I Year B.Tech. AI & DS – I Semester

L	T	P	C
3	0	0	3

(23A05302T) ADVANCED DATA STRUCTURES & ALGORITHM ANALYSIS

Course Objectives: The main objectives of the course is to

- provide knowledge on advance data structures frequently used in Computer Science domain
- Develop skills in algorithm design techniques popularly used
- Understand the use of various data structures in the algorithm design

Course Outcomes: After completion of the course, students will be able to

- Illustrate the working of the advanced tree data structures and their applications (L2)
- Understand the Graph data structure, traversals and apply them in various contexts. (L2)
- Use various data structures in the design of algorithms (L3)
- Recommend appropriate data structures based on the problem being solved (L5)
- Analyze algorithms with respect to space and time complexities (L4)
- Design new algorithms (L6)

UNIT – I:

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.

AVL Trees – Creation, Insertion, Deletion operations and Applications

B-Trees – Creation, Insertion, Deletion operations and Applications

UNIT – II:

Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications

Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications

Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication, Convex Hull

UNIT – III:

Greedy Method: General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths

Dynamic Programming: General Method, All pairs shortest paths, Single Source Shortest Paths – General Weights (Bellman Ford Algorithm), Optimal Binary Search Trees, 0/1 Knapsack, String Editing, Travelling Salesperson problem

UNIT – IV:

Backtracking: General Method, 8-Queens Problem, Sum of Subsets problem, Graph Coloring, 0/1 Knapsack Problem

Branch and Bound: The General Method, 0/1 Knapsack Problem, Travelling Salesperson problem

UNIT – V:

NP Hard and NP Complete Problems: Basic Concepts, Cook's theorem
NP Hard Graph Problems: Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson Decision Problem (TSP)
NP Hard Scheduling Problems: Scheduling Identical Processors, Job Shop Scheduling

Textbooks:

1. Fundamentals of Data Structures in C++, Horowitz, Ellis; Sahni, Sartaj; Mehta, Dinesh 2nd Edition Universities Press
2. Computer Algorithms/C++ Ellis Horowitz, SartajSahni, SanguthevarRajasekaran2nd Edition University Press

Reference Books:

1. Data Structures and program design in C, Robert Kruse, Pearson Education Asia
2. An introduction to Data Structures with applications, Trembley & Sorenson, McGraw Hill
3. The Art of Computer Programming, Vol.1: Fundamental Algorithms, Donald E Knuth, Addison-Wesley, 1997.
4. Data Structures using C & C++: Langsam, Augenstein&Tanenbaum, Pearson, 1995
5. Algorithms + Data Structures &Programs:,N.Wirth, PHI
6. Fundamentals of Data Structures in C++: Horowitz Sahni& Mehta, Galgottia Pub.
7. Data structures in Java:, Thomas Standish, Pearson Education Asia

Online Learning Resources:

1. https://www.tutorialspoint.com/advanced_data_structures/index.asp
2. http://peterindia.net/Algorithms.html
3. Abdul Bari,1. Introduction to Algorithms (youtube.com)

Unit-1

Introduction of Algorithm

Algorithm:— It is a step by step process which defines a set of instructions to be executed in certain order to get desired output.

In addition all algorithms should satisfy the following criterias

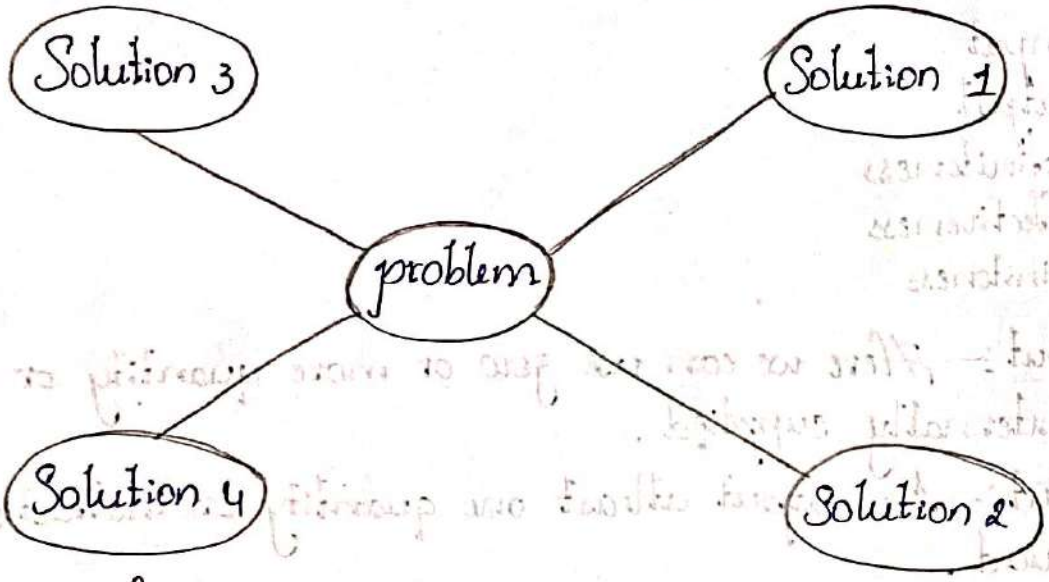
1. Input
2. Output
3. Definiteness
4. Effectiveness
5. Finiteness

1. Input:— Here we can use zero or more quantity or instruction or externally supplied.
2. Output:— In output atleast one quantity or instruction is produced.
3. Definiteness:— Here the each instruction should be clear.
4. Effectiveness:— The every instruction must be very basic. Here Basic is nothing but starting point, so that it can be carry.
5. Finiteness:— An algorithm must always terminate after a finite number of steps.

Studying (or) Issues of Algorithm:—

1. How to design an algorithm?
   We can use design for creating an algorithm.
2. How to Analyze an algorithm?
   We can analyze an algorithm by using time and space complexity.
3. How to express an algorithm?
   We can express an algorithm by using definiteness.

4. How to validate an algorithm?
We can validate an algorithm by using finiteness.
5. How to test an algorithm?
We can test the algorithm to check it for errors.
We design a algorithm to get a solution for a given problem, a problem can be solved in more than one way.



Algorithm specification:—
In this algorithm specification we can use in three ways.
1) Natural language like English.
2) Graphic Representation like flowchart.
3) Pseudocode Algorithm.

Space and Time Complexity:— Space complexity is an algorithms is the amount of memory it needs to run to complete. Here we are not using any memory when we write an algorithm, but we can measure (or) estimate how much memory is required for a particular problem, if it is implemented as program.

We can see what are the variables that we have taken based on that we can assign the number of variables are needed, so that here we are concerting how much amount of memory is needed to complete to execute a program. We call it as space complexity, In this space complexity we can be measure by two constrains.

$$S(p) = c(\text{fixed}) + sp(\text{variable})$$

1. fixed part:— It is independence of instance characteristics it means for a particular program declare a variables for that variable the space is needed to store the variable is fixed.

