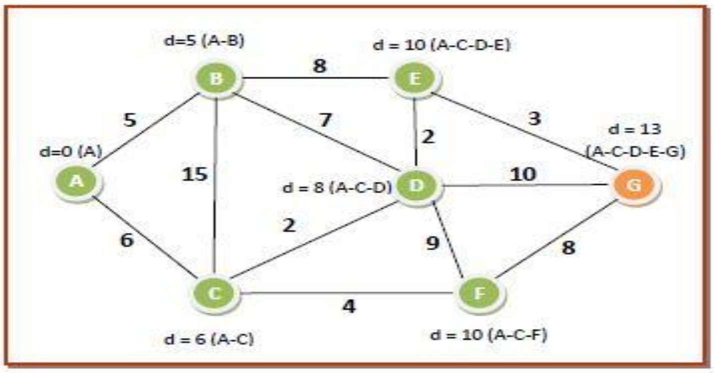
The distances and shortest paths after this pass are –



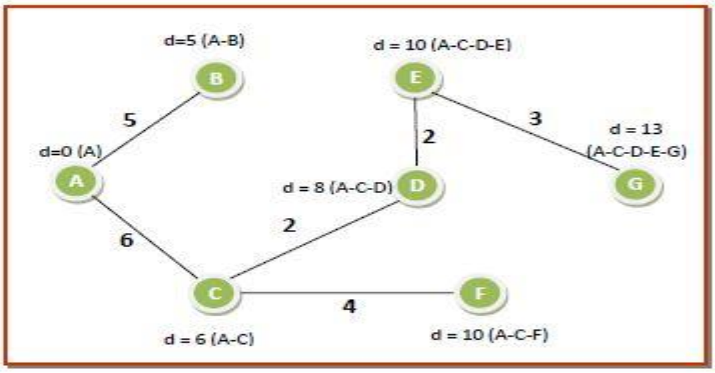
Pass 6 − We choose node **F** from **Q** since it has the lowest **dist[]** value of **10** and put it in **S**. The neighbouring nodes of **F** is **G**. The **dist[]** value corresponding to G is less than that through **F**. So it remains same. The values of the data structures become −

- dist[7]={0,5,6,8,10,10,13}
- Q={G}
- S={A,B,C,D,E,F}

The distances and shortest paths after this pass are –



Pass 7 − There is just one node in **Q**. We remove it from **Q** put it in S. The dist[] array needs no change. Now, **Q** becomes empty, **S** contains all the nodes and so we come to the end of the algorithm. We eliminate all the edges or routes that are not in the path of any route. So the shortest path tree from source node A to all other nodes are as follows −



Transitive Closure of a Directed Graph

A transitive closure of a graph is constructed to answer reachability questions

Definition

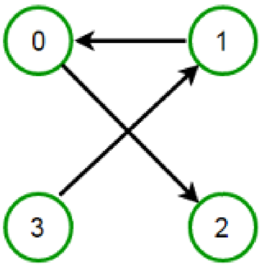
For a directed graph G = (V,E), where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph G* = (V,E*). In G*, for every vertex pair v, w in V there is an edge (v, w) in E* if and only if there is a valid path from v to w in G.

Where and Why is it Needed? Finding the transitive closure of a directed graph is an important problem in the following computational tasks

- Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallel systems.
- It is used in the construction of parsing automata in compiler construction
- Recently, transitive closure computation is being used to evaluate recursive database queries

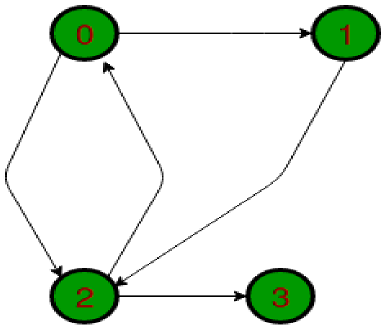
Algorithm

In order to determine the transitive closure of a graph, we define a matrix t where tk ij = 1, for i, j, k = 1, 2, 3, ... n if there exists a path in G from the vertex i to vertex j with intermediate vertices in the set (1, 2, 3, ..., k) and 0 otherwise. That is, G* is constructed by adding an edge (i, j) into E* if and only if tk ij = 1



Its connectivity matrix C is

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline & 1 & 0 & 1 & 0 \\ & 1 & 1 & 1 & 0 \\ & 0 & 0 & 1 & 0 \\ & 1 & 1 & 1 & 1 \end{array}$$



```
Transitive closure of graphs is

     1 1 1 1

     1 1 1 1

     1 1 1 1

     0 0 0 1
```

Floyd Warshall Algorithm-

- Floyd Warshall Algorithm is a famous algorithm.
- It is used to solve All Pairs Shortest Path Problem.
- It computes the shortest path between every pair of vertices of the given graph.
- Floyd Warshall Algorithm is an example of dynamic programming approach.

Advantages-

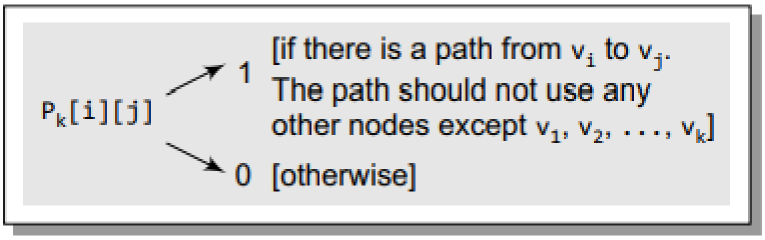
Floyd Warshall Algorithm has the following main advantages-

- It is extremely simple.
- It is easy to implement.

When Floyd Warshall Algorithm Is Used?

Floyd Warshall Algorithm is best suited for dense graphs.

- This is because its complexity depends only on the number of vertices in the given graph.
- For sparse graphs, Johnson’s Algorithm is more suitable.



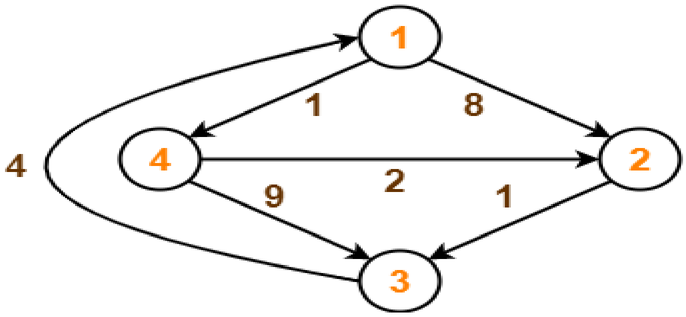
$P_k[i][j]$ → 1 [if there is a path from v_i to v_j. The path should not use any other nodes except $v_1, v_2, \ldots, v_k$]

$P_k[i][j]$ → 0 [otherwise]

Path matrix entry

Problem-

Consider the following directed weighted graph-



Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

Solution-

Step-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

Step-02:

- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

Step-03: Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array}$$

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

$$D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

The last matrix D4 represents the shortest path distance between every pair of vertices.

UNIT – V

SEARCHING

DEFINITION

It is a method of finding the given element in the given list of elements.

or

It is technique to find the location where the element is available or present.

or

It is an algorithm to check whether a particular element is present in the given list or not.

Types of Searching

- ✓ **Linear Search**
- ✓ **Binary Search**
- ✓ **Fibonacci Search**

LINEAR SEARCH

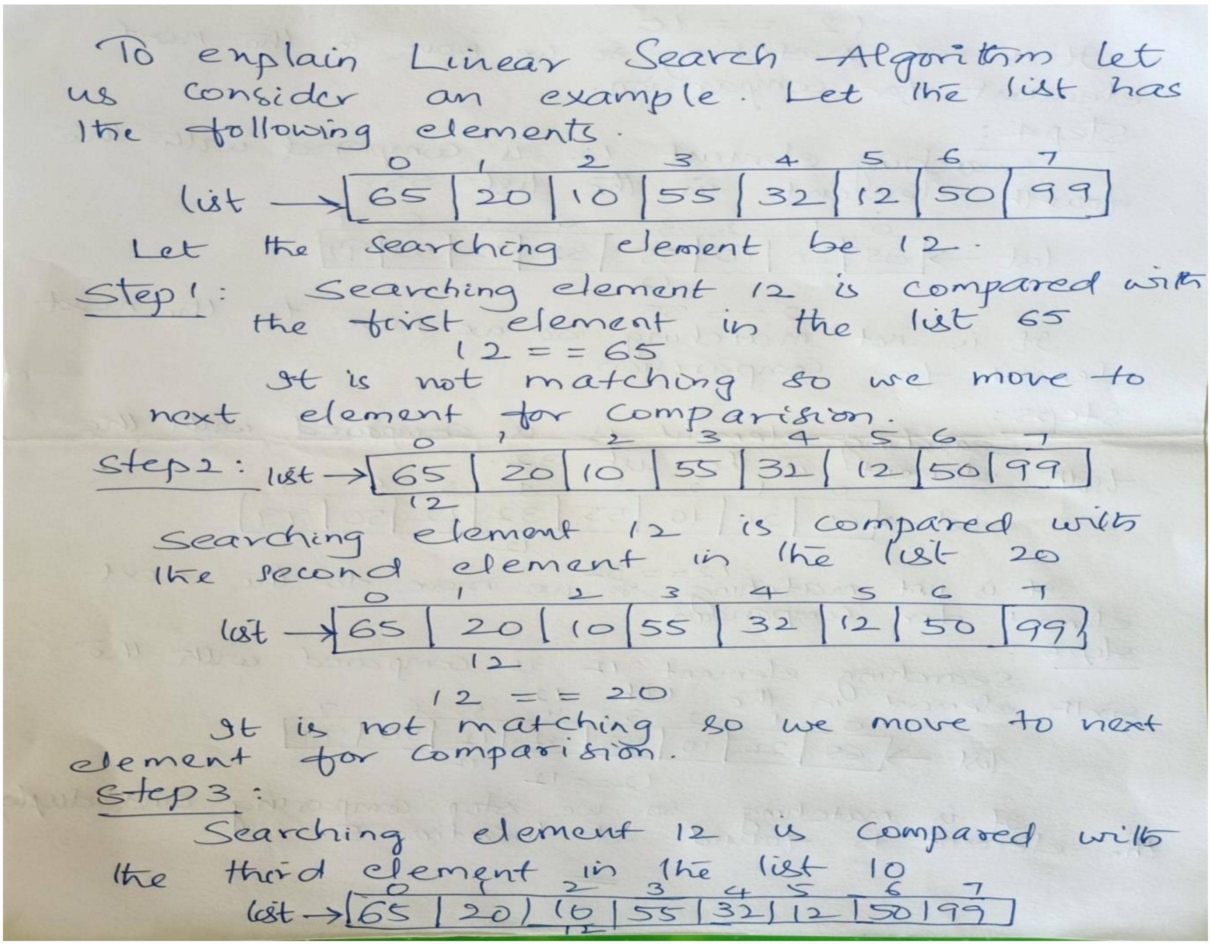
- ✓ It is a very **simple** search algorithm when compared with the other two search algorithms.
- ✓ It is also called as **sequential search** or **indexed search**.
- ✓ To perform linear search, the list of elements **need not be sorted**.
- ✓ An ordered or unordered list will be searched by comparing the search element with one by one element from the beginning of the list until the desired element is found or till the end of the list.
- ✓ If the desired element is found in the list then the search is **successful** otherwise **unsuccessful**.
- ✓ The **time complexity** for linear search is **O(n)** where n is the number of elements in the list.
- ✓ The time complexity increases with the increase of the input size **n**.

Algorithm for Linear Search

LINEAR_SEARCH(A, N, KEY)

```
Step 1: SET POS = -1
Step 2: SET I = 1
Step 3: Repeat Step 4 while I<=N
Step 4: IF A[I] = KEY
                SET POS = I
                PRINT POS
                Go to Step 6
        SET I = I + 1
Step 5: IF POS = –1
                PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
Step 6: EXIT
```

Example of Linear Search



To explain Linear Search Algorithm let us consider an example. Let the list has the following elements.

list →

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

Let the searching element be 12.

Step 1: Searching element 12 is compared with the first element in the list 65

12 == 65

It is not matching so we move to next element for comparision.

Step 2: list →

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99
12							

Searching element 12 is compared with the second element in the list 20

list →

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99
	12						

12 == 20

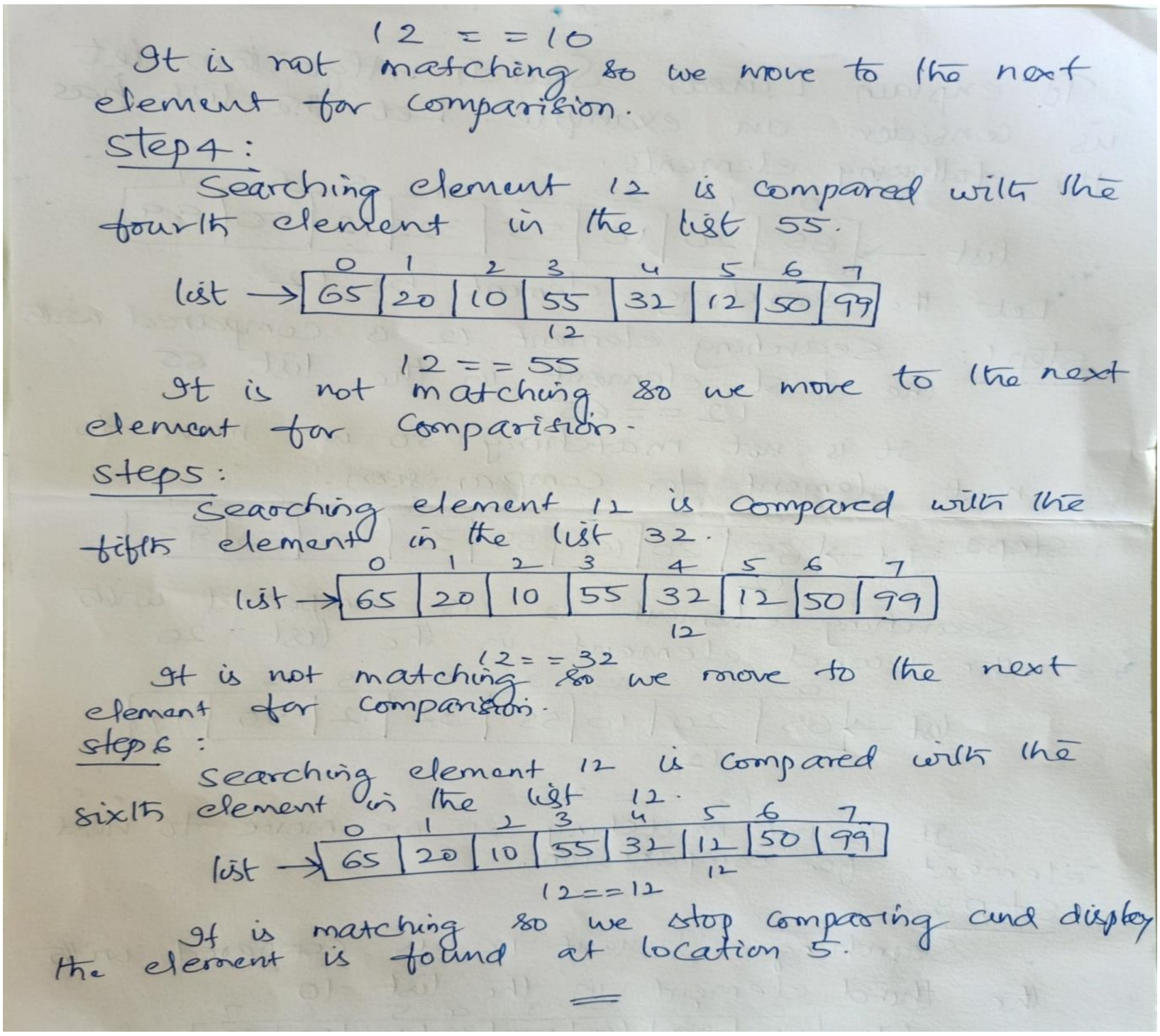
It is not matching so we move to next element for comparision.

Step 3:

Searching element 12 is compared with the third element in the list 10

list →

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99
		12					



12 = = 10

It is not matching so we move to the next element for comparision.

Step4:

Searching element 12 is compared with the fourth element in the list 55.

list →

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99
			12				

12 = = 55

It is not matching so we move to the next element for comparision.

steps:

Searching element 12 is compared with the fifth element in the list 32.

list →

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99
				12			

12 = = 32

It is not matching so we move to the next element for comparision.

step6:

Searching element 12 is compared with the sixth element in the list 12.

list →

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99
					12		

12==12

It is matching so we stop comparing and display the element is found at location 5.