2. Variable part:—
   * It is dependent.
   * Space for variables whose size is dependent for particular problem.
   * If we take fixed part it is simple variables while we declaring like int a, int b. These type of variables are called fixed part.
   * Whereas variable part if we declare an array, it can give some size, but how much memory required an array to store. We can't say exactly but we can exactly say when we know the size of an array that is called as variable part.

Time Complexity:— The time complexity of an algorithm is the amount of computer time it needs to run to complete.

Rules for Algorithm:—

for algorithm heading → 0
for Braces → 0
for expressions → 1
for looping statements → It depend on Naof times } n
loop repeating

Best case complexity:— In this algorithm is taking minimum amount of time for its execution is called Best case (less time).

Worst case complexity:— In this algorithm is taking maximum amount of time for its execution is called Worst case (More time).

Average Case Complexity :— In this algorithm is taking Average amount of time for its execution is called Average case (Not less Not More).

Example :—

Black colour first → Best case

Black colour Middle → Avg case

Black colour last → Worst case

* Time Complexity :

Stack	Insert	Delete
	O(1)	O(1)

* Space complexity of stack = O(n)

* Time Complexity :

Queue	Insert	Delete
	O(1)	O(1)

* Space complexity of Queue = O(n)

* Time complexity :

List	Insert	Delete
	O(1)	O(1)

* Space complexity of list = O(n)

Algorithm :— (Time Complexity)

```
Algorithm Sum(a,n)
{
S=0.0;
count = count+1;
for i=1 to n do 8
{
count=count+1;
S=S+a[i];
count=count+1;
}
count=count+1;
count=count+1;
return s;
}
```

If the count is zero to start with, then it will be $2n+3$ on termination. So each invocation of sum execute a total of $2n+3$ steps.

Statement	S/e	Frequency	Total
1. Algorithm Sum(a,n)	0	-	0
2. {	0	-	0
3. S=0.0;	1	1	1
4. For I=1 to n do	1	n+1	n+1
5. S=S+a[i]	1	n	n
6. return s;	1	1	1
7. }	0	~~0~~	0
Total		2n+3	

Asympotatic Notations :—

There are different kinds of mathematical notations used to represent time complexity. They are.,

1. Big-O
2. Omega
3. Theta
4. Little-oh
5. Little-Omega

1. Big-O Notation :— It is used to represent upper bound of algorithm run-time.

=> It measures the worst case time complexity.

=> It is the longest amount of time an algorithm.

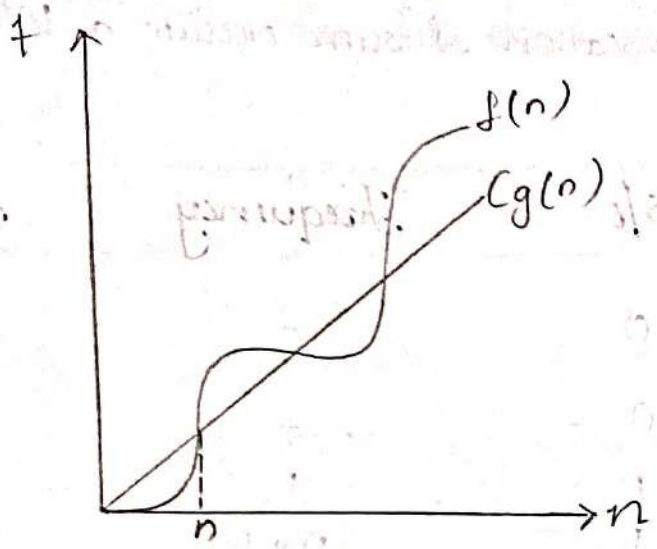
=> It can be used to describe the execution time required (or) the space used by the algorithm.

=> Big-O Notation defined as follows

$$f(n) = O(g(n))$$

where $f(n)$ is the function of time complexity and $g(n)$ is the function of inputs.

Here the c is constant.



$f(n) = Og(n)$

$f(n) \leq c * g(n)$ → formula

Example:— If $f(n) = 3n+2$ then prove that $f(n) = Og(n)$.

Let $f(n) = 3n+2$

$c = 4$, $g(n) = n$ (if $n=1$)

$3n+2 \leq 4*n$

$3n+2 \leq 4*1$

$3n+2 \leq 4$

$3(1)+2 \leq 4$

$5 \leq 4$ (x) false

Now If $n=2$

$3n+2 \leq 4*2$

$3(2)+2 \leq 8$

$8 \leq 8$ (✓) true

2. Omega Notation:— It is used to represent lower bound of an algorithm run-time.

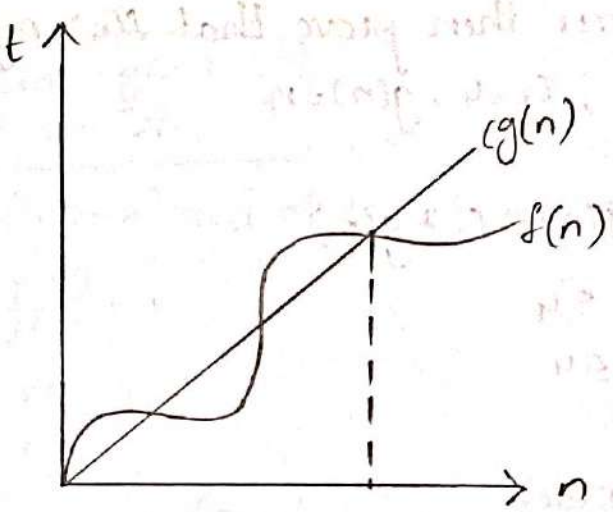
⇒ It measures the best case time. complexity.

⇒ It is the less amount of time of algorithm.

⇒ Omega Notation defined as follows

$$\boxed{f(n) = \Omega g(n)}$$

formula:— $f(n) \geq c * g(n)$



Example:— If $f(n) = 3n+2$ then prove that $f(n) = \Omega g(n)$

Let $f(n) = 3n+2$, $c=3$, $g(n)=n$

$f(n) \geq c * g(n)$

If $n=1$

$3n+2 \geq 3 * n$

$3+2 \geq 3$

$5 \geq 3$ (True)

3. Theta Notation:— It is used represent the running time between the upper bound and lower bound.

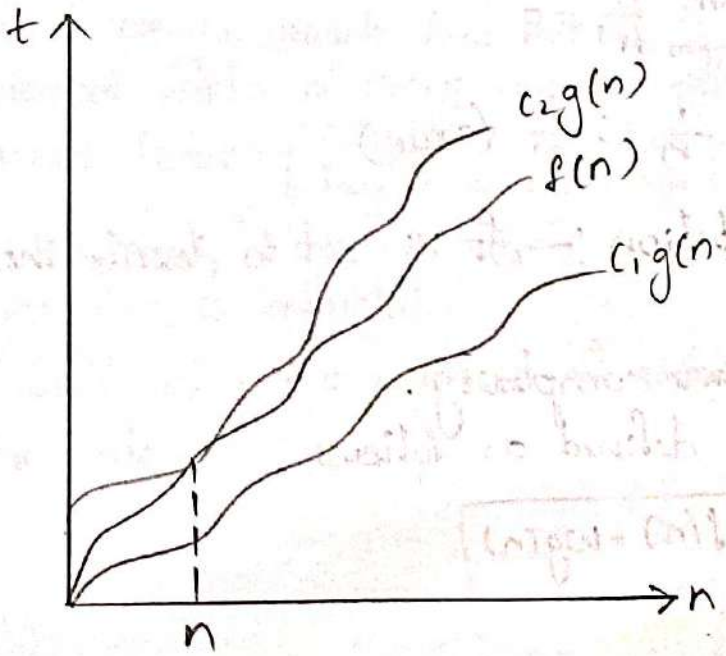
=> It measures the average time complexity.