Program for Linear Search

```
#include<stdio.h>
int main(void)
{
        int a[20], n, i, key;
        printf("Enter size of the list: ");
        scanf("%d", &n);
        printf("Enter the elements");
        for(i = 0; i < n; i++)
                scanf("%d", &a[i]);
        printf("Enter the element to be Search: ");
        scanf("%key);
for(i = 0; i < n; i++)
        {
                if(key == a[i])
                {
                        printf("Element is found at %d index", i);
                        break;
                }
        }
        if(i == n)
                printf("Given element is not found in the li st!!!");
        return 0;
}
```

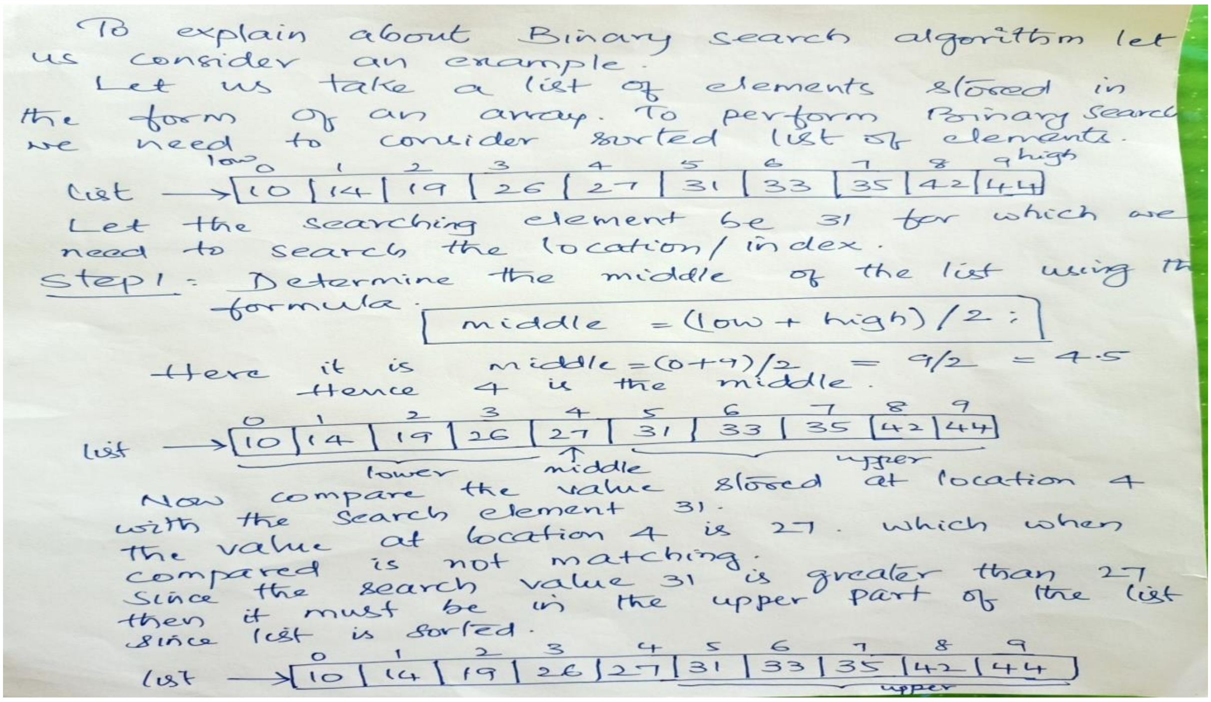
BINARY SEARCH

- ✓ It is the **fastest** searching algorithm when compared with the other two algorithms.
- ✓ It works on the principle **divide – conquer** strategy.
- ✓ To apply binary search algorithm the list of elements should be in **sorted order**.
- ✓ The time complexity for binary search algorithm is **O(log n)**.
- ✓ It is applied to **very large set** of elements
- ✓ The process carried by binary search algorithm is find the middle element and compare it with search element it match return the index of the element and say success otherwise see if the search element is greater than or less than the middle element.
- ✓ If it is greater than the middle element then search the element in the upper part of the list otherwise in the lower part of the list.
- ✓ Again find middle element and do the same process till the element is found or not found.
- ✓ Using binary search algorithm we can reduce the number of comparisons hence it is best.

Algorithm forBinary Search

```
BINARY_SEARCH(A, lower_bound, upper_bound, KEY)
Step 1: SET BEG = lower_bound
            END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3: SET MID = (BEG + END)/2
Step 4: IF A[MID] = KEY
               SET POS = MID
               PRINT POS
               Go to Step 6
          ELSE IF A[MID] > VAL
                    SET END = MID - 1
               ELSE
                    SET BEG = MID + 1
Step 5: IF POS = -1
               PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
Step 6: EXIT
```

Example of Binary Search



To explain about Binary search algorithm let us consider an example.

Let us take a list of elements stored in the form of an array. To perform Binary search we need to consider sorted list of elements.

list →

low 0	1	2	3	4	5	6	7	8	9 high
10	14	19	26	27	31	33	35	42	44

Let the searching element be 31 for which we need to search the location/index.

Step1: Determine the middle of the list using th formula.

$\boxed{\text{middle} = (\text{low} + \text{high})/2;}$

Here it is middle $=(0+9)/2 = 9/2 = 4.5$

Hence 4 is the middle.

list →

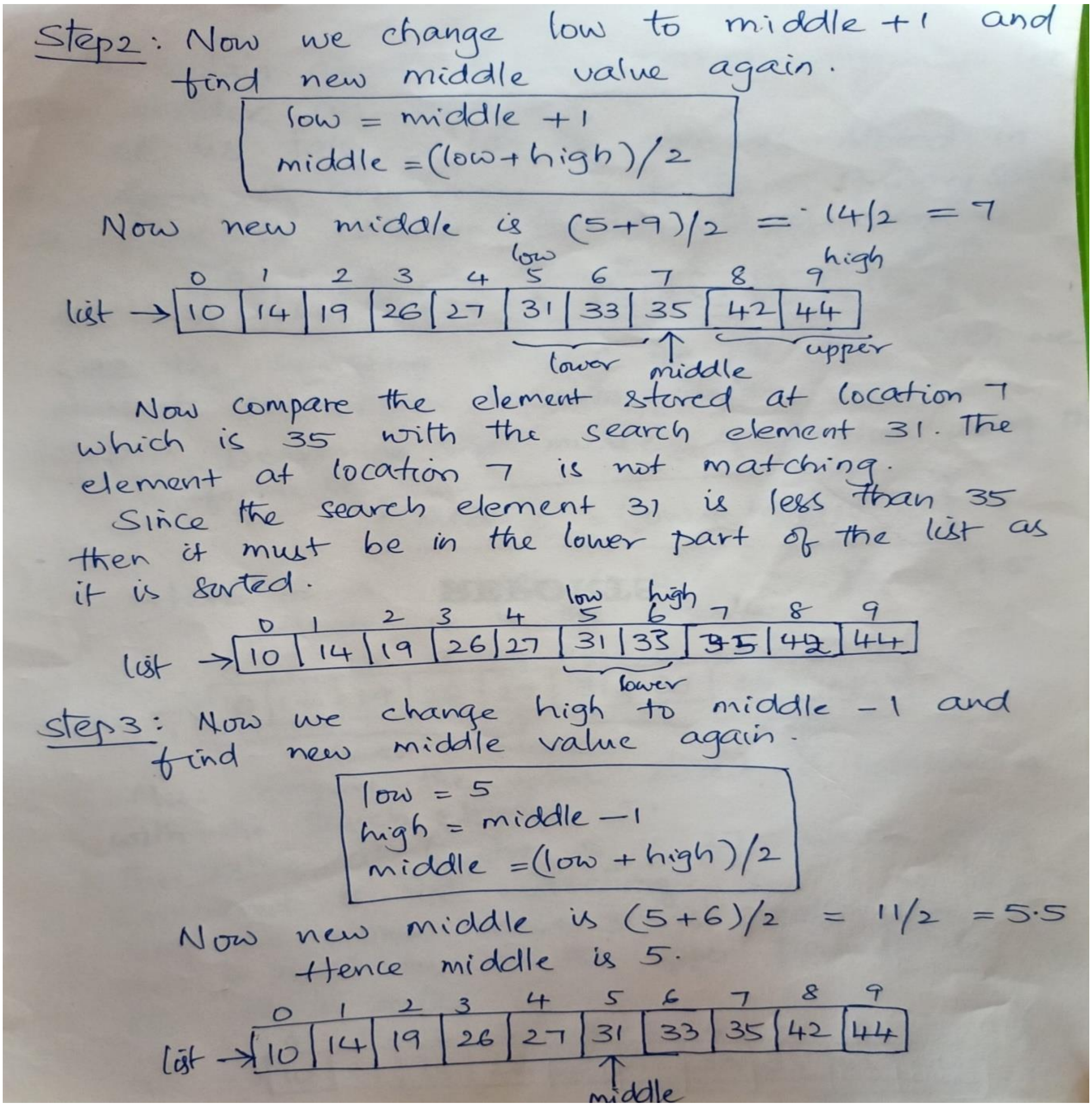
0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44
lower				↑ middle	upper				

Now compare the value stored at location 4 with the search element 31. which when the value at location 4 is 27. compared is not matching.

Since the search value 31 is greater than 27 then it must be in the upper part of the list since list is sorted.

list →

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44
					upper				



Step2 : Now we change low to middle + 1 and find new middle value again.

$$\boxed{\begin{array}{l} low = middle + 1 \\ middle = (low + high)/2 \end{array}}$$

Now new middle is $(5+9)/2 = 14/2 = 7$

	0	1	2	3	4	5 low	6	7	8	9 high
list →	10	14	19	26	27	31	33	35	42	44
						lower	lower	↑ middle	upper	upper

Now compare the element stored at location 7 which is 35 with the search element 31. The element at location 7 is not matching.

Since the search element 31 is less than 35 then it must be in the lower part of the list as it is sorted.

	0	1	2	3	4	5 low	6 high	7	8	9
list →	10	14	19	26	27	31	33	~~35~~	~~42~~	44
						lower	lower			

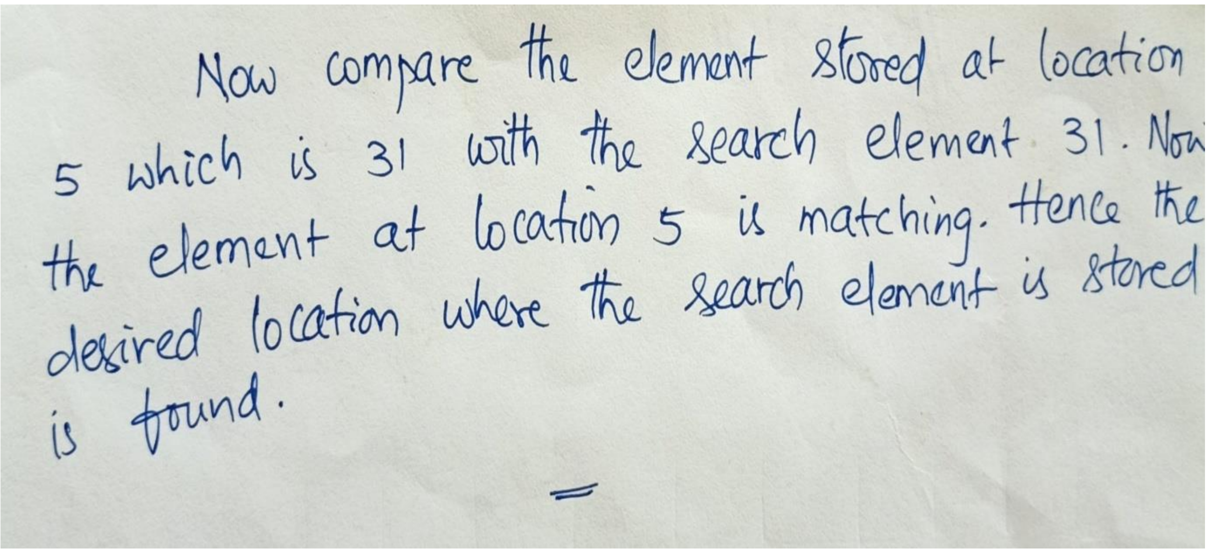
Step3 : Now we change high to middle − 1 and find new middle value again.

$$\boxed{\begin{array}{l} low = 5 \\ high = middle - 1 \\ middle = (low + high)/2 \end{array}}$$

Now new middle is $(5+6)/2 = 11/2 = 5.5$

Hence middle is 5.

	0	1	2	3	4	5	6	7	8	9
list →	10	14	19	26	27	31	33	35	42	44
						↑ middle				



Now compare the element stored at location 5 which is 31 with the search element 31. Now the element at location 5 is matching. Hence the desired location where the search element is stored is found.

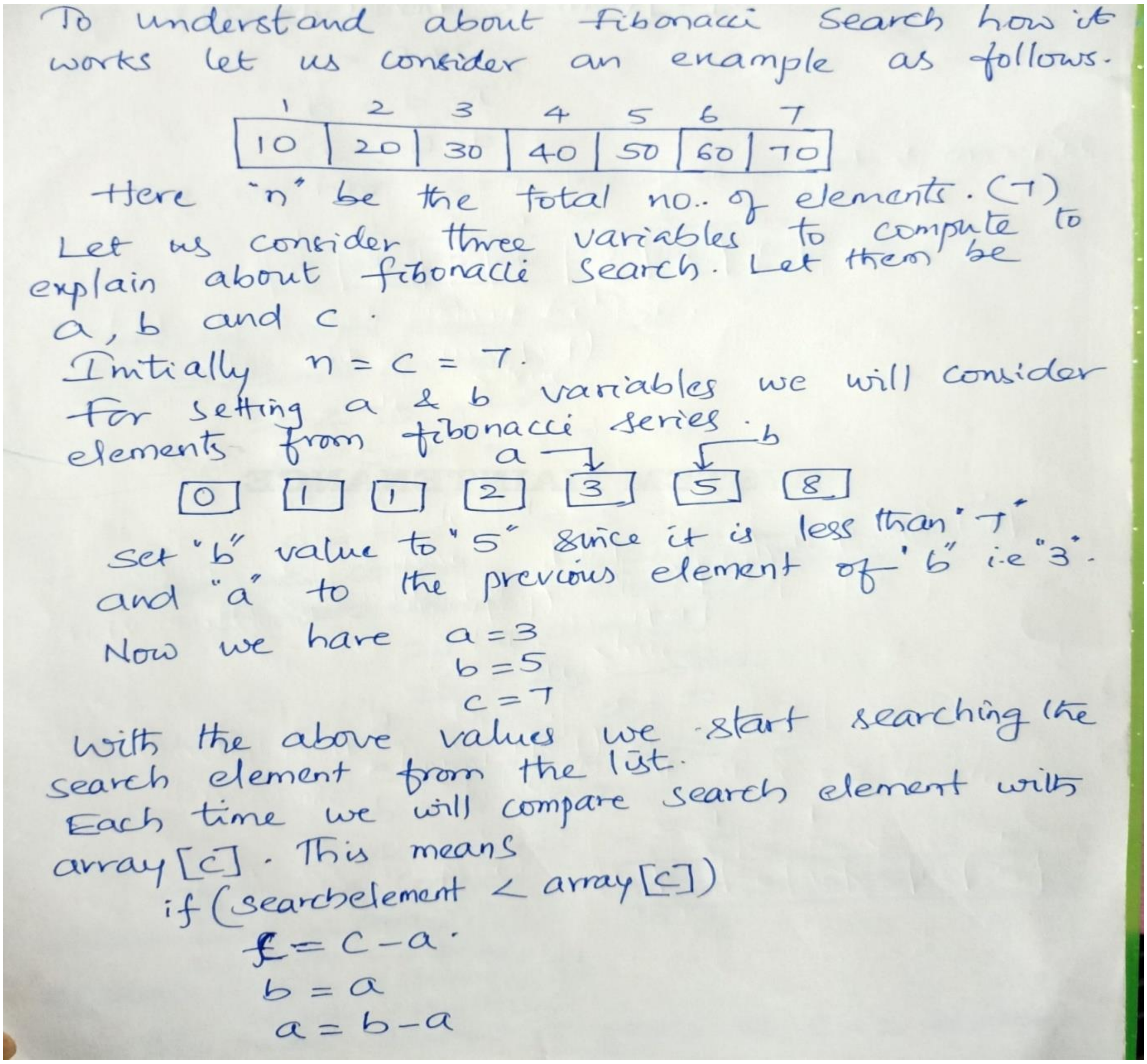
Program for Binary Search

```
#include <stdio.h>
int main(void)
{
        int i, low, high, middle, n, key, a[10];
        printf("Enter number of elements");
        scanf("%d", &n);
        printf("Enter the elements”);
        for (i = 0; i < n; i++)
                scanf("%d", &a[i]);
        printf("Enter value to find");
        scanf("%d", &key);
        low = 0;
        high = n - 1;
        middle = (low + high)/2;
        while(low <= high)
        {
                if(a[middle] < key)
                        low = middle + 1;
                else if(a[middle] == key)
                {
                        printf(“Element is found”);
                        break;
                }
                else
                        high = middle - 1;
                middle = (low + high)/2;
        }
        if(low > high)
                printf(“Element is not found” );
        return 0;
    }
```

FIBONACCI SEARCH

- ✓ It was developed by Kiefer in 1953.
- ✓ In Fibonacci search we consider the indices as numbers from fibonacci series.
- ✓ To apply fibonacci search algorithm the list that contains elements should be in sorted order.
- ✓ The time complexity of fibonacci search algorithm is **O(log n)**
- ✓ It works on the principle **divide - conquer** strategy.

Example of Fibonacci Search



To understand about Fibonacci Search how it works let us consider an example as follows.

1	2	3	4	5	6	7
10	20	30	40	50	60	70

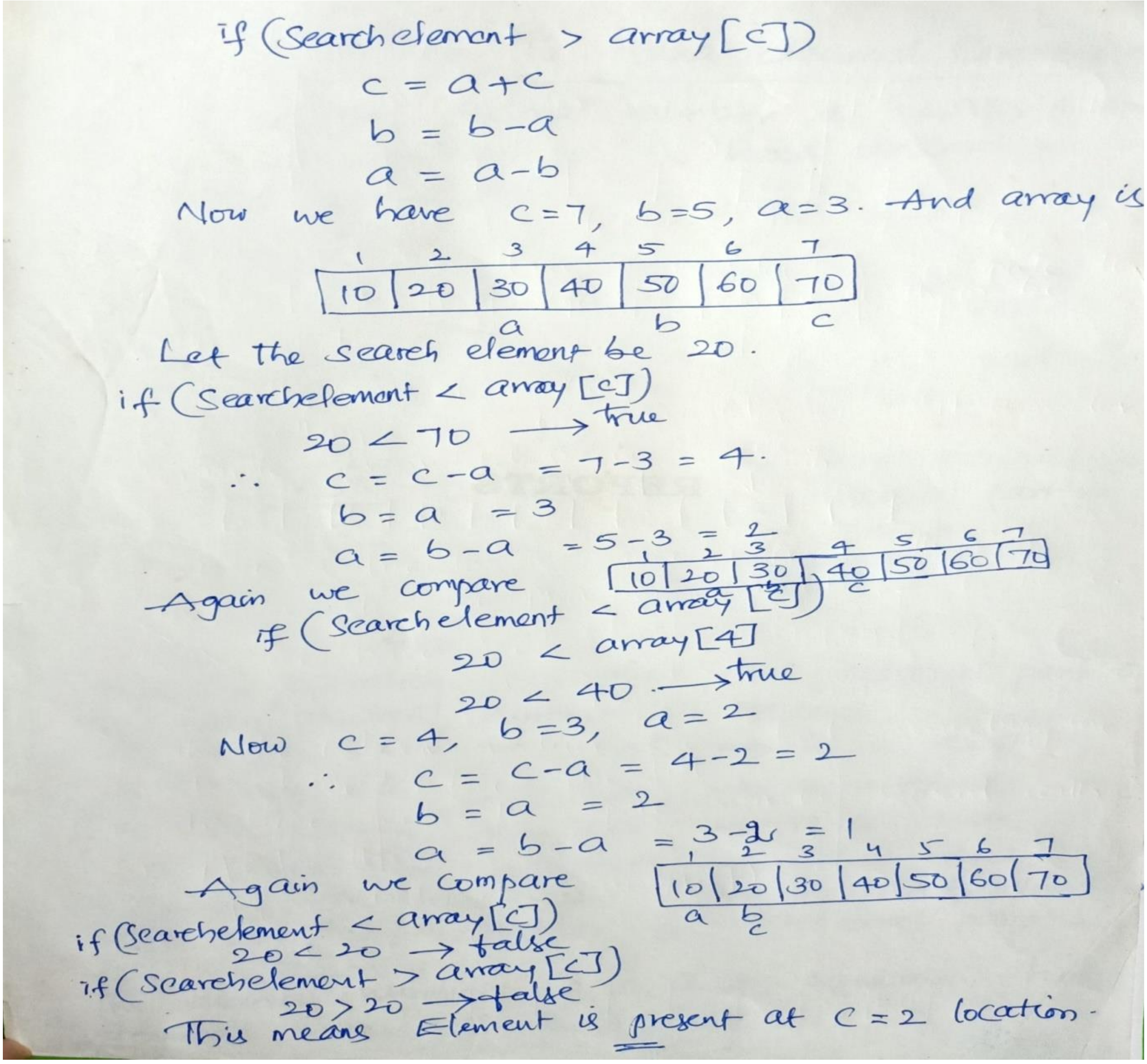
Here "n" be the total no.. of elements. (7)
Let us consider three variables to compute to explain about fibonacci search. Let them be a, b and c.
Initially n = c = 7.
For setting a & b variables we will consider elements from fibonacci series.

				a	b	
0	1	1	2	3	5	8

set "b" value to "5" since it is less than "7" and "a" to the previous element of "b" i.e "3".
Now we have a = 3
b = 5
c = 7

With the above values we start searching the search element from the list.
Each time we will compare search element with array[c]. This means

```
if (searchelement < array[c])
    c = c - a.
    b = a
    a = b - a
```



if (Searchelement > array[c])

c = a+c

b = b-a

a = a-b

Now we have c=7, b=5, a=3. And array is

1	2	3	4	5	6	7
10	20	30	40	50	60	70
		a		b		c

Let the search element be 20.

if (Searchelement < array[c])

20 < 70 → true

∴ c = c-a = 7-3 = 4.

b = a = 3

a = b-a = 5-3 = 2

1	2	3	4	5	6	7
10	20	30	40	50	60	70
	a	b	c			

Again we compare

if (Searchelement < array[c])

20 < array[4]

20 < 40 → true

Now c = 4, b=3, a=2

∴ c = c-a = 4-2 = 2

b = a = 2

a = b-a = 3-2 = 1

1	2	3	4	5	6	7
10	20	30	40	50	60	70
a	b c					

Again we compare

if (Searchelement < array[c])

20 < 20 → false

if (Searchelement > array[c])

20 > 20 → false

This means Element is present at c = 2 location.

Program for Fibonacci Search

```
#include<stdio.h>
int main(void)
{
        int n, key, i, ar[20];
        void search(int ar[], int n, int key, int f, int a, int b);
        int fib(int n);
        clrscr();
        printf("\n Enter the number of elements in array");
        scanf("%d", &n);
        printf("\n Enter the elements");
        for(i=0;i<n;i++)
```

```
                scanf("%d", &ar[i]);
        printf("Enter the element to be searched");
        scanf("%d", &key);
        search(ar, n, key, n, fib(n), fib(fib(n)));
        return 0;
}
int fib(int n)
{
        int a, b, f;
        if(n<1)
                return n;
        a=0;
        b=1;
        while(b<n)
        {
                f=a+b;
                a=b;
                b=f;
        }
        return a;
}
void search(int ar[], int n, int key, int f, int b, int a)
{
        if(f<1 || f>n)
                printf("the number is not present");
        else if(key<ar[f])
        {
                if(a<=0)
                        printf("The element is not present in the list");
                else
                        search(ar, n, key, f-a, a, b-a);
        }
        else if(key>ar[f])
        {
                if(b<=1)
                        printf("The element is not present in the list");
                else
                        search(ar, n, key, f+a, b-a, a-b);
        }
        else
                printf("Element is present %d", f);
}
```

SORTING

DEFINITION

Sorting is a technique to rearrange the list of elements either in ascending or descending order, which can be numerical, alphabetical or any user-defined order.

Types of Sorting

Internal Sorting

- ✓ If the data to be sorted remains in main memory and also the sorting is carried out in main memory then it is called internal sorting.
- ✓ Internal sorting takes place in the main memory of a computer.
- ✓ The internal sorting methods are applied to small collection of data.
- ✓ The following are some internal sorting techniques:
  - ✓ **Insertion sort**
  - ✓ **Merge Sort**
  - ✓ **Quick Sort**
  - ✓ **Heap Sort**

External Sorting

- ✓ If the data resides in secondary memory and is brought into main memory in blocks for sorting and then result is returned back to secondary memory is called external sorting.
- ✓ External sorting is required when the data being sorted do not fit into the main memory.
- ✓ The following are some external sorting techniques:
  - ✓ **Two-Way External Merge Sort**
  - ✓ **K-way External Merge Sort**

INSERTION SORT

In this method, the elements are inserted at their appropriate place. Hence the name insertion sort.

- ✓ This sorting is very simple to implement.

- ✓ This method is very efficient when we want to sort small number of elements.
- ✓ This method has excellent performance when almost the elements are sorted.
- ✓ It is more efficient than bubble and selection sorts.
- ✓ This sorting is stable.
- ✓ This is an in-place sorting technique.
- ✓ The time complexity of insertion sort for best case is O(n), average case and worst case is O(n^2).

Algorithm for Insertion Sort

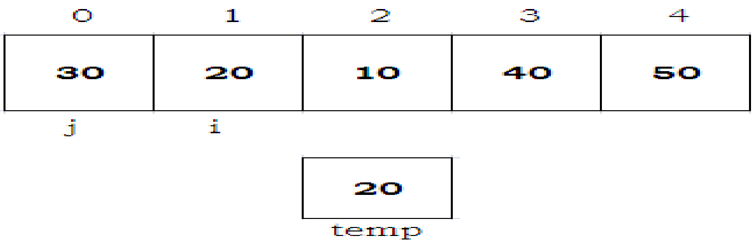
INSERTION-SORT (A, N)

```
Step 1: Repeat Steps 2 to 5 for I = 1 to N – 1
Step 2: SET TEMP = A[I]
Step 3: SET J = I - 1
Step 4: Repeat while TEMP <= A[J]
                    SET A[J + 1] = A[J]
                    SET J = J - 1
Step 5: SET A[J + 1] = TEMP
Step 6: EXIT
```

Example for insertion sort

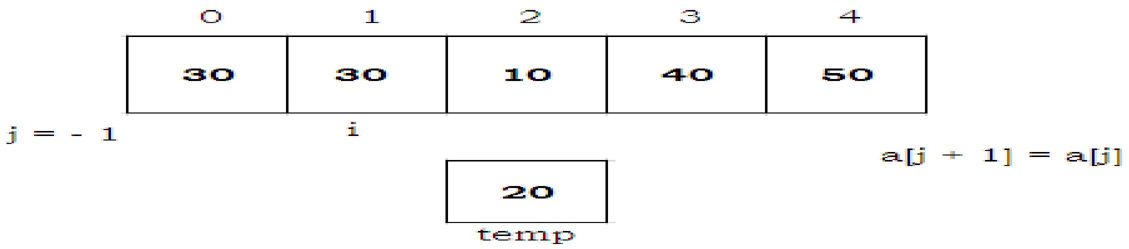
Let us consider the array of elements to sort them using insertion sort technique

30, 20, 10, 40, 50



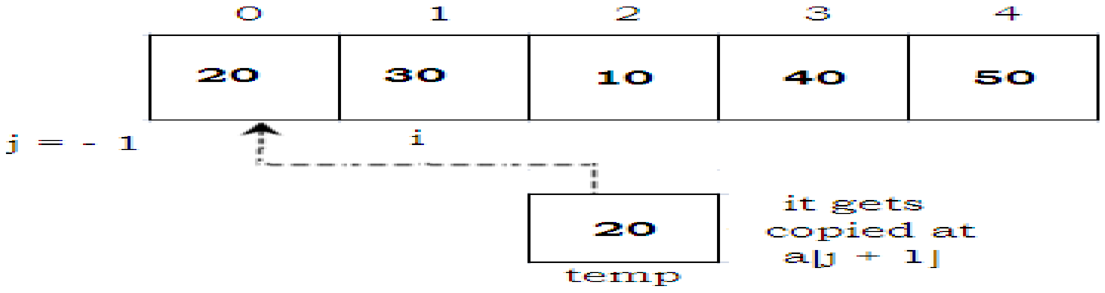
initially it enters into the outer loop temp = a[i], j=i-1

The control moves to while loop as j>=0 and a[j] > temp is true, the while is executed.



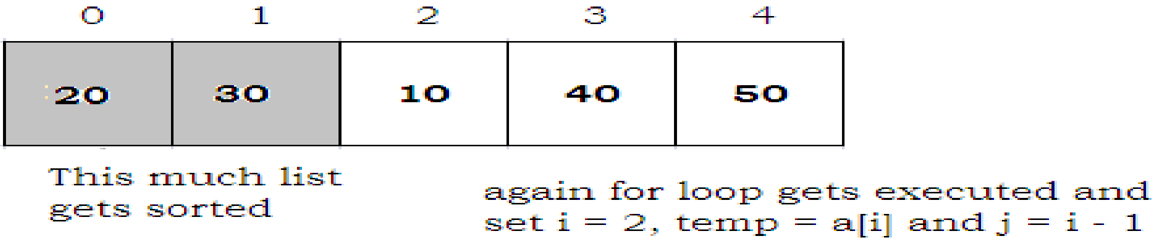
a[j + 1] = a[j]

Now since j >= 0 is false, control comes out of while loop



it gets copied at a[j + 1]

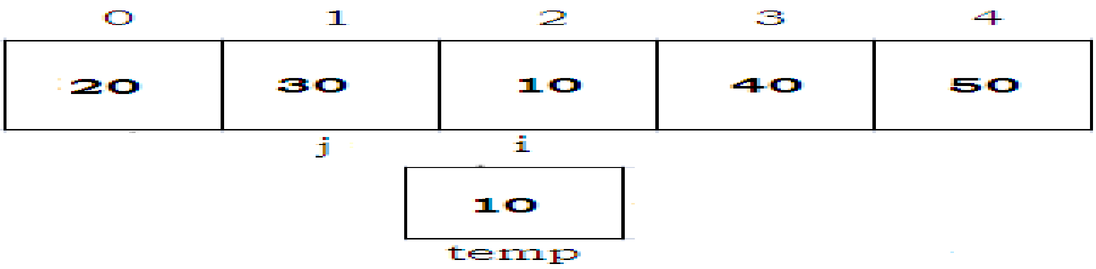
then the list becomes

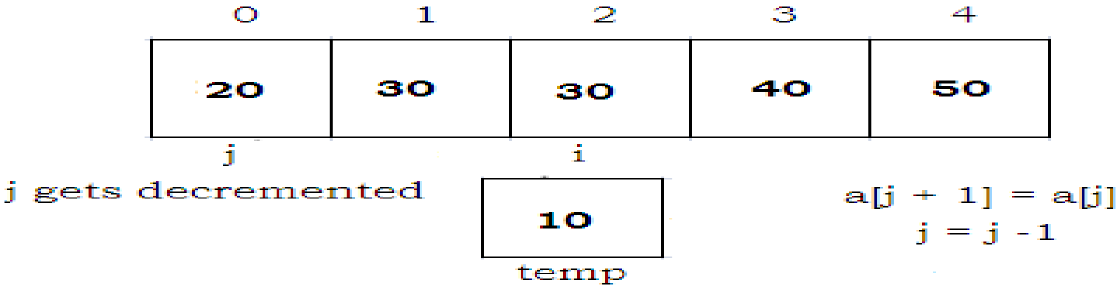


This much list gets sorted

again for loop gets executed and set i = 2, temp = a[i] and j = i - 1

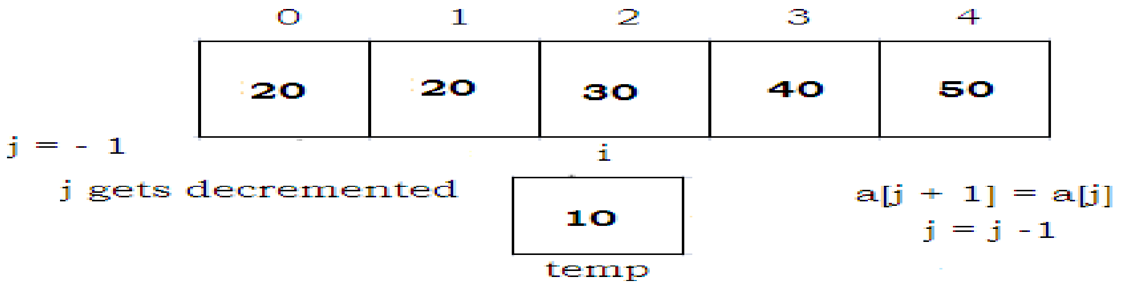
The control moves to while loop as j>=0 and a[j] > temp is true, the while is executed.



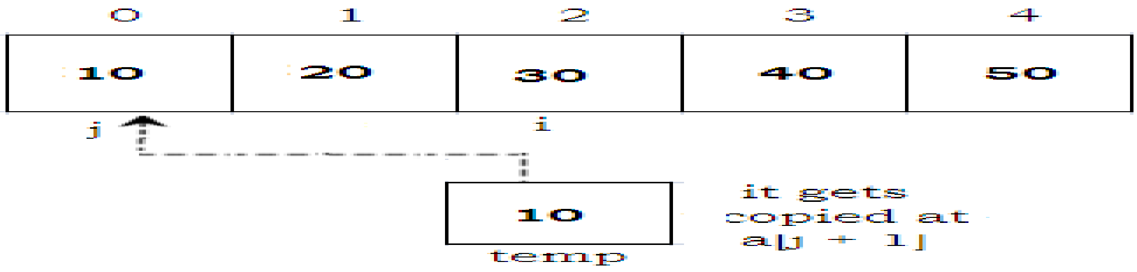


j gets decremented

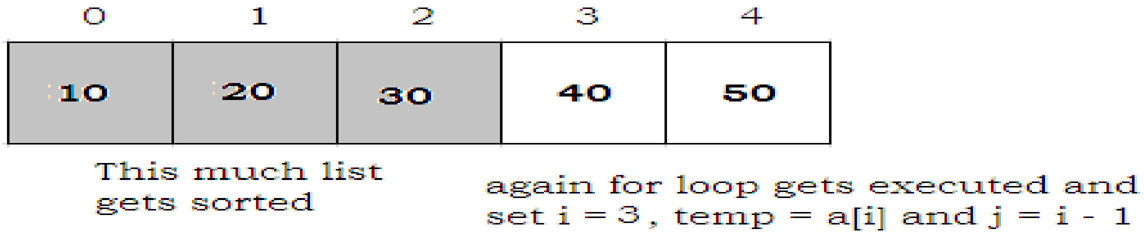
a[j + 1] = a[j]
j = j - 1



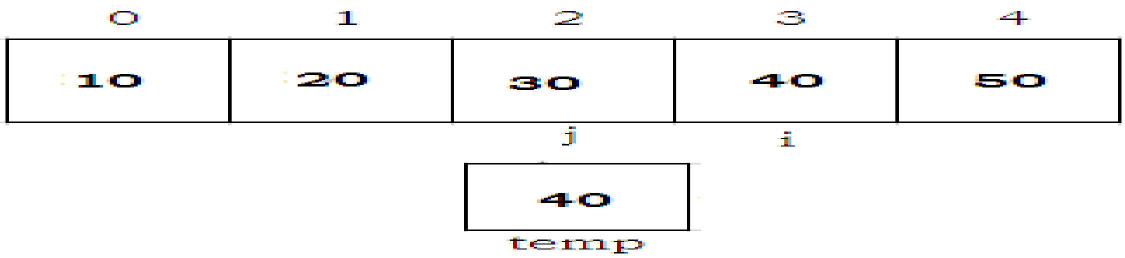
Now since j >= 0 is false, control comes out of while loop

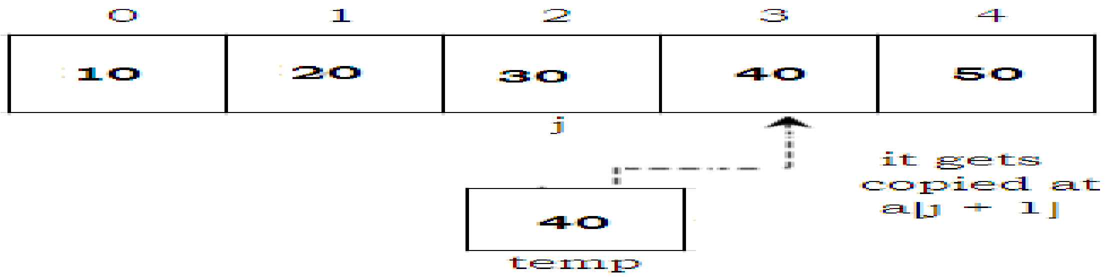


Then the list becomes

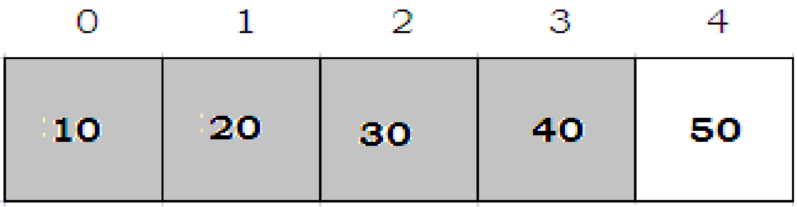

0
1
2
3
4
10
20
30
40
50
This much list
gets sorted
again for loop gets executed and
set i = 3 , temp = a[i] and j = i - 1

The control moves to while loop as j>=0 and a[j] > temp is false, the while is not executed.