=> It is not more not less amount of time of an algorithm.

=> Theta notation defined as follows.

$$f(n) = \Theta g(n)$$

formula:— $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$



Example:— If $f(n) = 3n+2$ then prove that $f(n) = o\,g(n)$.

let $f(n) = 3n+2$; $c_1 = 3$, $c_2 = 4$, $g(n) = n$

If $n = 1$

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

$$3 \leq 3n+2 \leq 4$$

$$3 \leq 3n+2 \leq 4$$

$$3 \leq 5 \leq 4$$

$3 \leq 5$ (True)

$5 \leq 4$ (False)

If $n = 2$

$$3(2) \leq 3(2)+2 \leq 4(2)$$

$6 \leq 8 \leq 8$ (True)

✓ ✓

4. Little-oh Notation:— It is used to describe an upper bound that can be tight, it is the worst case time complexity.

⇒ The little-oh defined as follows

$$\boxed{f(n) = o\,g(n)}$$

⇒ formula:— $$\boxed{\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0}$$

Example:— If $f(n) = n^2$ and $g(n) = n^3$ then prove that $f(n) = o\,g(n)$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$\lim_{n\to\infty} \frac{n^2}{n^3} = 0$$

$$\frac{1}{n} \Rightarrow \frac{1}{\infty} \text{ (True)}$$

5. Little Omega Notation:— It is used to describe the lower bound that can be tight.

⇒ It is best case time complexity.

⇒ The little-Omega defined as follows

$$\boxed{f(n) = \omega g(n)}$$

⇒ formula :— $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

Example :— If $f(n) = n^3$ and $g(n) = n^2$ then prove that $f(n) = \omega g(n)$.

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

$$\lim_{n \to \infty} \frac{n^2}{n^3} = 0$$

$$\frac{1}{n} \Rightarrow \frac{1}{\infty} \text{ (True)}$$

Examples :—

$f(n) = 10n^2 + 4n + 2$, $c = 11$

formula :— $f(n) \leq c * g(n)$

If $n = 1$ $\quad 10n^2 + 4n + 2 \leq 11 * n$

$10 + 4 + 2 \leq 11$

$16 \leq 11$ (False)

If $n = 2$ $\quad 10(2)^2 + 4(2) + 2 \leq 11 * 2$

$40 + 8 + 2 \leq 22$

$50 \leq 22$ (false)

If $c = 20$ then

$40 + 4 + 2 \leq 20$

$16 \leq 20$ (True)

AVL Tree :— AVL introduced in the year 1962 by Adelson velsky & landis.

⇒ AVL Tree is a height balanced binary search tree that means, AVL tree is also a binary search tree but it is a balanced tree.

⇒ In AVL tree balanced factor of every note is either 0, 1, -1.

⇒ formula ⇒ Balanced factor = [Height of left sub tree - height of right sub tree]

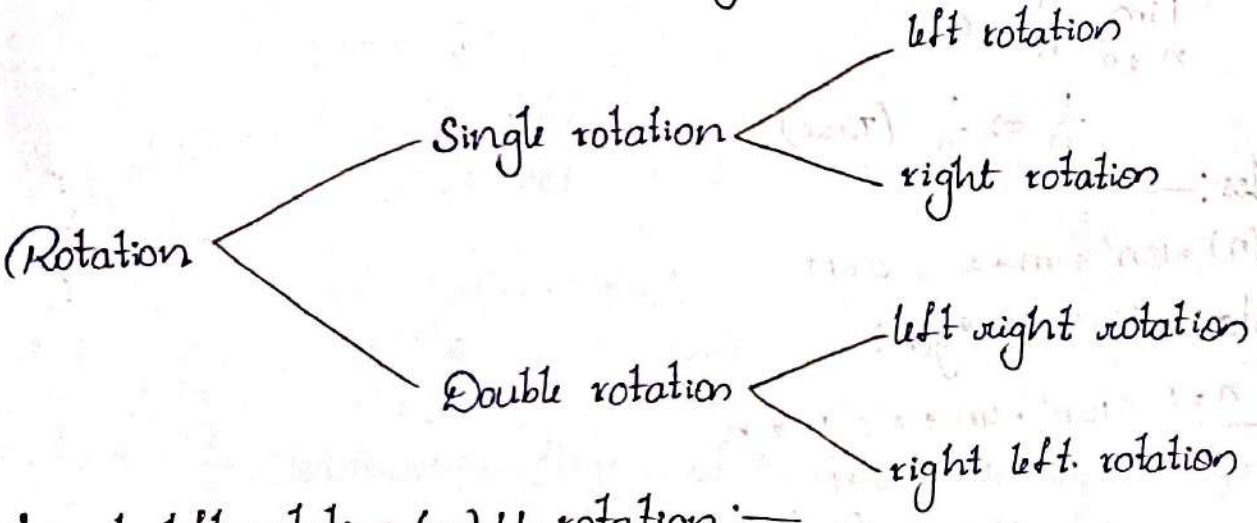
⇒ for all nodes the balance factor are 0, 1, -1 in a AVL tree. We can say that the AVL tree is balanced.

Note :— Please don't count the number of nodes, just count the height of the node.

AVL tree rotations :— AVL rotations are used to make the tree balance rotation is the process of moving nodes either left to right or right to left to make the tree balanced.

⇒ Rotations are used to convert the larger height to smaller height of a tree.

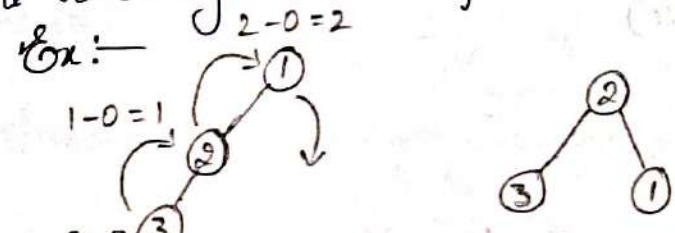
⇒ There are four rotations, and they are classified in to two types.