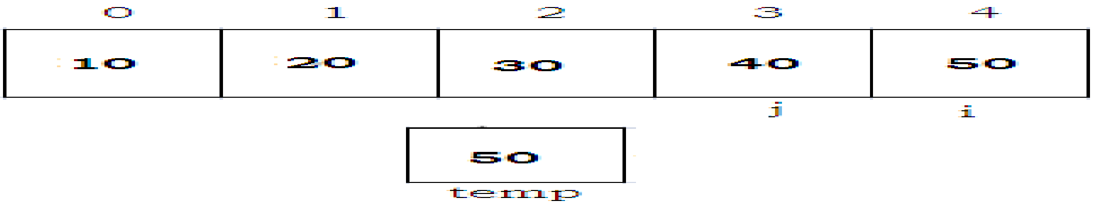
Then the list becomes

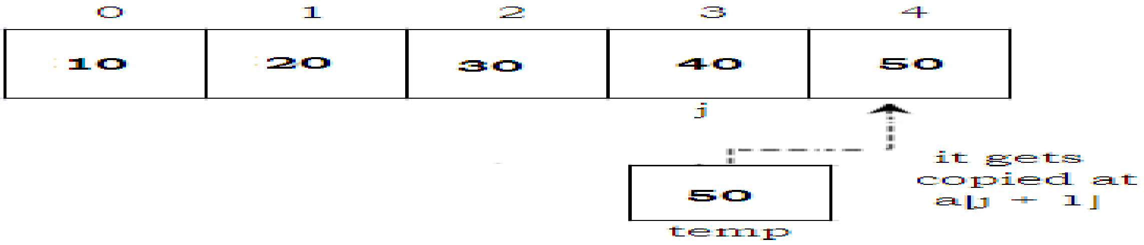


This much list gets sorted

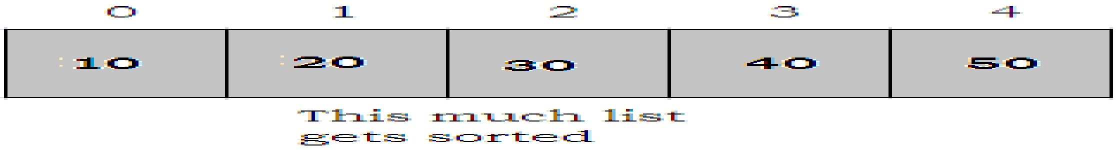
again for loop gets executed and set i = 4, temp = a[i] and j = i - 1

The control moves to while loop as j>=0 and a[j] > temp is false, the while is not executed.





Then the list becomes



Program to illustrate insertion sort technique.

```
#include<stdio.h>
void insert_sort(int [], int);
int main(void)
{
        int n, a[10], i;
        clrscr();
        printf(" Enter the size of the array ");
        scanf("%d", &n);
        printf(" Enter the elements of the array ");
        for(i=0; i<n; i++)
                scanf("%d", &a[i]);
        insert_sort(a, n);
        return 0;
}
void insert_sort(int a[], int n)
{
        int i,j,temp;
        for(i=1; i<n; i++)
        {
                temp = a[i];
                j = i - 1;
                while(j >= 0 && a[j] > temp)
                {
                        a[j+1] = a[j];
                        j = j - 1;
                }
                a[j+1]=temp;
        }
        printf(" \n The sorted list of elements are ");
        for(i=0; i<n; i++)
                printf("%d\t", a[i]);
}
```

SELECTION SORT

- ✓ It is **easy** and **simple** to implement
- ✓ It is used for **small** list of elements
- ✓ It uses less **memory**
- ✓ It is **efficient** than **bubble sort** technique
- ✓ It is **not efficient** when used with **large list** of elements

- ✓ It is **not efficient** than **insertion sort** technique when used with **large** list
- ✓ The time complexity of selection sort is **O(n^2)**
- ✓ Consider an array **A** with **N** elements. First find the **smallest element** in the **array** and place it in the **first position**. Then, find the **second smallest element** in the **array** and place it in the **second position**. Repeat this procedure until the entire array is sorted.
- ✓ In **Pass 1**, find the position **POS** of the **smallest element** in the array and then **swap A[POS]** and **A[0]**. Thus, **A[0]** is sorted.
- ✓ In **Pass 2**, find the position **POS** of the **smallest element** in **sub-array of N–1 elements**. Swap **A[POS]** with **A[1]**. Now, **A[0]** and **A[1]** is sorted.
- ✓ In **Pass N–1**, find the position **POS** of the **smaller** of the elements **A[N–2]** and **A[N–1]**. Swap **A[POS]** and **A[N–2]** so that **A[0], A[1], ..., A[N–1]** is sorted.

Algorithm for Selection Sort

Algorithm for Selection Sort

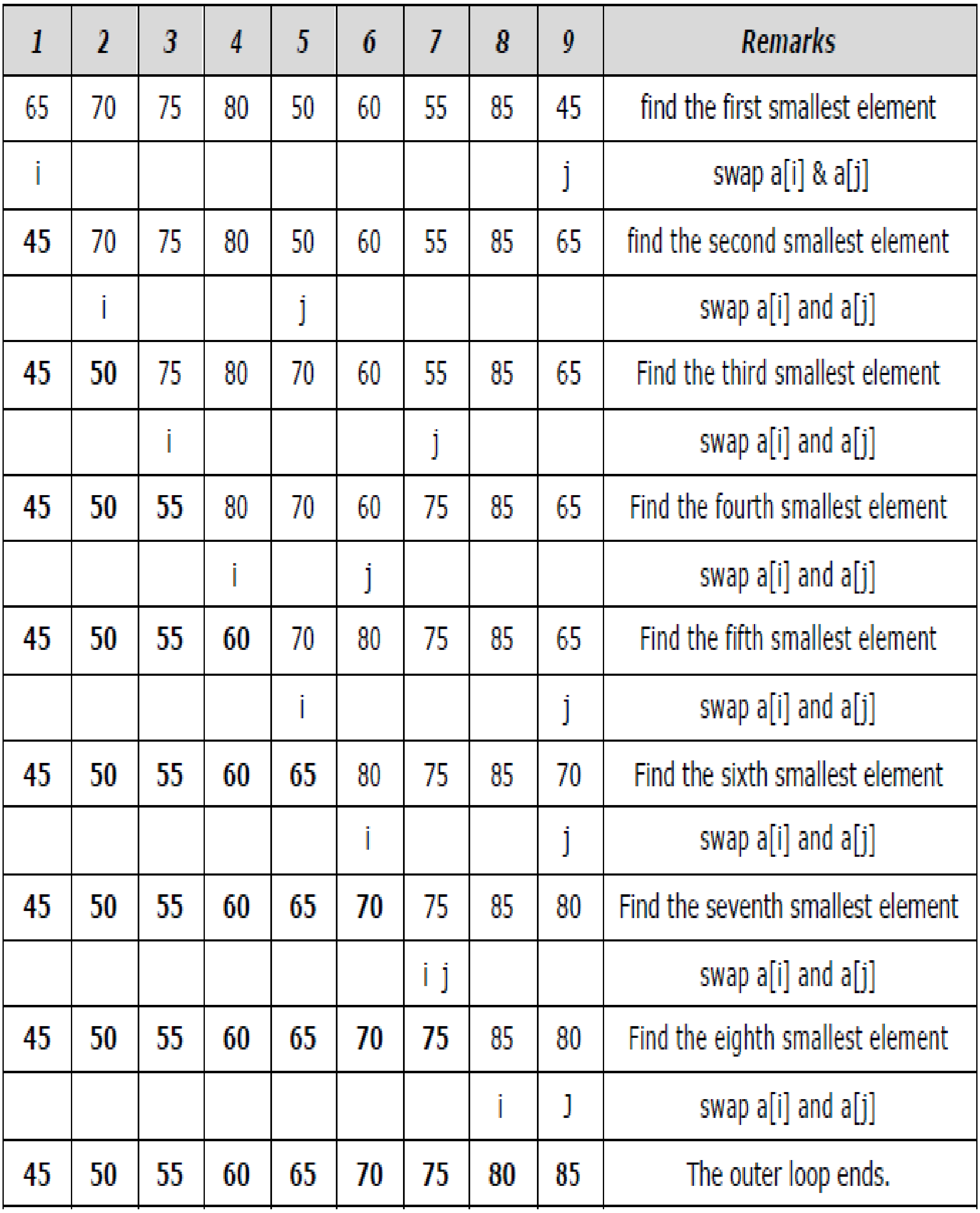
SELECTION SORT(A, N)

```
Step 1: Start
Step 2: Repeat Steps 3 and 4 for I = 1 to N
Step 3: Call SMALLEST(A, I, N, pos)
Step 4: Swap A[I] with A[pos]
Step 5: Stop
```

SMALLEST (A, I, N, pos)

```
Step 1: Start
Step 2: SET small = A[I]
Step 3: SET POS = I
Step 4: Repeat for J = I+1 to N
                If small> A[J]
                        SET small = A[J]
                SET pos = J
Step 4: Return pos
Step 5: Stop
```

Example for Selection Sort



1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

Program for Selection Sort

```
# include<stdio.h>
void selection_sort( int low, int high );
int a[25];
int main(void)
{
        int n, i= 0;
        printf( "Enter the number of elements: " );
        scanf("%d", &n);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < n; i++)
            scanf( "%d", &a[i] );
        selection_sort( 0, n-1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< n; i++ )
            printf( "%d\t ", a[i] );
        return 0;
}
void selection_sort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
            minindex = i;
            for( j=i+1; j <= high; j++ )
            {
                if( a[j] < a[minindex] )
                    minindex = j;
            }
            temp = a[i];
            a[i] = a[minindex];
            a[minindex] = temp;
```

```
          }
}
```

BUBBLE SORT

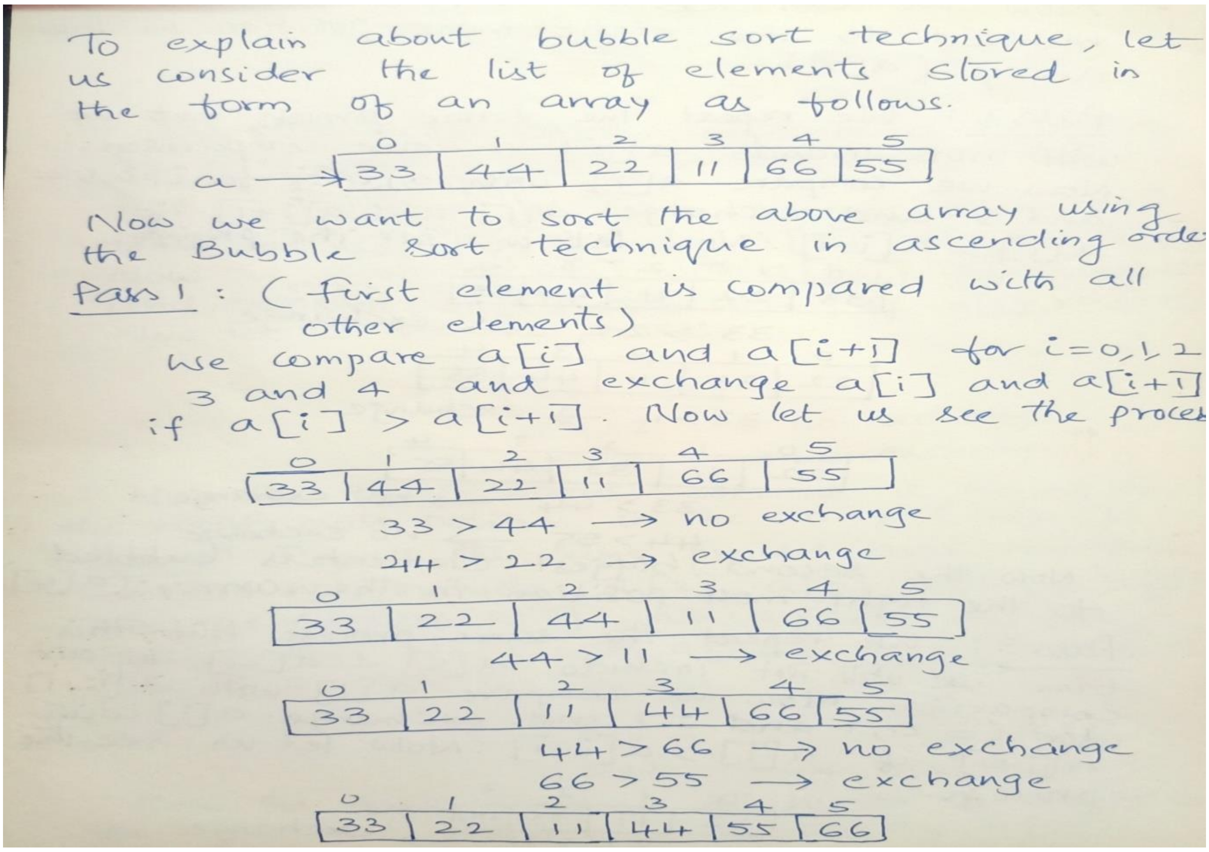
- ✓ It is known as exchange sort
- ✓ It is also known as comparison sort
- ✓ It is easiest and simple sort technique but inefficient.
- ✓ It is not a stable sorting technique.
- ✓ The time complexity of bubble sort is O(n^2) in all cases.
- ✓ Bubble sort uses the concept of passes.
- ✓ The phases in which the elements are moving to acquire their proper positions is called passes.
- ✓ It works by comparing adjacent elements and bubbles the largest element towards right at the end of the first pass.
- ✓ The largest element gets sorted and placed at the end of the sorted list.
- ✓ This process is repeated for all pairs of elements until it moves the largest element to the end of the list in that iteration.
- ✓ Bubble sort consists of (n-1) passes, where n is the number of elements to be sorted.
- ✓ In 1st pass the largest element will be placed in the nth position.
- ✓ In 2nd pass the second largest element will be placed in the (n-1)th position.
- ✓ In (n-1)th pass only the first two elements are compared.

Algorithm for Bubble Sort

BUBBLE_SORT(A, N)

```
Step 1: Repeat Step 2 For I = to N-1
Step 2: Repeat For J = to N - I
Step 3: IF A[J] > A[J + 1]
                SWAP A[J] and A[J+1]
Step 4: EXIT
```

Example for Bubble Sort



To explain about bubble sort technique, let us consider the list of elements stored in the form of an array as follows.

a →

0	1	2	3	4	5
33	44	22	11	66	55

Now we want to sort the above array using the Bubble sort technique in ascending orde

Pass 1 : (first element is compared with all other elements)

We compare a[i] and a[i+1] for i=0,1,2 3 and 4 and exchange a[i] and a[i+1] if a[i] > a[i+1]. Now let us see the proces

0	1	2	3	4	5
33	44	22	11	66	55

33 > 44 → no exchange

44 > 22 → exchange

0	1	2	3	4	5
33	22	44	11	66	55

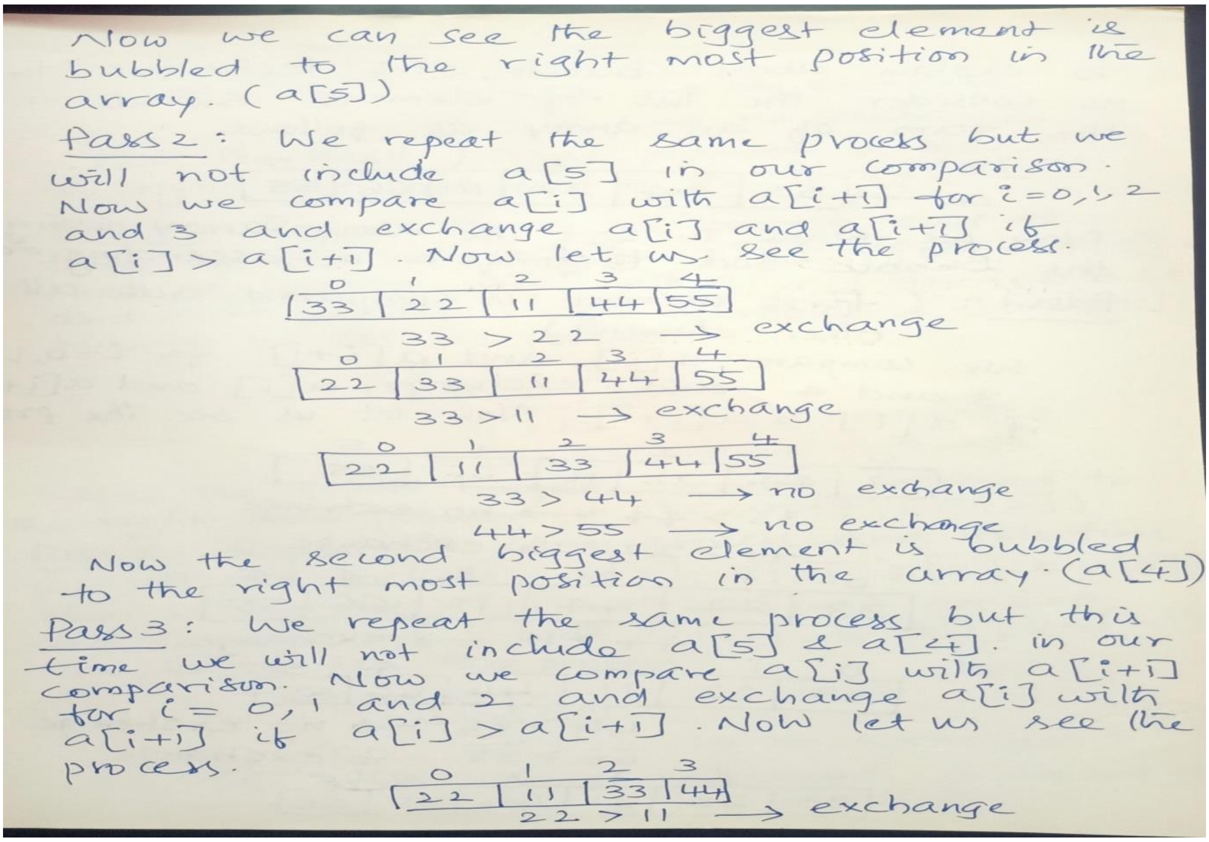
44 > 11 → exchange

0	1	2	3	4	5
33	22	11	44	66	55

44 > 66 → no exchange

66 > 55 → exchange

0	1	2	3	4	5
33	22	11	44	55	66



Now we can see the biggest element is bubbled to the right most position in the array. (a[5])

Pass 2 : We repeat the same process but we will not include a[5] in our comparison. Now we compare a[i] with a[i+1] for i=0,1,2 and 3 and exchange a[i] and a[i+1] if a[i] > a[i+1]. Now let us see the process.

0	1	2	3	4
33	22	11	44	55

33 > 22 → exchange

0	1	2	3	4
22	33	11	44	55

33 > 11 → exchange

0	1	2	3	4
22	11	33	44	55

33 > 44 → no exchange

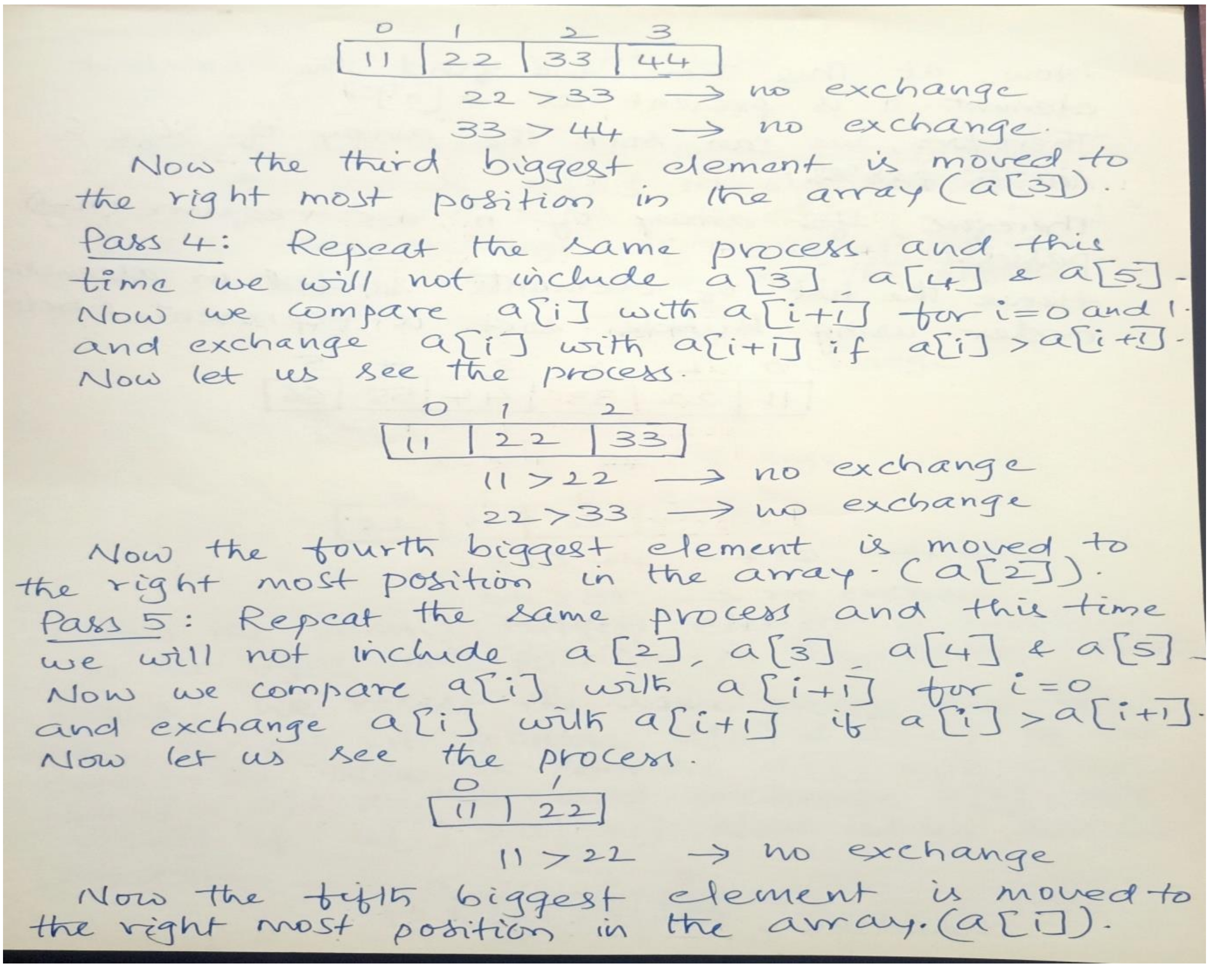
44 > 55 → no exchange

Now the second biggest element is bubbled to the right most position in the array. (a[4])

Pass 3 : We repeat the same process but this time we will not include a[5] & a[4]. in our comparison. Now we compare a[i] with a[i+1] for i = 0, 1 and 2 and exchange a[i] with a[i+1] if a[i] > a[i+1]. Now let us see the process.

0	1	2	3
22	11	33	44

22 > 11 → exchange



0	1	2	3
11	22	33	44

22 > 33 → no exchange

33 > 44 → no exchange.

Now the third biggest element is moved to the right most position in the array (a[3]).

Pass 4: Repeat the same process and this time we will not include a[3], a[4] & a[5]. Now we compare a[i] with a[i+1] for i=0 and 1. and exchange a[i] with a[i+1] if a[i] > a[i+1]. Now let us see the process.

0	1	2
11	22	33

11 > 22 → no exchange

22 > 33 → no exchange

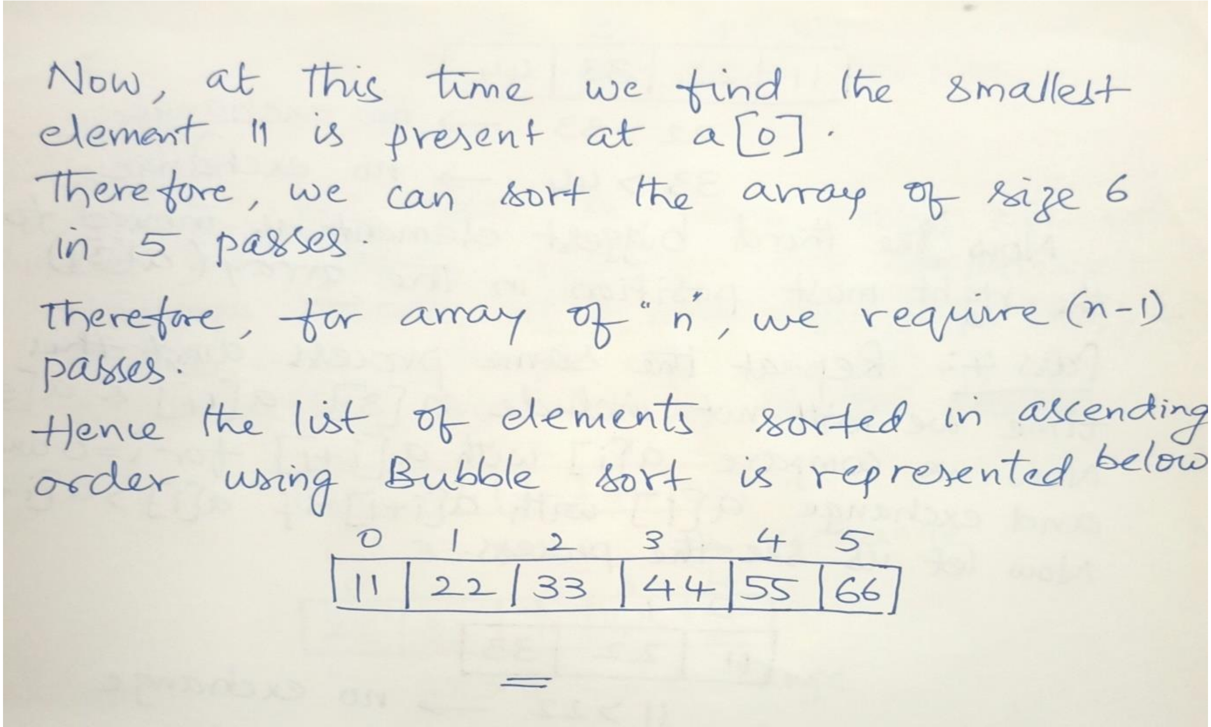
Now the fourth biggest element is moved to the right most position in the array. (a[2]).

Pass 5: Repeat the same process and this time we will not include a[2], a[3], a[4] & a[5]. Now we compare a[i] with a[i+1] for i=0 and exchange a[i] with a[i+1] if a[i] > a[i+1]. Now let us see the process.

0	1
11	22

11 > 22 → no exchange

Now the fifth biggest element is moved to the right most position in the array. (a[1]).



Now, at this time we find the smallest element 11 is present at a[0].

Therefore, we can sort the array of size 6 in 5 passes.

Therefore, for array of "n", we require (n-1) passes.

Hence the list of elements sorted in ascending order using Bubble sort is represented below

0	1	2	3	4	5
11	22	33	44	55	66

Program for Bubble Sort

```
#include<stdio.h>
void bubble_sort(int [], int);
int main(void)
{
        int n, a[10], i;
        clrscr();
        printf(" Enter the size of the array ");
        scanf("%d", &n);
        printf(" Enter the elements of the array ");
        for(i=0; i<n; i++)
                scanf("%d", &a[i]);
        bubble_sort(a,n);
        return 0;
}
void bubble_sort(int a[], int n)
{
        int i, j, m, temp;
        for(i=1; i<n-1; i++)
        {
                for(j=0; j<n; j++)
                {
                        if(a[j] > a[j+1])
                        {
                                temp = a[j];
                                a[j] = a[j+1];
                                a[j+1] = temp;
                        }
                }
        }
        printf(" The sorted list of elements are ");
        for(i=0; i<n; i++)
        printf("%d\t", a[i]);
}
```

QUICK SORT

- ✓ It is developed by **C.A.R. Hoare**.
- ✓ It is also known as **partition exchange sort.**
- ✓ This sorting algorithm uses **divide** and **conquer** strategy.
- ✓ In this method, the **division** is carried out **dynamically**.
- ✓ It contains three steps:
- ✓ **Divide –** split the array into two sub arrays so that each element in the right sub array is greater than the middle element and each element in the left sub array is less than the middle element. The splitting is done based on the middle element called **pivot**. All the elements **less** than **pivot** will be in the left sub array and all the elements **greater** than **pivot** will be on right sub array.
- ✓ **Conquer –** recursively sort the two sub arrays.
- ✓ **Combine –** combine all the sorted elements in to a single list.
- ✓ Consider an array **A[i]** where **i** is ranging from **0 to n – 1** then the division of elements is as follows:

  A[0]……A[m – 1], A[m], A[m + 1] …….A[n]
- ✓ The **partition algorithm** is used to arrange the elements such that all the elements are **less** than **pivot** will be on left sub array and **greater** then **pivot** will be on right sub array.
- ✓ The time complexity of quick sort algorithm in worst case is **O(n^2)**, best case and average case is **O(n log n)**.
- ✓ It is **faster** than other sorting techniques whose time complexity is **O(n log n)**

Algorithm for Quick Sort

```
QUICK_SORT (A, LOW, HIGH)
Step 1: IF (LOW < HIGH)
            CALL PARTITION (A, LOW, HIGH, MID)
            CALL QUICKSORT(A, LOW, MID - 1)
            CALL QUICKSORT(A, MID + 1, HIGH)
    Step 2: EXIT
```

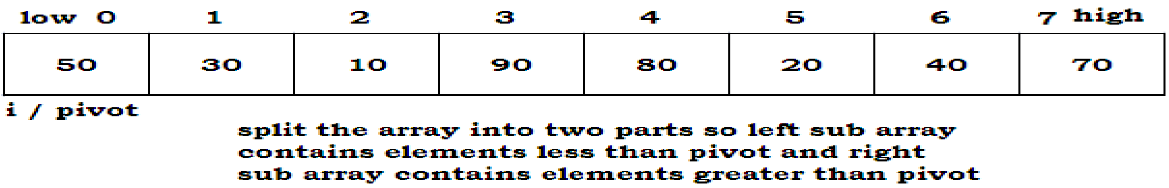
Algorithm for Partition

```
PARTITION (A, LOW, HHIGH, MID)
Step 1: SET PIVOT = A[LOW], I =LOW, J = HIGH
Step 2: Repeat Steps 3 to 5 while I <= LOW
Step 3: Repeat while A[LOW] <= A[PIVOT]
                    SET I = I + 1
Step 4: Repeat while A[j] >= PIVOT
                    SET J = J – 1
Step 5: Repeat if I <= J
            SWAP A[I], A[J]
Step 6: SWAP A[LOW], A[J]
Step 7: Return J
Step 8: EXIT
```

Example for Quick Sort

Let us consider the array of elements to sort them using quick sort technique

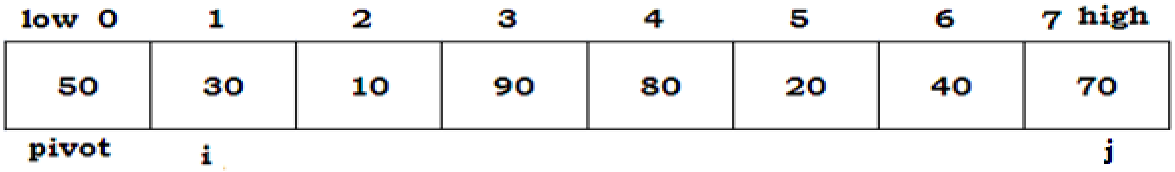
50, 30, 10, 90, 80, 20, 40, 70



low 0	1	2	3	4	5	6	7 high
50	30	10	90	80	20	40	70
i / pivot							

split the array into two parts so left sub array contains elements less than pivot and right sub array contains elements greater than pivot

We will increment i, if(a[i] <= pivot), we will continue incrementing i until the condition is false.



low 0	1	2	3	4	5	6	7 high
50	30	10	90	80	20	40	70
pivot	i						j

low 0	1	2	3	4	5	6	7 high
50	30	10	90	80	20	40	70
pivot		i					j

low 0	1	2	3	4	5	6	7 high
50	30	10	90	80	20	40	70
pivot			i				j

Now a[i] > pivot, so stop incrementing i. As a[j] > pivot we will decrement j until it becomes false

low 0	1	2	3	4	5	6	7 high
50	30	10	90	80	20	40	70
pivot			i			j	

Now we cannot decrement j because a[j] < pivot. Hence we swap a[i] and a[j] since i < j.

low 0	1	2	3	4	5	6	7 high
50	30	10	40	80	20	90	70
pivot			i			j	

Now again a[i] < pivot so increment i

low 0	1	2	3	4	5	6	7 high
50	30	10	40	80	20	90	70
pivot				i		j	

Now a[i] > pivot so stop incrementing i and a[j] > pivot so decrement j

low 0	1	2	3	4	5	6	7 high
50	30	10	40	80	20	90	70
pivot				i	j		

Now a[j] < pivot so stop decrementing j. since i <j swap a[i] and a[j]

low 0	1	2	3	4	5	6	7 high
50	30	10	40	20	80	90	70
pivot				i	j		

Now again a[i] < pivot so increment i

low 0	1	2	3	4	5	6	7 high
50	30	10	40	20	80	90	70
pivot					i j		

Now a[i] > pivot, so stop incrementing i. As a[j] > pivot we will decrement j until it becomes false

.

low 0	1	2	3	4	5	6	7 high
50	30	10	40	20	80	90	70
pivot				j	i		

As a[i] > pivot and a[j] < pivot and j crossed i we will swap a[low] and a[j]

low 0	1	2	3	4	5	6	7 high
20	30	10	40	50	80	90	70
pivot				j	i		

low 0	1	2	3	4	5	6	7 high
20	30	10	40	50	80	90	70
		left sub array		j pivot	i	right sub array	

We will now start left array to be sorted and then right sub array. The new pivot for the left sub array is 20

low 0	1	2	3	4	5	6	7 high
20	30	10	40	50	80	90	70
pivot	i		j				

Now since a[i] > pivot stop incrementing i and a[j] > pivot so decrement j

low 0	1	2	3	4	5	6	7 high
20	30	10	40	50	80	90	70
pivot	i	j					

Now j cannot be decremented and i < j so swap a[i] and a[j]

low 0	1	2	3	4	5	6	7 high
20	10	30	40	50	80	90	70
pivot	i	j					

Now again a[i] < pivot so increment i

low 0	1	2	3	4	5	6	7 high
20	10	30	40	50	80	90	70
pivot		i j					

Now a[i] > pivot so stop incrementing i and a[j] > pivot so decrement j

low 0	1	2	3	4	5	6	7 high
20	10	30	40	50	80	90	70
pivot	j	i					

Since a[j] < pivot so j cannot be decremented and j crossed i so swap a[low] and a[j]

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	80	90	70
pivot	j	i					

There is one element in the left sub array hence all the elements in the right sub array is to be sorted.

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	80	90	70
left sub array	pivot	right sub array					

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	80	90	70
					pivot	i	j

Since a[i] > pivot and a[j] < pivot we stop incrementing I and decrementing j and I < j we swap a[i] and a[j]

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	80	70	90
					pivot	i	j

Since a[i] < pivot so increment i

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	80	70	90
					pivot		i j

Since a[i] > pivot so stop incrementing i and a[j] > pivot so decrement j

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	80	70	90
					pivot	j	i

Since a[j] < pivot so j cannot be decremented and j crossed i so swap a[low] and a[j]

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	70	80	90
					pivot	j	i

low 0	1	2	3	4	5	6	7 high
10	20	30	40	50	70	80	90
						pivot	

Now the left contains 70 and right contains 90 we cannot further subdivide the array. Hence if we look at the array all the elements are sorted.