Single left rotation (or) LL rotation :—

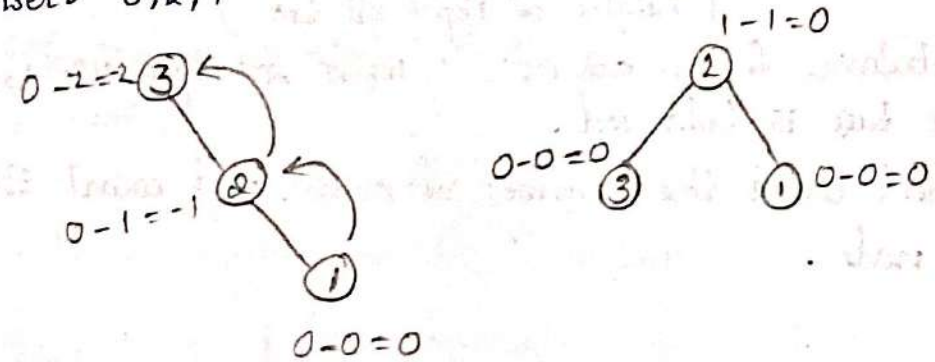
In this rotation every node moves one position to left from the current position, To understand single left rotation let us consider the following insertion operation in AVL tree insert 1, 2, 3.

Ex :—

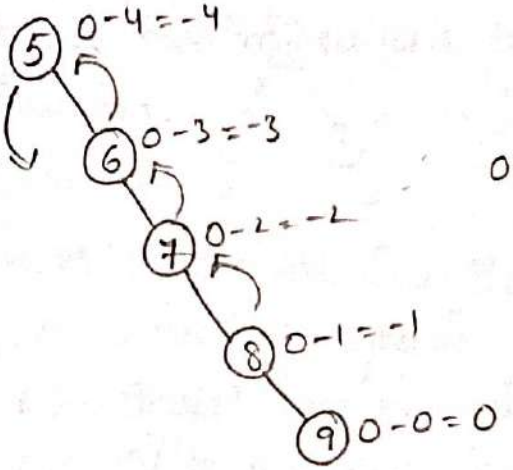


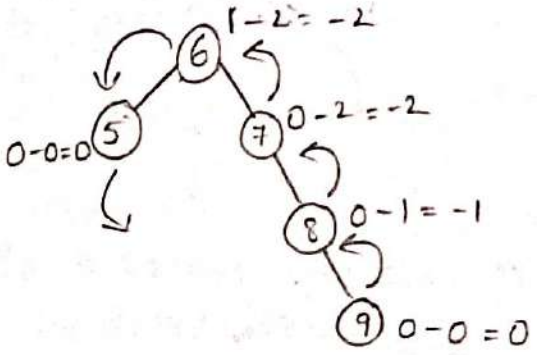
Single Right Rotation :— In RR Rotation every node one position to right from the current position. To understand RR rotation let us consider the following insertion operation in AVL tree.

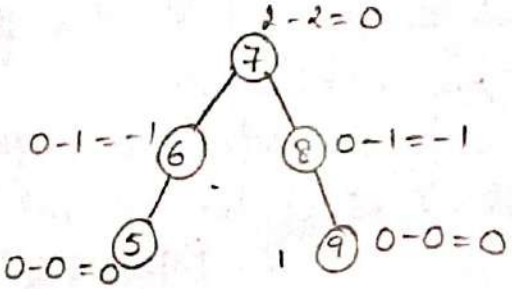
Insert 3, 2, 1



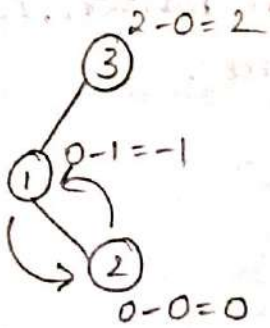
Insert 5, 6, 7, 8, 9



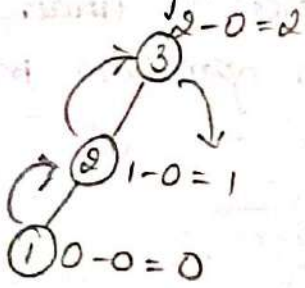




Left Right Rotation :— In this rotation is a sequence of single left rotation followed by a single right rotation in LR rotation at first every node moves one position to the left and one position to the right from the current position. To understand LR rotation, let us consider the following insertion operation in AVL tree insert 3, 1, 2.

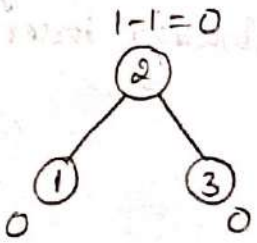


left rotation

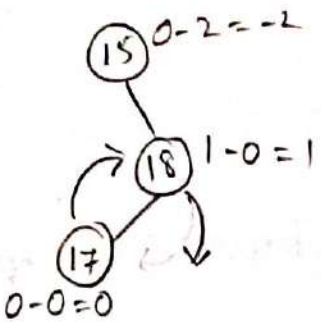


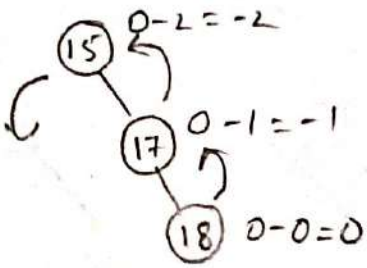
right rotation

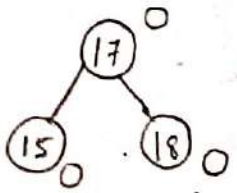
$\Rightarrow$



left right rotation

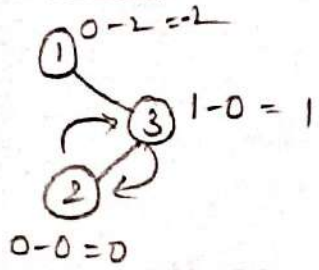


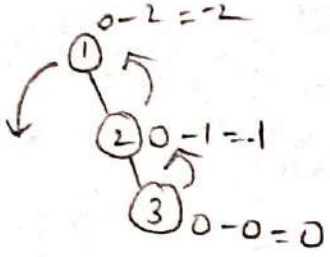


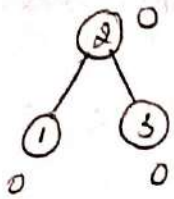


RL-rotation

1, 3, 2

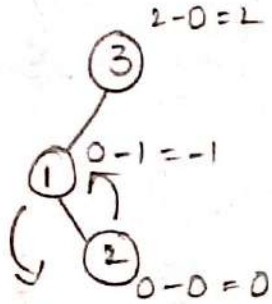


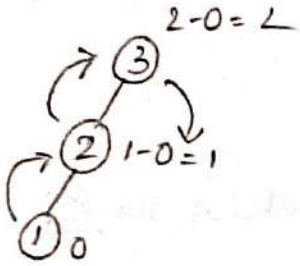


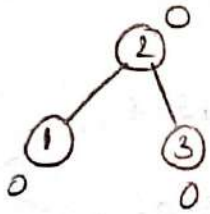


RL-rotation

3, 1, 2



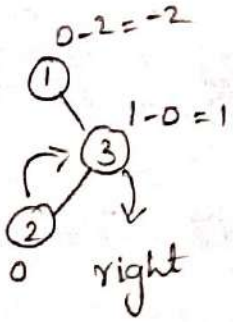


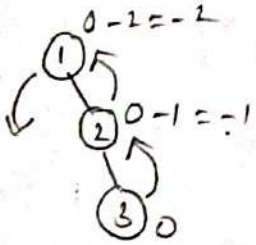


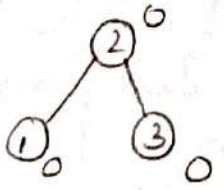
left rotation right rotation ⇒ LR rotation

Right Left rotation :— The RL rotations is a sequence of single right rotation followed by a single left rotatation in RL rotation. at first every node moves one position to the right and one position to the left from current position. To understand RL rotation, Let us consider the following insertion operation in AVL tree.

RL

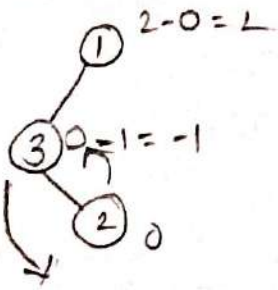


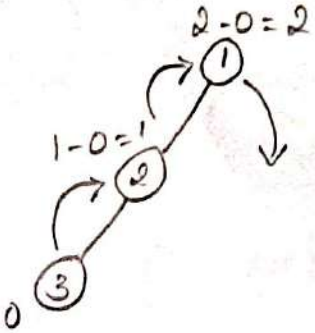


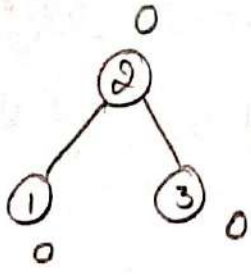


Right left rotation

LR - 1, 3, 2







AVL Tree Operations :—

There are two operations in AVL Tree.

- Insertion
- Deletion

Insertion :— In AVL Tree the insertion operation is performed by adding a element in specific location in AVL Tree.

Step-1 :— Insert new element into the tree using BST insertion logic.

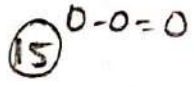
Step-2 :— After insertion check the balance factor of every node.

Step-3 :— If the balance factor of every node 0, 1, -1 then go for next operation.

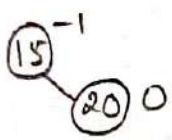
Step-4 :— If the balance factor of any node is other than 0, 1, -1, then the tree is said to be in balanced in this case we can perform suitable rotations to make it balance & go for next operation.

Example :— Construct an AVL Tree by inserting numbers:
15, 20, 24, 10, 13.