0	1	2	3	4	5	6	7
10	20	30	40	50	70	80	90

Program for Quick Sort

```
#include <stdio.h>
#define size 100
int partition(int a[], int low, int high);
void quick_sort(int a[], int low, int high);
int main(void)
{
        int a[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &a[i]);
        }
        quick_sort(a, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", a[i]);
        return 0;
}
int partition(int a[], int low, int high)
{
        int left, right, temp, mid, flag;
        mid = left = low;
        right = high;
        flag = 0;
        while(flag != 1)
        {
                while((a[mid] <= a[right]) && (mid!=right))
                        right--;
                if(mid==right)
```

```
                        flag =1;
                else if(a[mid]>a[right])
                {
                        temp = a[mid];
                        a[mid] = a[right];
                        a[right] = temp;
                        mid = right;
                }
                if(flag!=1)
                {
                        while((a[mid] >= a[left]) && (mid!=left))
                                left++;
                        if(mid==left)
                                flag =1;
                        else if(a[mid] <a[left])
                        {
                                temp = a[mid];
                                a[mid] = a[left];
                                a[left] = temp;
                                mid = left;
                        }
                }
        }
        return mid;
}
void quick_sort(int a[], int low, int high)
{
        int mid;
        if(low<high)
        {
                mid = partition(a, low, high);
                quick_sort(a, low, mid -1);
                quick_sort(a, mid+1, high);
        }
}
```

RADIX SORT

- ✓ It is a linear sorting algorithm.
- ✓ It is also known as bucket sort technique or binsort technique or card sort technique since it uses buckets for sorting.
- ✓ It can be applied for integers as well as letters. For integers it used 10 buckets and for letters it uses 26 buckets.
- ✓ If the input is integers then we sort them from least significant digit to most significant digit.
- ✓ The number passes used in radix sort depends on the number of digits.
- ✓ The time complexity of radix sort in all cases is O(n log n)
- ✓ It takes more space compared to other sorting algorithms.
- ✓ It is used only for digits and letters
- ✓ It depends on the number of digits and letters.

Algorithm for Radix Sort

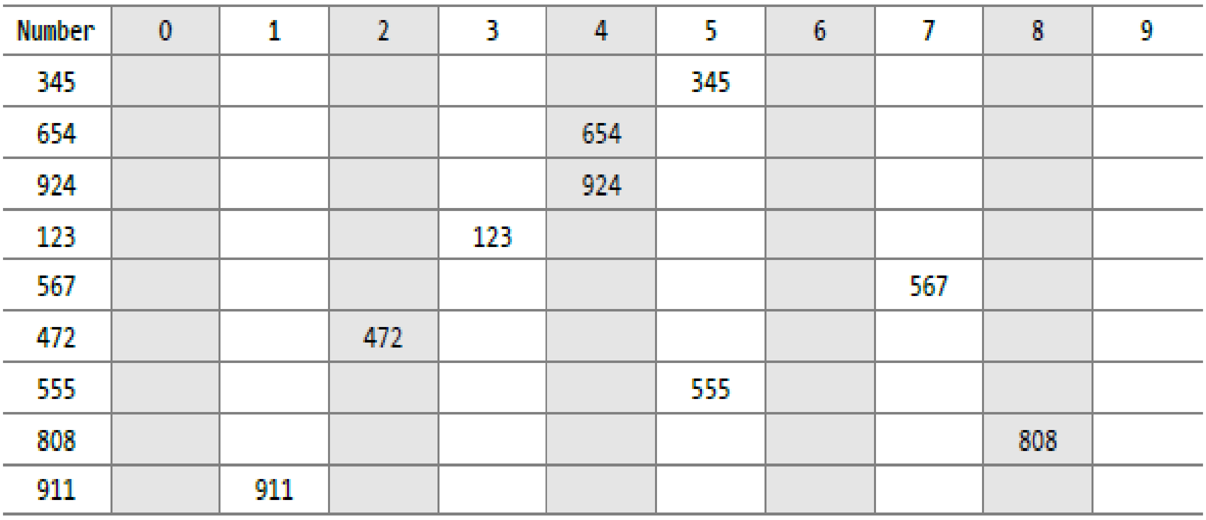
```
RadixSort (A, N)
Step 1: Find the largest number in A as LARGE
Step 2: SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5: SET I = 0 and INITIALIZE buckets
Step 6: Repeat Steps 7 to 9 while I<N-1
Step 7: SET DIGIT = digit at PASSth place in A[I]
Step 8: Add A[I] to the bucket numbered DIGIT
Step 9: INCEREMENT bucket count for bucket numbered DIGIT
Step 10: Collect the numbers in the bucket
Step 11: EXIT
```

Example for Radix Sort

Let us consider the array of elements to sort them using radix sort technique

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at one's place

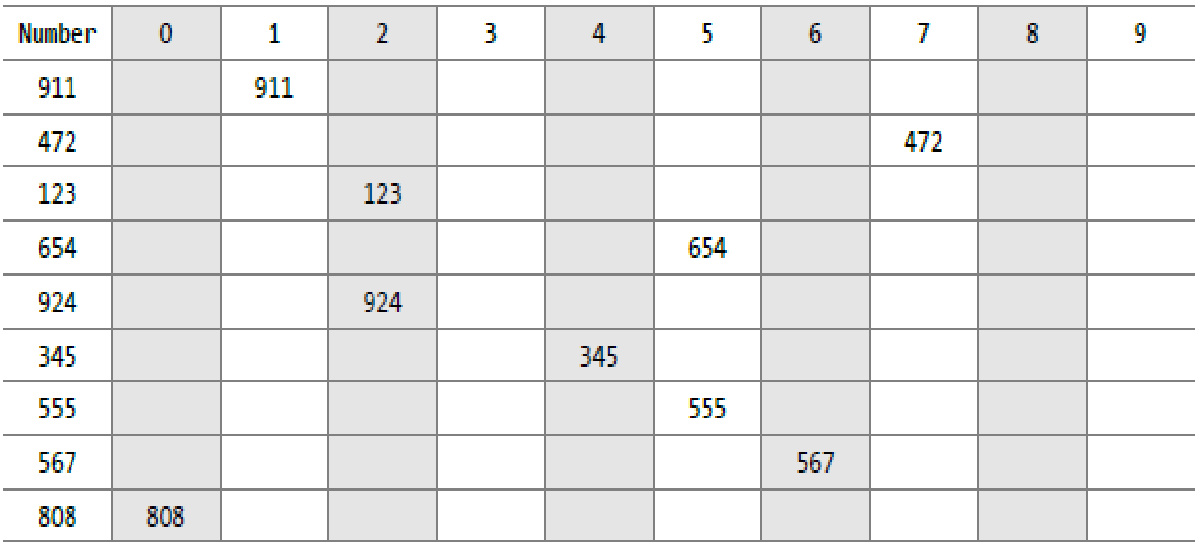


Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After the first pass the numbers are collected bucket by bucket. Thus the new list for the second pass is

911, 472, 123, 654, 924, 345, 555, 567, 808

In the second pass the numbers are sorted according to the digit at ten's place.

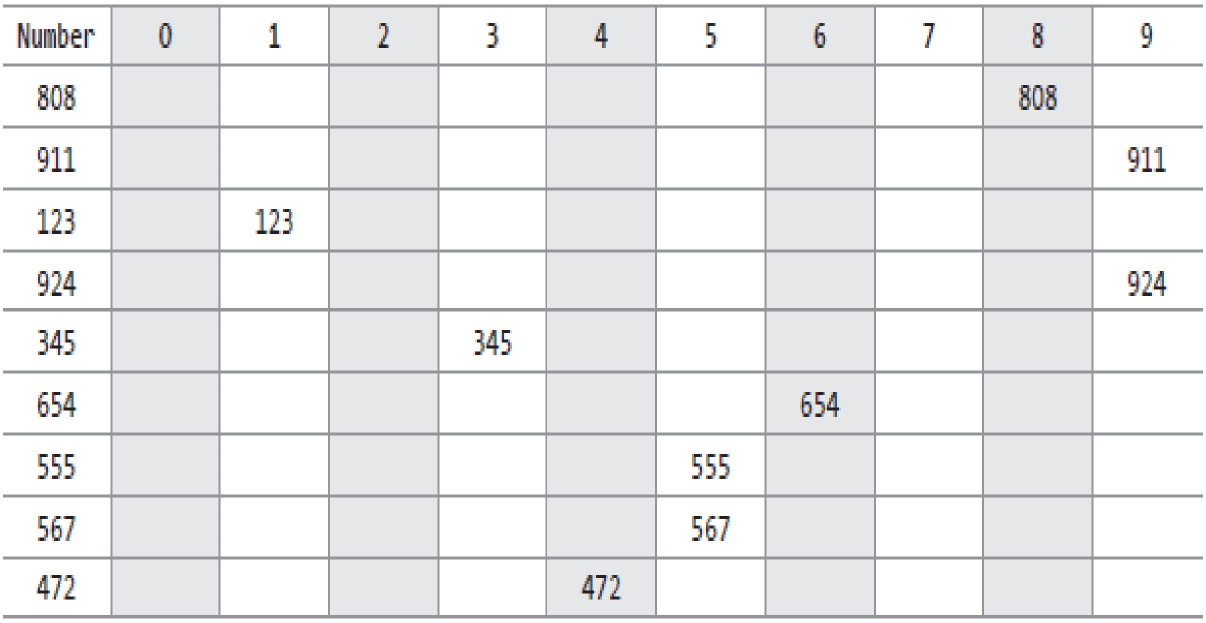


Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

After the second pass the numbers are collected bucket by bucket. Thus the new list for the third pass is

808, 911, 123, 924, 345, 654, 555,567, 472

In the third pass the numbers are sorted according to the digit at hundred place.



Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

After the third pass the numbers are collected bucket by bucket. Thus the new list formed is the final result. It is

123, 345, 472, 555, 567, 654, 808, 911, 924

Program for Radix Sort

```
#include<stdio.h>
int main(void)
{
        int a[100][100], i, n, r=0, c=0, b[100], temp;
        printf(” Enter the size of the array ”);
        scanf(“%d”, &n);
        for(r=0;r<100;r++)
        {
                for(c=0;c<100;c++)
                        a[r][c] = 1000;
        }
        printf(" Enter the array elements ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &b[i]);
                r = b[i] / 100;
                c = b[i] % 100;
                a[r][c] = b[i];
        }
        for(r=0;r<100;r++)
        {
                for(c=0;c<100;c++)
                {
                        for(i=0;i<n;i++)
                        {
                                if(a[r][c] =  =b[i])
                                {
                                        printf("\n\t");
                                        printf("%d", a[r][c]);
                                }
                        }
                }
        }
        return 0;
}
```

MERGE SORT

- ✓ This sorting algorithm uses **divide** and **conquer** strategy.
- ✓ In this method, the division is carried out **dynamically**.
- ✓ It contains three steps:
- ✓ **Divide –** split the array into two sub arrays **s1** and **s2** with each **n/2** elements. If **A** is an array containing **zero** or **one** element, then it is already sorted. But if there are more elements in the array, divide **A**

into two sub-arrays, **s1** and **s2**, each containing half of the elements of **A**.

- ✓ **Conquer –** sort the two sub arrays **s1** and **s2**.
- ✓ **Combine –** combine or merge **s1** and **s2** elements into a unique sorted list.
- ✓ The time complexity of merge sort is **O(n log n)** in all cases.

Algorithm for Merge Sort

```
MERGE_SORT(A,LOW, HIGH)
Step 1: IF LOW < HIGH
            SET MID = (LOW +HIGH)/2
            CALL MERGE_SORT (A, LOW, MID)
            CALL MERGE_SORT (A, MID + 1, HIGH)
            COMBINE (A, LOW, MID, HIGH)
Step 2: EXIT
```

Algorithm for Combine

```
COMBINE (A, LOW, MID, HIGH)
Step 1: SET I = LOW, J = MID + 1, INDEX = LOW
Step 2: Repeat while (I <= MID) AND (J<=HIGH)
                IF A[I] < A[J]
                    SET TEMP[INDEX] = A[I]
                    SET I = I + 1
                    SET INDEX = INDEX + 1
                ELSE
                    SET TEMP[INDEX] = A[J]
                    SET J = J + 1
                    SET INDEX = INDEX + 1

Step 3: [Copy the remaining elements of right sub-array, if any]
    IF I > MID
    Repeat while J <= HIGH
```

```
                                SET TEMP[INDEX] = A[J]
                                SET J = J + 1
                                SET INDEX = INDEX + 1
 [Copy the remaining elements of left sub-array, if any]
        ELSE
                IF A[I]<= MID
                        SET TEMP[INDEX] = A[I]
                        SET I = I + 1
                        SET INDEX = INDEX + 1
Step 4: EXIT
```

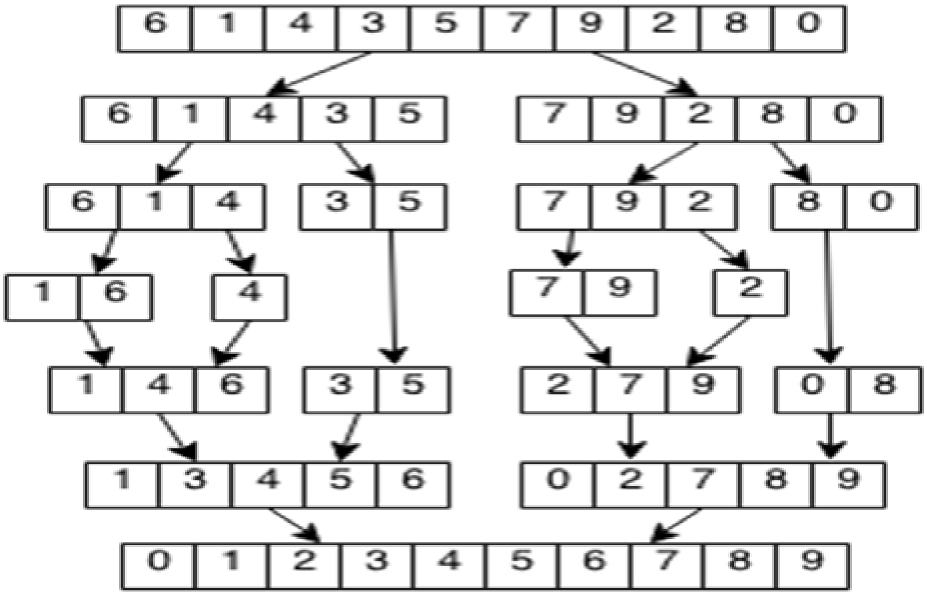
Example for Merge Sort

Let us consider the array of elements to sort them using Merge sort technique

6, 1, 4, 3, 5, 7, 9, 2, 8, 0

0	1	2	3	4	5	6	7	8	9
6	1	4	3	5	7	9	2	8	0
low					mid				high

We then first make the two sublists and combine the two sorted sublists as a unique sorted list.



Now let us see the combine operations

1, 3, 4, 5, 6, 0, 2, 7, 8, 9

0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	0	2	7	8	9
i					j				

Initially k = 0. Then k will be increment

0	1
0	
	k

```
if(a[i] <= a[j])
{
        temp[k] = a[i];
        k++;
        i++;
}
else
{
        temp[k] = a[j];
        k++;
        j++;
}
```

Now i remains there and j is incremented.

0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	0	2	7	8	9
i						j			

Initially k = 1 Then k will be increment

0	1	2
0	1	
		k

```
if(a[i] <= a[j])
{
        temp[k] = a[i];
        k++;
        i++;
}
else
{
        temp[k] = a[j];
        k++;
        j++;
}
```

Now j remains there and i is incremented.

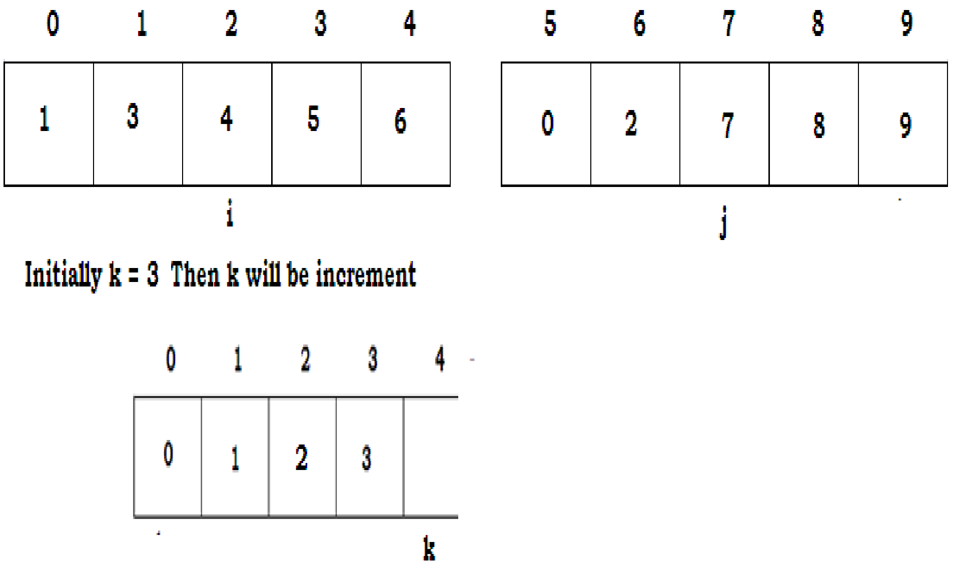
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	0	2	7	8	9
	i					j			

Initially k = 2 Then k will be increment

0	1	2	3
0	1	2	
			k

```
if(a[i] <= a[j])
{
        temp[k] = a[i];
        k++;
        i++;
}
else
{
        temp[k] = a[j];
        k++;
        j++;
}
```

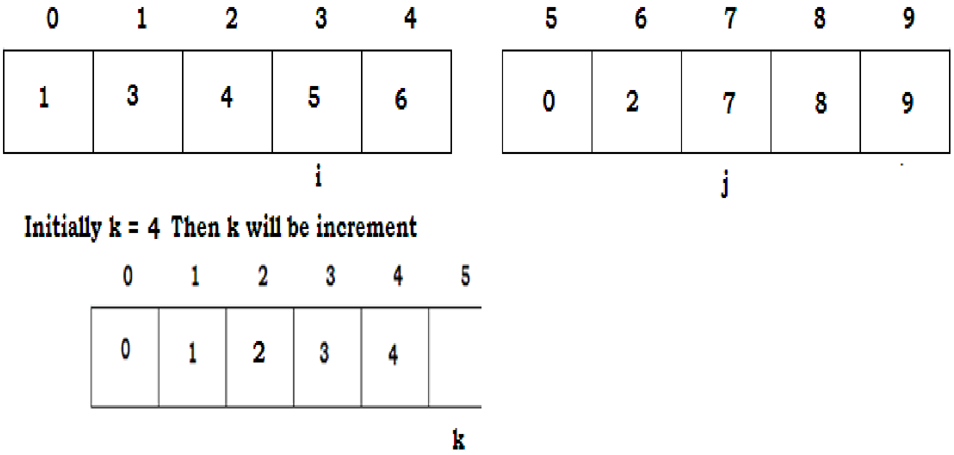
Now i remains there and j is incremented.