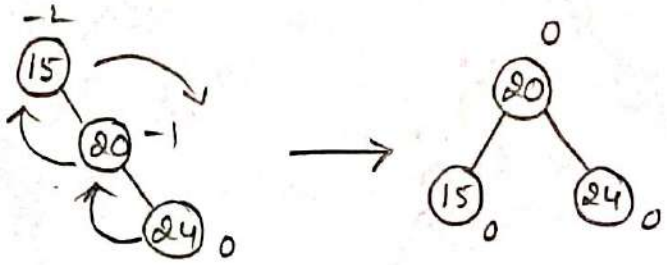
Step-1 :— Insert 15



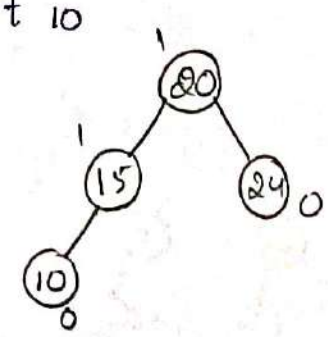
Step-2 :— Insert 20



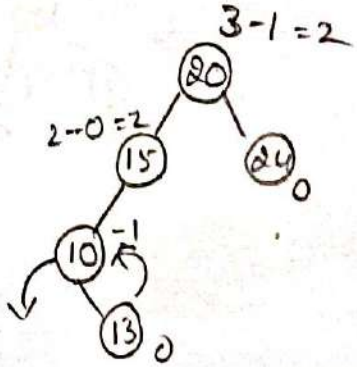
Step-3 :— Insert 24

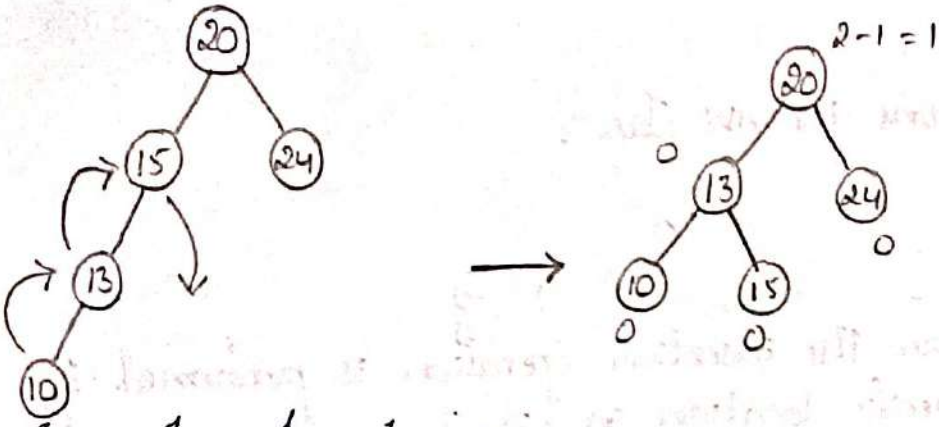


Step-4 :— Insert 10



Step-5 :— Insert 13

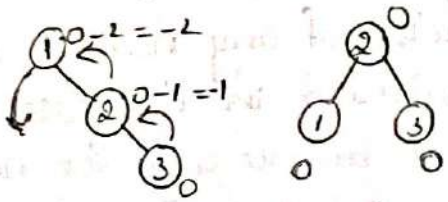


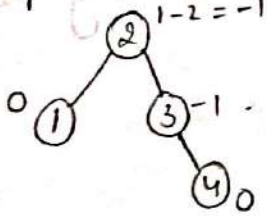


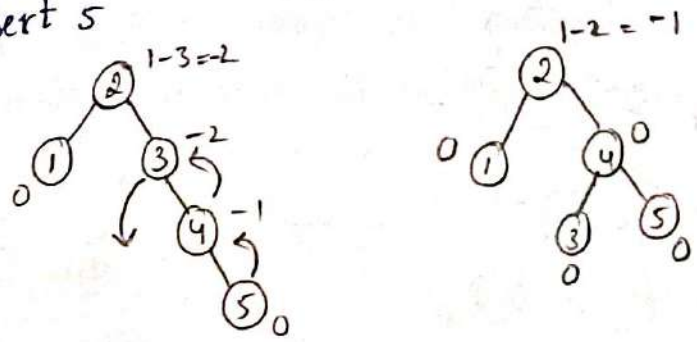
Example :- Insert 1 to 8

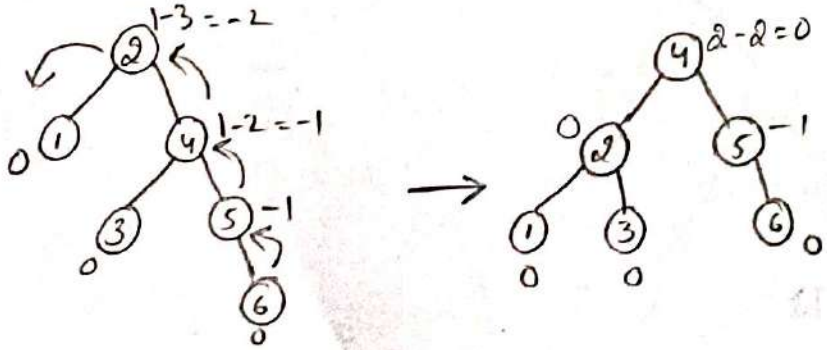
Step-1 :- Insert 1 

Step-2 :- Insert 2 

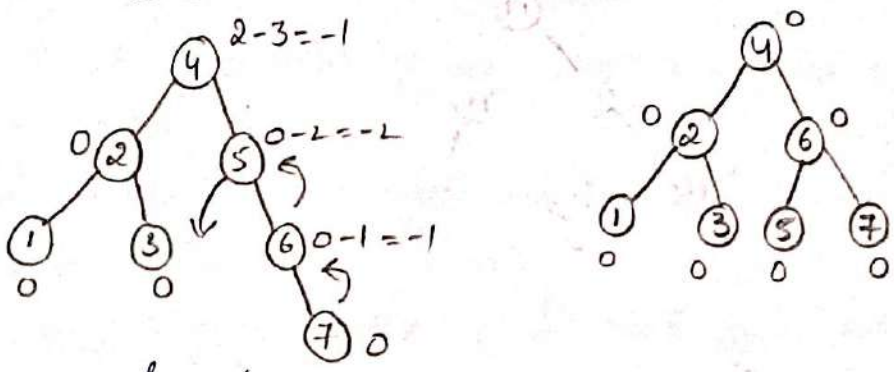
Step-3 :- Insert 3 

Step-4 :- Insert 4 

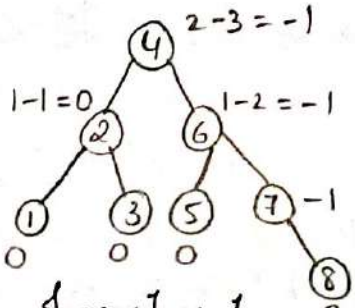
Step-5 :- Insert 5 

Step-6 :- Insert 6 

Step-7 :— Insert 7



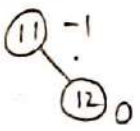
Step-8 :— Insert 8



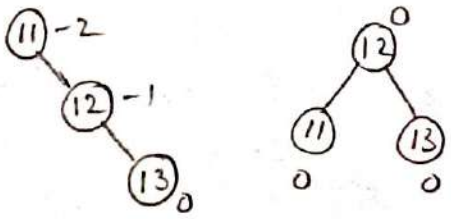
Example :— Insert 11 to 20.

Insert 11

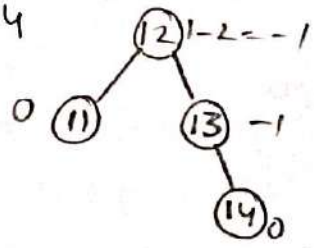
Insert 12



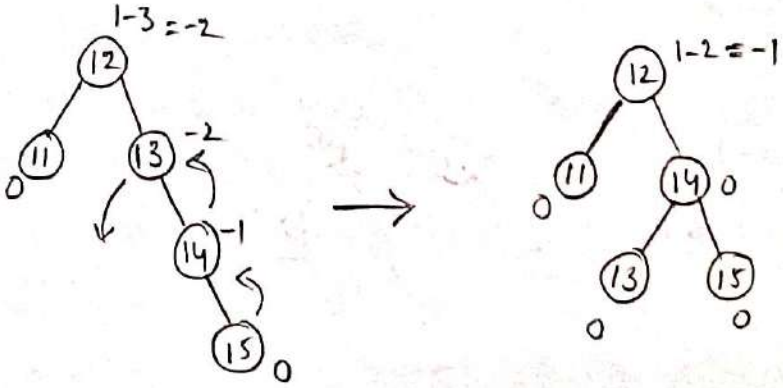
Insert 13



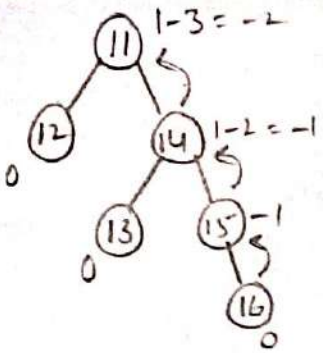
Insert 14

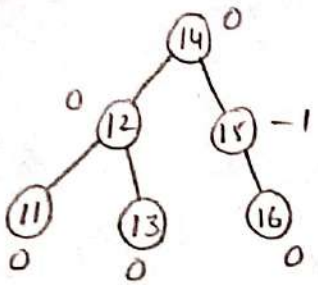


Insert 15

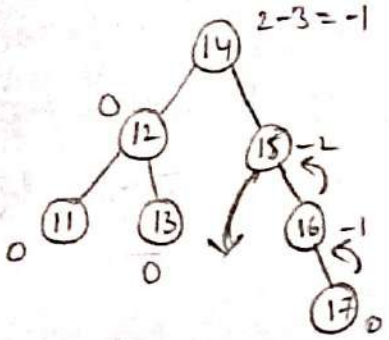


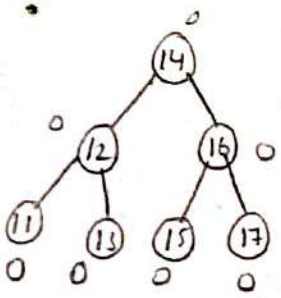
Insert 16



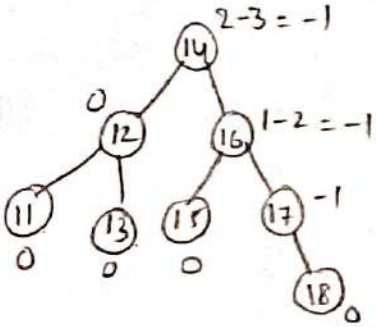


Insert 17

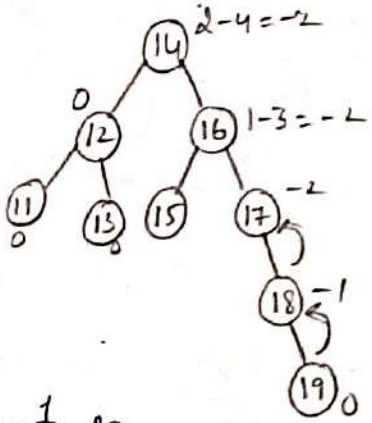


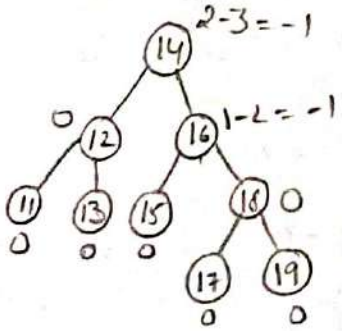


Insert 18

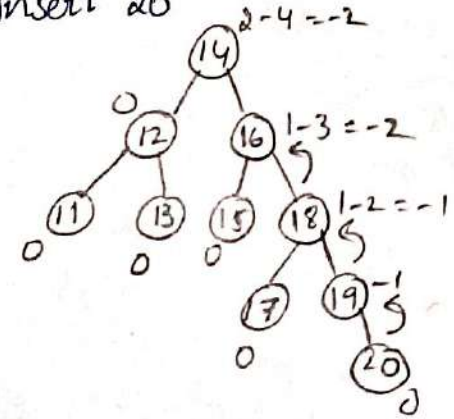


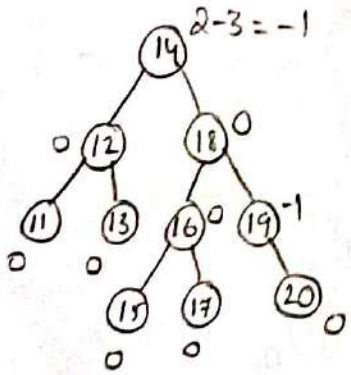
Insert 19





Insert 20





Deletion Operation in AVL Tree :—

The Delete operation in AVL Tree is similar to delete operation in binary search tree.