Initially k = 3 Then k will be increment

```
if(a[i] <= a[j])
{
        temp[k] = a[i];
        k++;
        i++;
}
else
{
        temp[k] = a[j];
        k++;
        j++;
}
```

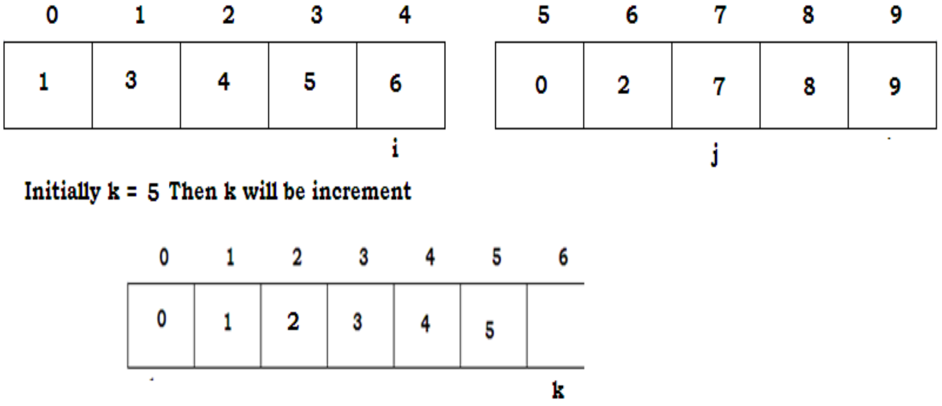
Now j remains there i is incremented



Initially k = 4 Then k will be increment

```
if(a[i] <= a[j])
{
        temp[k] = a[i];
        k++;
        i++;
}
else
{
        temp[k] = a[j];
        k++;
        j++;
}
```

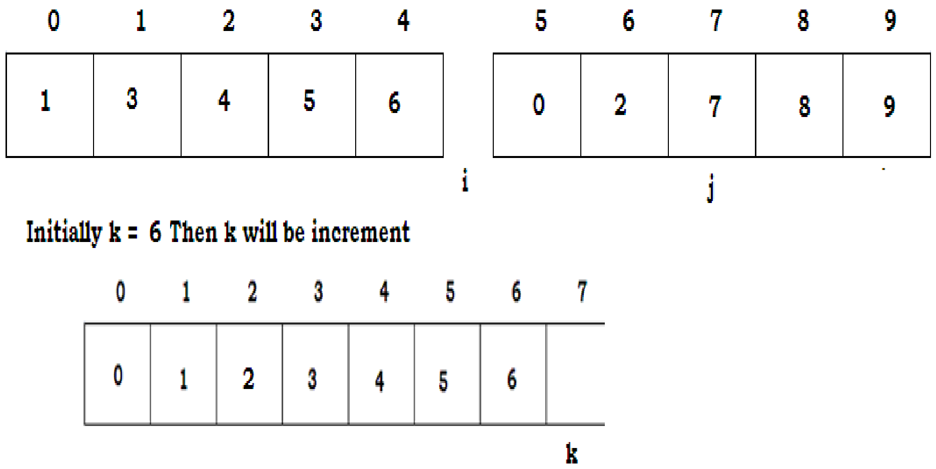
Now again i is incremented



Initially k = 5 Then k will be increment

```
if(a[i] <= a[j])
{
        temp[k] = a[i];
        k++;
        i++;
}
else
{
        temp[k] = a[j];
        k++;
        j++;
}
```

Now again i is incremented.



0	1	2	3	4
1	3	4	5	6

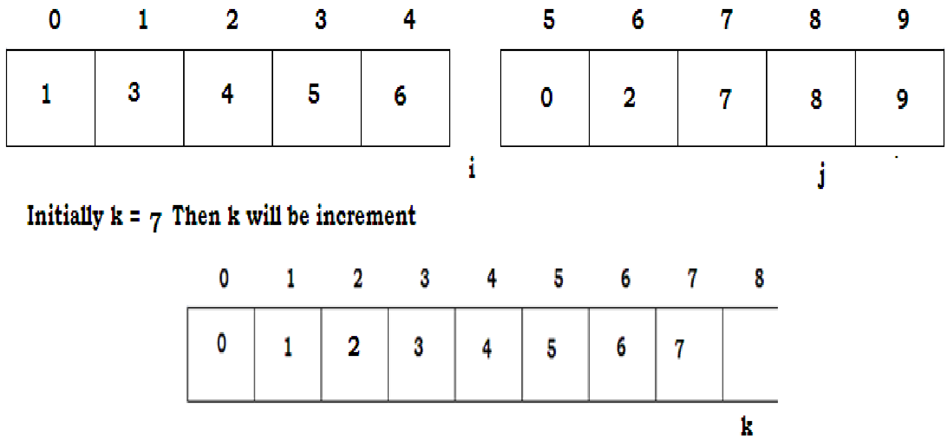
5	6	7	8	9
0	2	7	8	9

Initially k = 6 Then k will be increment

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	

```
if(a[i] <= a[j])
{
        temp[k] = a[i];
        k++;
        i++;
}
else
{
        temp[k] = a[j];
        k++;
        j++;
}
```

Now again i is incremented. But the left sub list is completed then j is incremented until the right sub list is completed



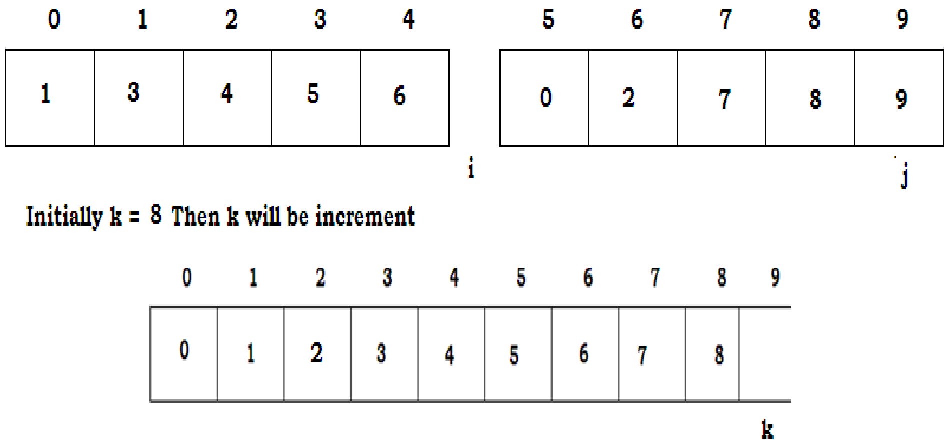
0	1	2	3	4
1	3	4	5	6

5	6	7	8	9
0	2	7	8	9

Initially k = 7 Then k will be increment

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	

```
if(j <= high)
{
        temp[k] = a[j];
        k++;
        j++;
}
```



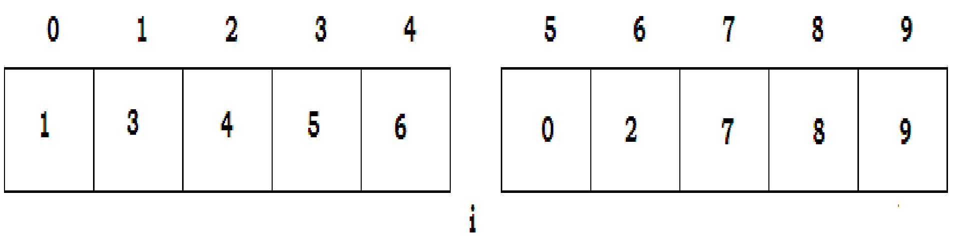
0	1	2	3	4
1	3	4	5	6

5	6	7	8	9
0	2	7	8	9

Initially k = 8 Then k will be increment

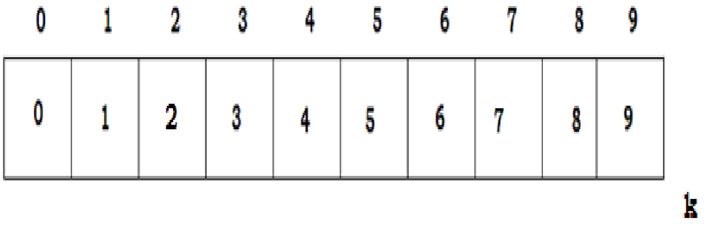
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	

```
if(j <= high)
{
        temp[k] = a[j];
        k++;
        j++;
}
```



```
if(j <= high)
{
    temp[k] = a[j];
    k++;
    j++;
}
```

Initially k = 9 Then k will be increment



Finally we see the array is in sorted order.

Program for Merge Sort

```
#include <stdio.h>
#define size 100
void combine(int a[], int, int, int);
void merge_sort(int a[],int, int);
int main(void)
{
        int a[size], i, n;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
                scanf("%d", &a[i]);
        merge_sort(a, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", a[i]);
        return 0;
}
```

```
void combine(int a[], int low, int mid, int high)
{
        int i=low, j=mid+1, index=low, temp[size], k;
        while((i<=mid) && (j<=high))
        {
                if(a[i] < a[j])
                {
                        temp[index] = a[i];
                        i++;
                }
                else
                {
                        temp[index] = a[j];
                        j++;
                }
                index++;
        }
        if(i>mid)
        {
                while(j<=high)
                {
                        temp[index] = a[j];
                        j++;
                        index++;
                }
        }
        else
        {
                while(i<=mid)
                {
                        temp[index] = a[i];
                        i++;
                        index++;
                }
        }
        for(k=low;k<index;k++)
                a[k] = temp[k];
}
void merge_sort(int a[], int low, int high)
{
        int mid;
        if(low<high)
        {
                mid = (low+high)/2;
                merge_sort(a, low, mid);
                merge_sort(a, mid+1, high);
                combine(a, low, mid, high);
        }
}
```

HEAP SORT

- ✓ Heap is a complete binary tree and also a Max(Min) tree.
- ✓ A Max(Min) tree is a tree whose root value is larger(smaller) than its children.
- ✓ This sorting technique has been developed by J.W.J. Williams.
- ✓ It is working under two stages.

  Heap construction

  Deletion of a Maximum element key
- ✓ The heap is first constructed with the given numbers. The maximum key value is deleted for n -1 times to the remaining heap. Hence we will get the elements in decreasing order. The elements are deleted one by one and stored in the array from last to first. Finally we get the elements in ascending order.
- ✓ The important points about heap sort technique are:

- ✓ The time complexity of heap sort is O(n log n)
- ✓ This is a in-place sorting algorithm.
- ✓ For random input it works slower than quick sort
- ✓ Heap sort is not a stable sorting method
- ✓ The space complexity of heap sort is O(1).

Algorithm Heap Sort

- ✓ Build a max heap from the input data.
- ✓ At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
- ✓ Repeat above steps while size of heap is greater than 1

Procedure for Working of Heap Sort

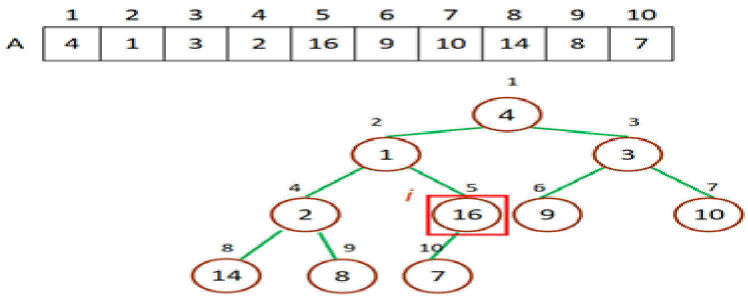
Initially on receiving an unsorted list,

- ✓ First step in heap sort is to build Max-Heap.
- ✓ Repeat Second, Third and Fourth steps, until we have the complete sorted list in our array.
- ✓ Second step - Once heap is built, the first element of the Heap is largest, so we exchange first and last element of a heap.
- ✓ Third step - We delete and put last element(largest) of the heap in our array.
- ✓ Fourth step - Then we again make heap using the remaining elements, to again get the largest element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

Example for Heap Sort

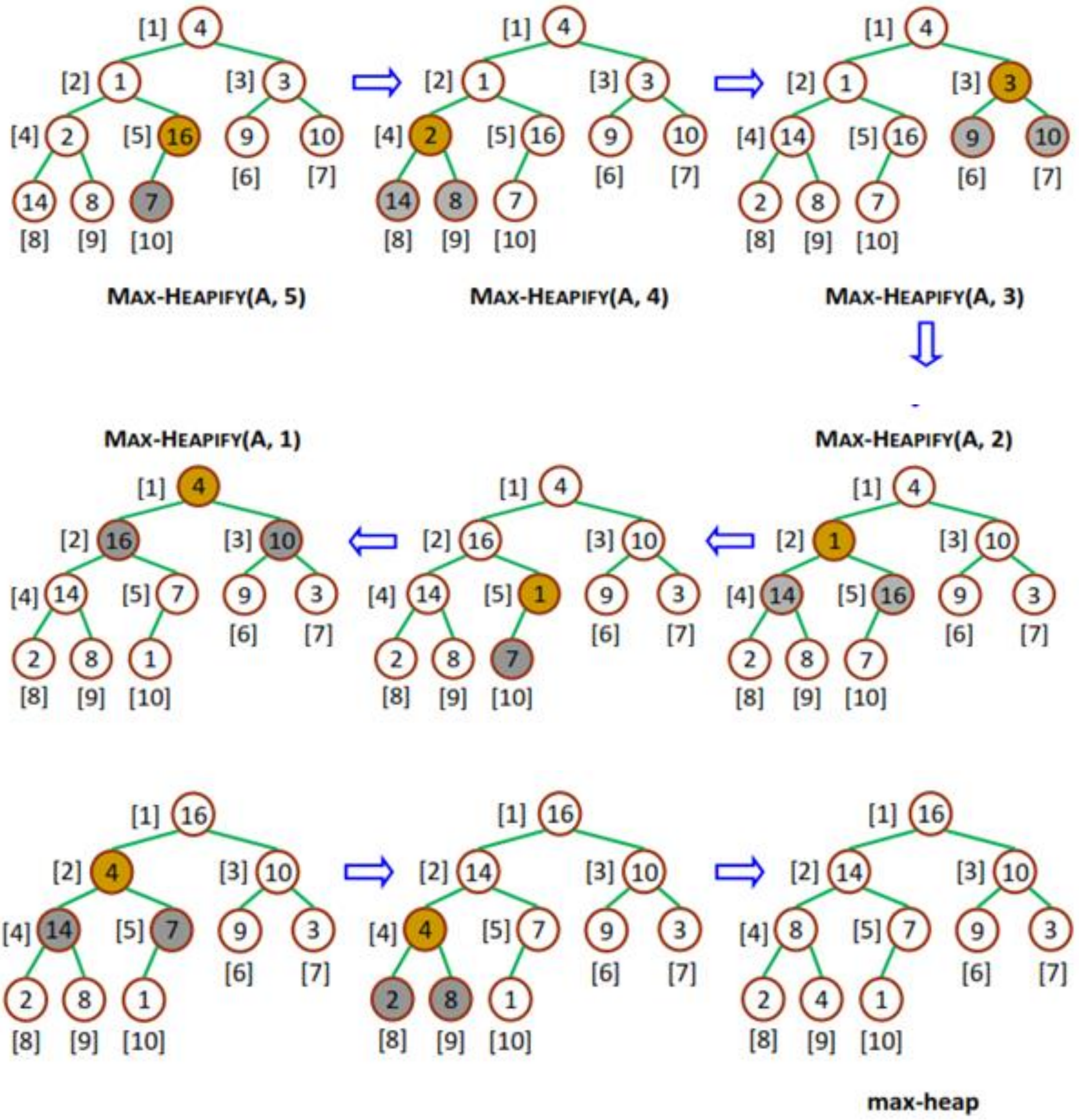
Let us consider the array of elements to sort them using heap sort technique

4, 1, 3, 2, 16, 9, 10, 14, 8, 7

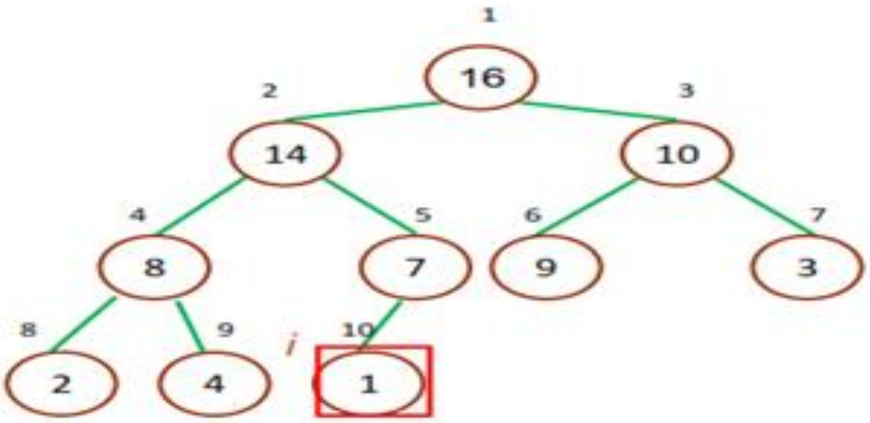


	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

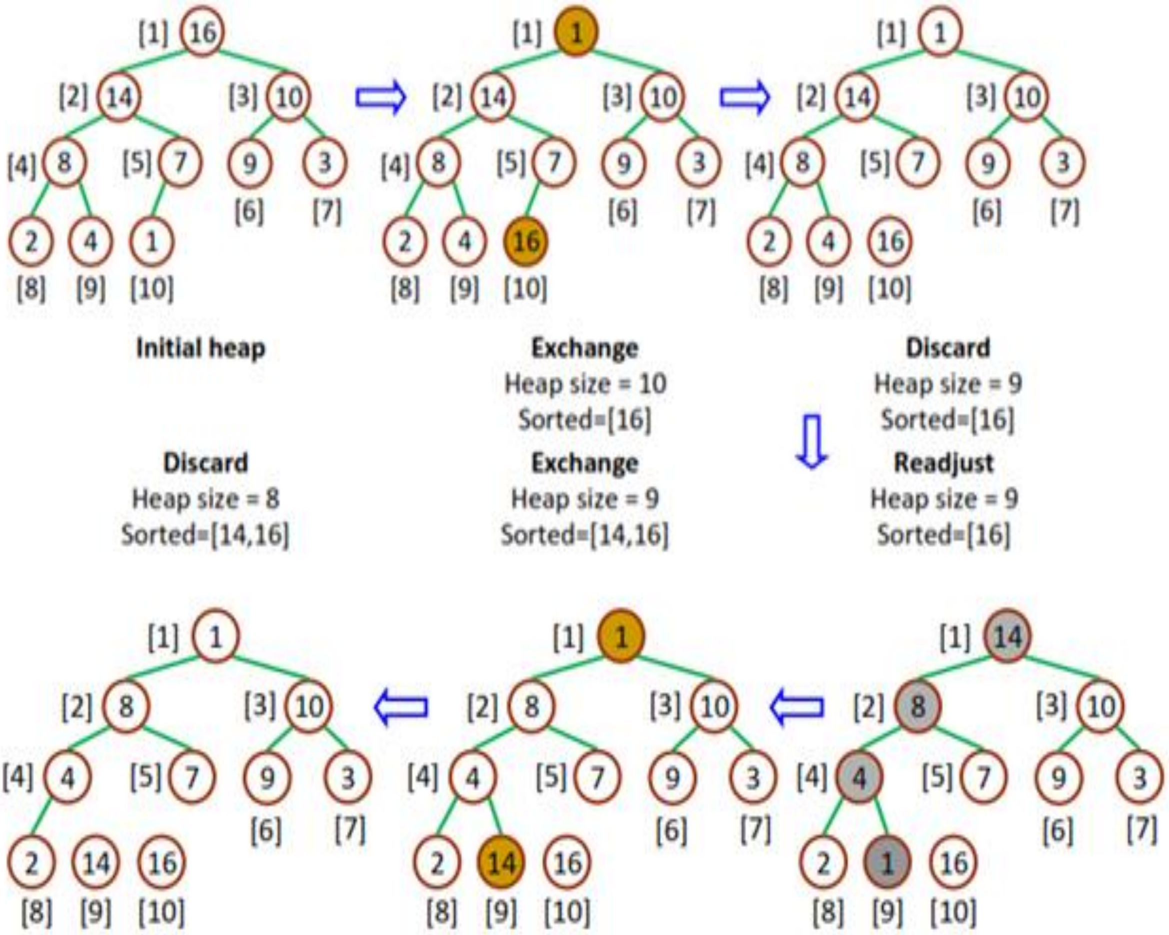
Stage -1 construction of heap



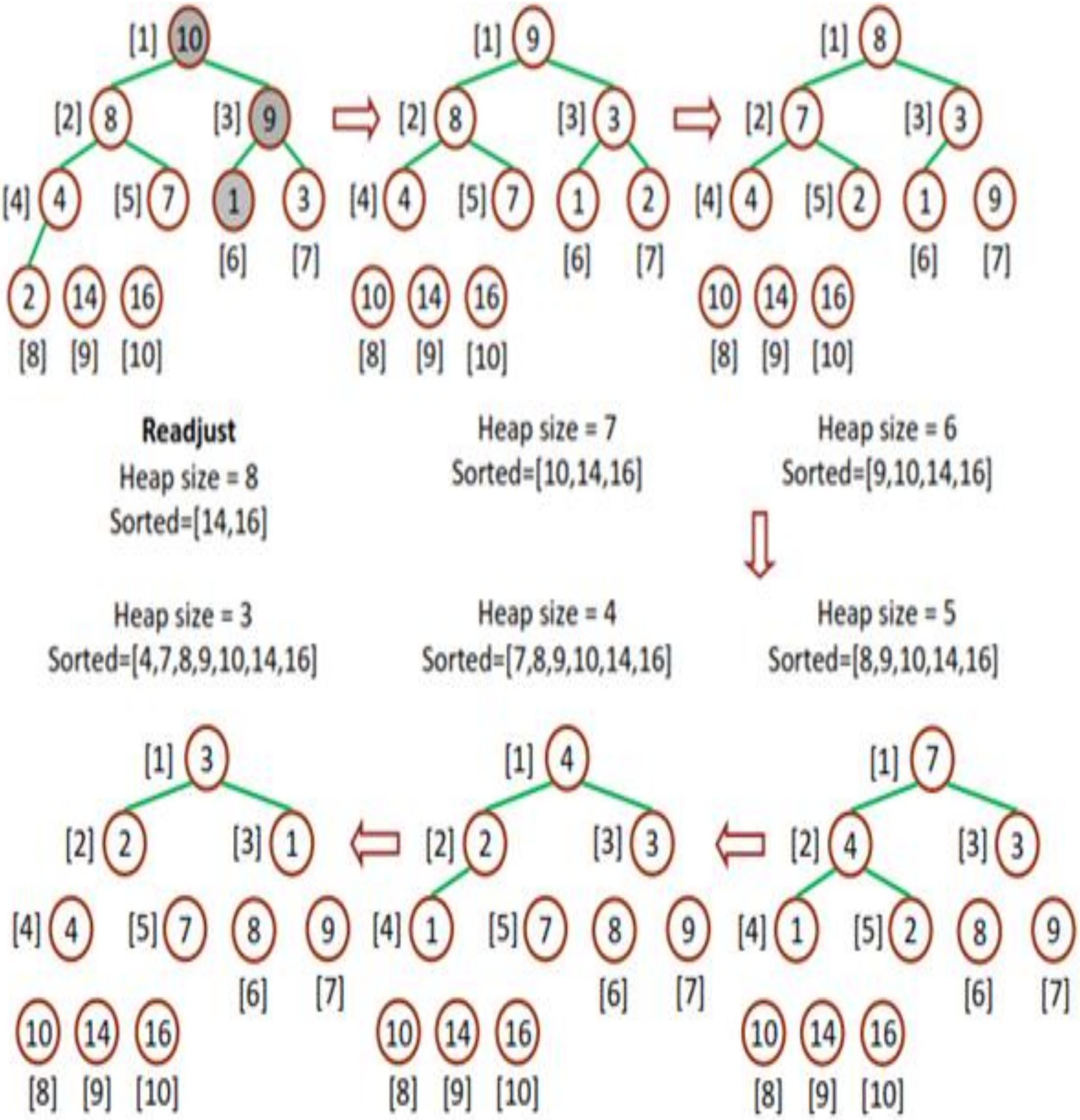
Stage – 2 deletion of maximum key element



	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16



Dr. Ratna Raju Mukiri M.Tech(CSE)., S.E.T., Ph.D.,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Program for Heap Sort

```
#include<stdio.h>
void heap_sort(int[], int);
void makeheap(int[], int);
int main(void)
{
        int a[10], n, i;
        printf(" Enter the size of the array ");
        scanf("%d", &n);
        printf(" Enter the array elements ");
        for(i=0;i<n;i++)
                scanf("%d", &a[i]);
        makeheap(a, n);
        heap_sort(a, n);
        printf(" The elements after sorting are ");
        for(i=0;i<n;i++)
                printf("\t%d", a[i]);
        return 0;
}
void makeheap(int a[], int n)
{
        int i, val, j, parent;
        for(i=1;i<n;i++)
        {
                val = a[i];
                j = i;
                parent = (j - 1) / 2;
                while(j>0 && parent < val)
                {
                        a[j] = a[parent];
                        j = parent;
                        parent = (j - 1) / 2;
                }
                a[j] = val;
        }
}

void heap_sort(int a[], int n)
{
        int i, j, k, temp;
        for(i=n-1;i>0;i--)
        {
                temp = a[i];
                a[i] = a[0];
                k = 0;
                if(i == 1)
                        j = -1;
                else
                        j = 1;
                if(i > 2 && a[2] > a[1])
                        j = 2;
                while(j >=0 && temp < a[j])
                {
                        a[k] = a[j];
                        k = j;
                        j = 2 * k +1;
                        if(j+1 <= i-1 && a[j] < a[j+1])
                                j++;
                        if(j > i-1)
                                j = -1;
                }
                a[k] = temp;
        }
}
```

Code No: R1632024 **R16** SET - 1

III B. Tech II Semester Regular Examinations, April/May - 2019

DATA STRUCTURES

(Electrical and Electronics Engineering)

Time: 3 hours Max. Marks: 70

Note: 1. Question Paper consists of two parts (**Part-A** and **Part-B)**
2. Answer **ALL** the question in **Part-A**
3. Answer any **FOUR** Questions from **Part-B**

~~~~~~

PART –A

1. a) Define data structure. [2M]
b) What is a double-ended-queue? [2M]
c) What is a self referential structure? Give an example. [2M]
d) Discuss the role of mid element in binary search. [3M]
e) Construct the graph whose adjacency matrix is given below and also analyze its properties. [3M]

$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 1 & 0 & 0 & 1 & 1 \\ B & 1 & 0 & 1 & 0 & 0 & 1 \\ C & 0 & 1 & 0 & 1 & 1 & 0 \\ D & 0 & 0 & 1 & 0 & 0 & 1 \\ E & 1 & 0 & 1 & 0 & 0 & 0 \\ F & 1 & 1 & 0 & 1 & 0 & 0 \end{array}$$

f) Present the time complexity of merge sort in different cases. [2M]

PART -B

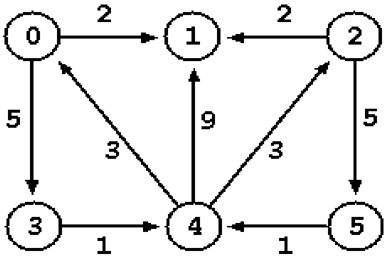
2. a) Explain about different notations for time complexity and space complexity of algorithms. [7M]
b) Write a C program to traverse an array in reverse order. [7M]

3. a) Discuss the role of stacks in executing recursive procedures. [7M]
b) What is a priority queue? Explain different methods of implementing them. [7M]

4. a) Write a C function to implement insert operation in a circular linked list. [7M]
b) Explain with an example, how linked lists can be used for sparse matrix representation and computations. [7M]

5. a) What is a threaded binary tree? Discuss its advantages and limitations. [7M]
b) Insert the sequence of integers 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6 and 18 in an initially empty Binary Search Tree. Then delete 5 and 1. (Present the operations one after the other in the same order). [7M]
~~~~~~

6. a) Compute shortest paths between every pair of vertices in the graph below using appropriate algorithm. [7M]



b) Write notes on basic operations performed on graphs and challenges involved. [7M]

7. a) Write and explain Fibonacci Search algorithm. [7M]

b) Sort the below list of elements in ascending order using heap sort: [7M]
6, 8, 7, 9, 1, 4, 3, 2, 5, 0.

*****

Code No: R1632024 **R16** SET - 2

III B. Tech II Semester Regular Examinations, April/May - 2019

DATA STRUCTURES

(Electrical and Electronics Engineering)

Time: 3 hours Max. Marks: 70

Note: 1. Question Paper consists of two parts (**Part-A** and **Part-B)**
2. Answer **ALL** the question in **Part-A**
3. Answer any **FOUR** Questions from **Part-B**

~~~~~~

PART –A

1. a) Give the syntax for declaring a 3-dimensional array. [2M]
b) Write brief notes on polish notation. [3M]
c) What is a spare matrix? Give an example. [2M]
d) What are balanced binary trees? Why it is needed to balance binary trees? [2M]
e) Define the terms: simple graph, directed graph and connected graph. [3M]
f) What is stable sorting? Give an example. [2M]

PART -B

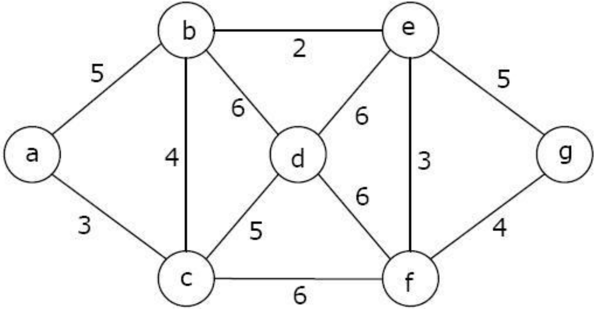
2. a) Write about the classification of data structures. [7M]
b) Explain about different operations in String ADT. [7M]

3. a) Write an algorithm/program that gives solution for Towers of Hanoi problem with n disks. [7M]
b) With array representation, explain the basic queue operations. [7M]

4. a) Write a C program to traverse a given single linked list in reverse order. [7M]
b) Explain with an example, how linked lists can be used for polynomial representation. [7M]

5. a) Discuss about different representations of binary trees. Give an example for each. [7M]
b) Insert the sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree. Then delete 40 and 70. (Present the operations one after the other.) [7M]

6. a) Differentiate between BFS and DFS with algorithms and examples. [7M]
b) Compute minimum cost spanning tree for the graph below using Prim’s algorithm: [7M]



7. a) Write and explain Binary Search algorithm. Also mention its time complexity. [7M]
b) Sort the below list of elements in ascending order using quick sort: [7M]
29, 23, 17, 57, 34, 89, 65, 27.

*****
~~~~~~

Code No: R1632024 **R16** SET - 3

III B. Tech II Semester Regular Examinations, April/May - 2019

DATA STRUCTURES

(Electrical and Electronics Engineering)

Time: 3 hours Max. Marks: 70

Note: 1. Question Paper consists of two parts (**Part-A** and **Part-B)**
2. Answer **ALL** the question in **Part-A**
3. Answer any **FOUR** Questions from **Part-B**

~~~~~~

PART –A

1. a) List different operations that can be performed on an array. [2M]
b) What is a queue data structure? [2M]
c) Write C structure to declare node of a double-linked list. [2M]
d) Define the terms: root, leaf and siblings with respect to trees. [3M]
e) List some real world applications of directed, undirected and hybrid graphs. [3M]
f) Present the time complexity of quick sort in different cases. [2M]

PART -B

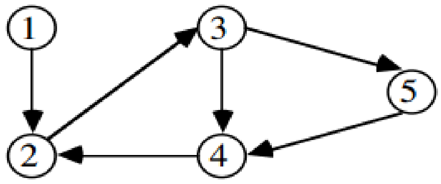
2. a) With a neat sketch, explain the model of ADT. [7M]
b) Explain how linear arrays are stored and traversed. [7M]

3. a) Explain how postfix expressions are evaluated using stacks. Give an example. [7M]
b) Differentiate between regular queues and circular queues with insert and delete operations. [7M]

4. a) Discuss the advantages and limitations of linked lists. [7M]
b) Write a C program to implement queues using linked lists. [7M]

5. a) Write recursive functions for inorder, preorder and postorder traversal in a binary tree. [7M]
b) Construct a max heap from the sequence of integers 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6 and 18. Then delete 2 minimum elements. (Present the operations one after the other in the same order.) [7M]

6. a) What is transitive closure? Compute transitive closure of the graph given below, using Warshall's algorithm: [7M]



b) Write and explain Kruskal's algorithm for finding minimum cost spanning tree of a graph. [7M]

7. a) Give a comparison between several searching techniques. [7M]
b) Write and explain iterative merge sort algorithm, with an example. [7M]

*****
~~~~~~

Code No: R1632024 **R16** SET - 4

III B. Tech II Semester Regular Examinations, April/May - 2019

DATA STRUCTURES

(Electrical and Electronics Engineering)

Time: 3 hours Max. Marks: 70

Note: 1. Question Paper consists of two parts (**Part-A** and **Part-B)**
2. Answer **ALL** the question in **Part-A**
3. Answer any **FOUR** Questions from **Part-B**

~~~~~~

PART –A

1. a) What is the key difference between strings and regular arrays? [2M]
   b) What is a stack data structure? [2M]
   c) Define a header linked list. [2M]
   d) What is a complete binary tree? [3M]
   e) What is a biconnected component? [3M]
   f) What are the limitations of binary search? [2M]

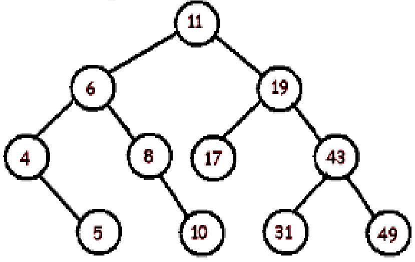
PART -B

2. a) List and explain different operations performed on data structures. [7M]
   b) Write a C program to add two matrices, using multidimensional arrays. [7M]

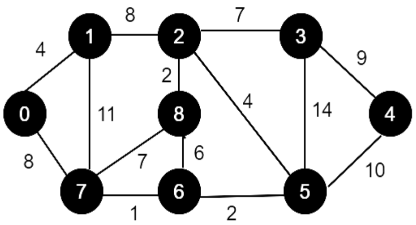
3. a) Explain how an infix expression can be converted into postfix expression, using stacks. Give an example. [7M]
   b) Write and explain the queue ADT. [7M]

4. a) Differentiate between arrays and linked lists. [7M]
   b) Write a C program to implement stacks using linked lists, doubly and circular linked lists. [7M]

5. a) Explain the procedure for deletion of an element from a binary search tree. [7M]
   b) Present the preorder, inorder and postorder traversal of the below binary tree: [7M]



6. a) Explain about various graph representations. Discuss the pros and cons of each. [7M]
   b) Using Dijkstra's algorithm, find shortest paths from vertex 0 to remaining vertices in the graph given below. [7M]



7. a) What is hashing? Explain its role, advantages and disadvantages w.r.to searching. [7M]
   b) Sort the below list of elements in ascending order using shell sort: [7M]
   3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2

*****
~~~~~~