* After every deletion operation, we need to check the balance factor condition, if the tree is balance after deletion go for next operation. Otherwise perform suitable rotation to make the tree balance.

* There are two types of rotations:

1) Left Rotation
2) Right Rotation.

Here we will discuss Right-R Rotations and left Rotation are the mirror images of right rotation again left rotation and right rotation divided into six types. In R-rotations we have three types of rotations.

1) R_0
2) R_1
3) R_{-1}

In left rotations, there are three types of rotations.

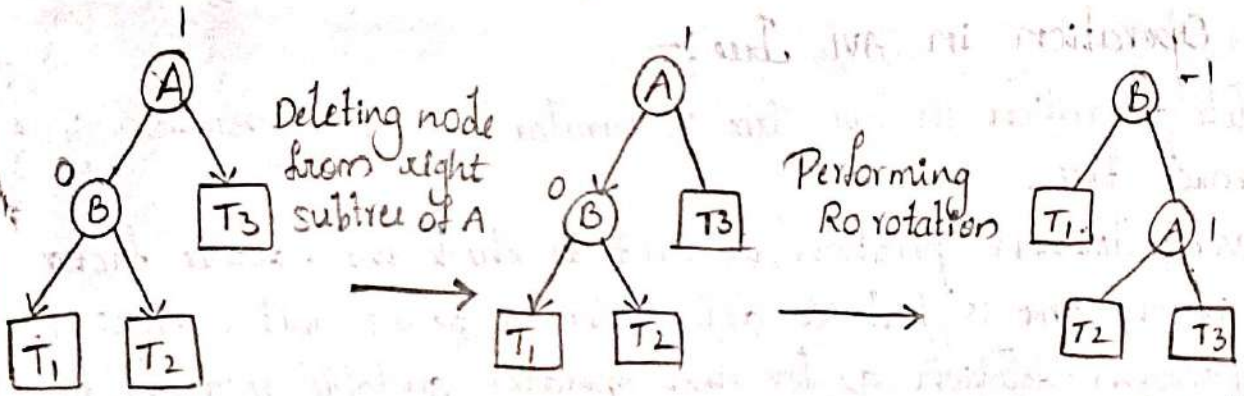
1) L_0
2) L_1
3) L_{-1}

1) R_0 - Rotation (Node B has Balance factor 0) :—

If the node B has balance factor, and the balance factor of node A disturbed upon deleting the node x, then the tree will be re-balanced by rotating tree using R_0 rotation.

The critical node A moved to its right and the node B becomes the root of the tree with T_1 as its left sub-tree.

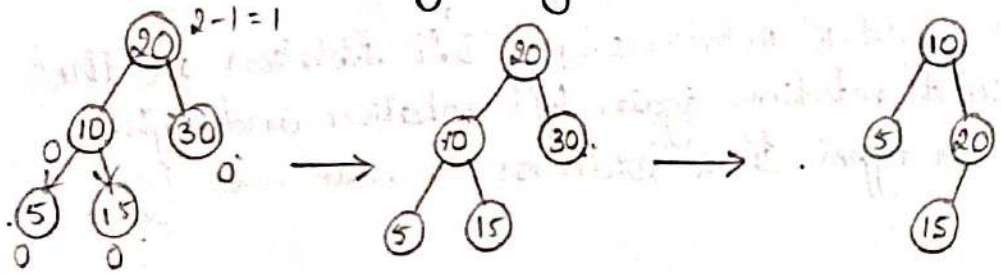
The subtrees T_2 and T_3 becomes the left and right subtree T_2 and T_3 becomes the left and right subtree of the node A. The process involved in R_0-Rotation is shown in the following image.



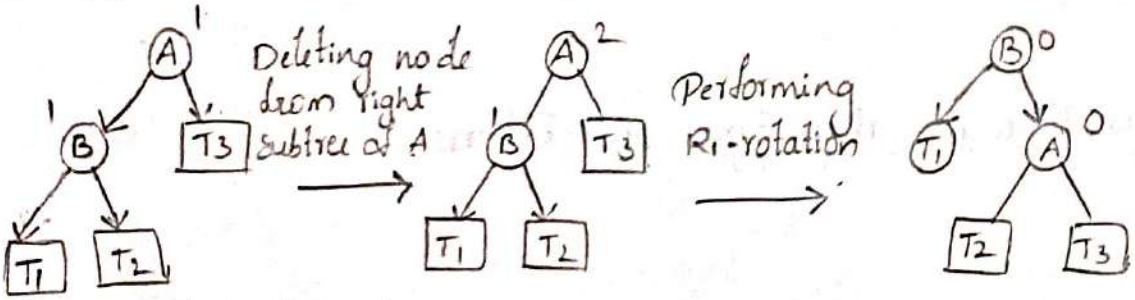
AVL Tree Non-AVL Tree R0 Rotated Tree

Example:— (i) Insert 20, 10, 30, 5, 15

(ii) Delete the node 30 from the AVL tree shown in the above following image.

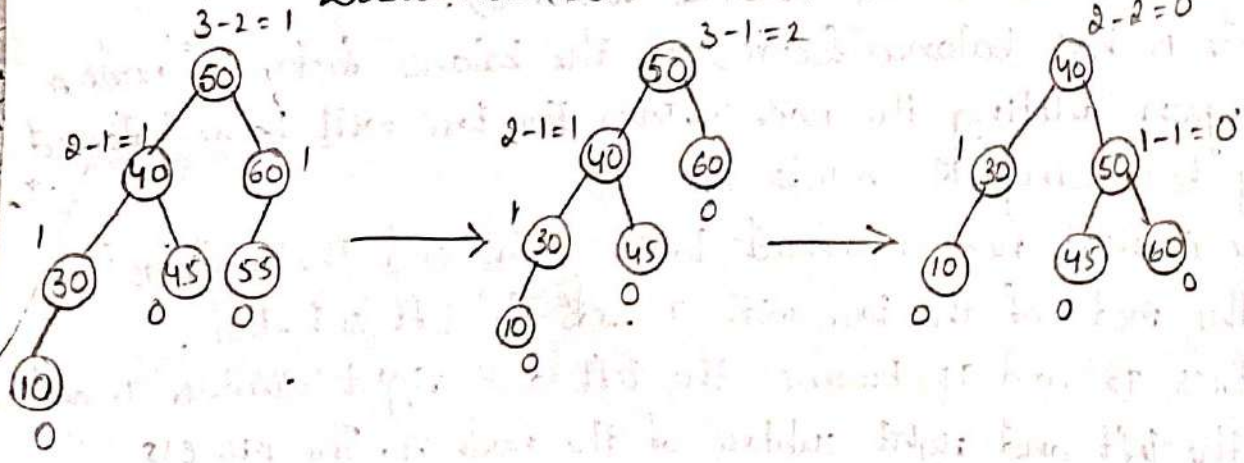


R1-Rotation:—



Example:— Insert 50, 40, 60, 30, 55, 45, 10

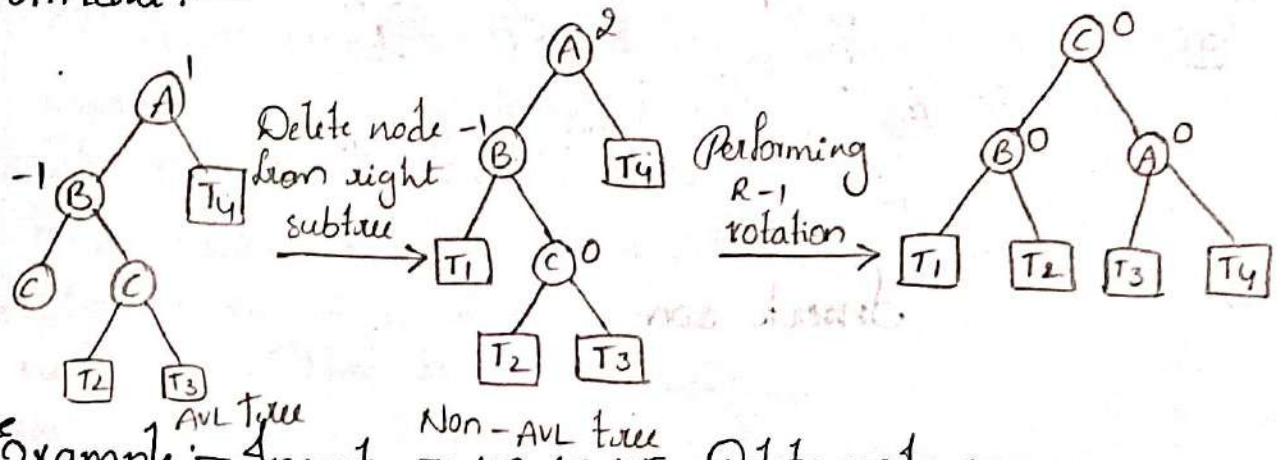
Delete node 55



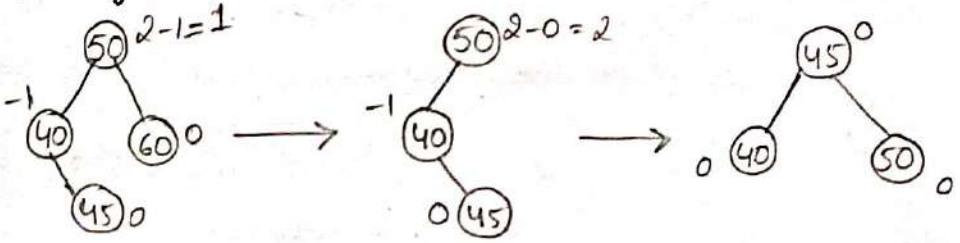
RL-Rotation:—

Example:- Insert 50, 40, 60, 45, Delete Node 60

Formula:—



Example:- Insert 50, 40, 60, 45, Delete node 60.

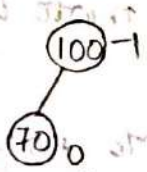


The L Rotations are the mirror image of R-Rotations.

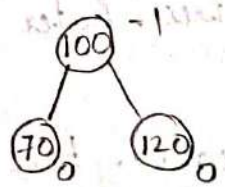
Example 1:- Insert 100, 70, 120, 50, 200, 45, 250, 10, 500.

Insert 100

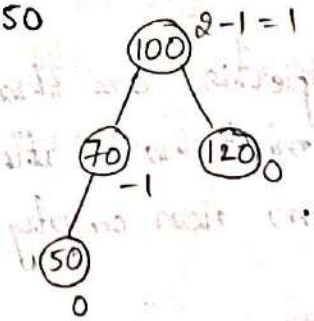
Insert 70



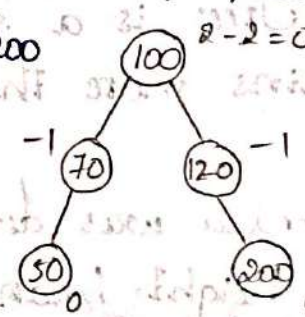
Insert 120



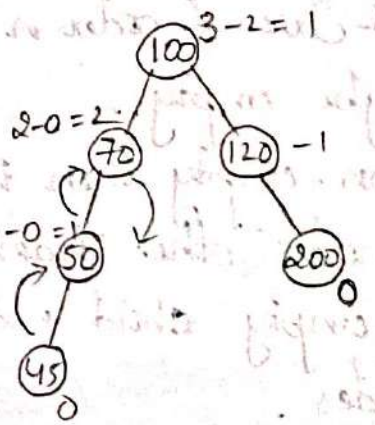
Insert 50

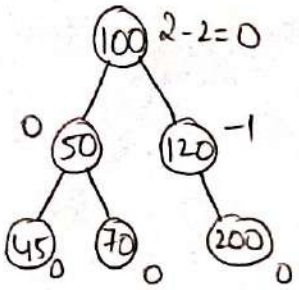


Insert 200

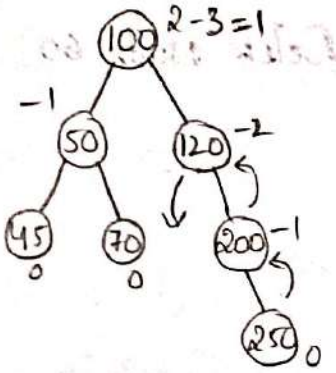


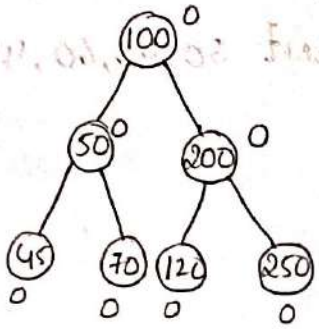
Insert 45



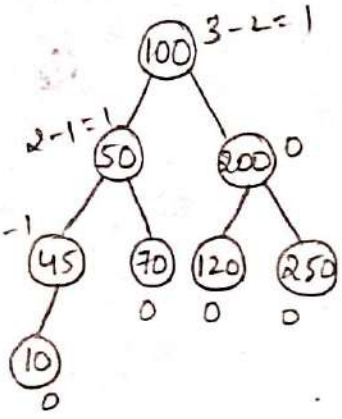


Insert 250

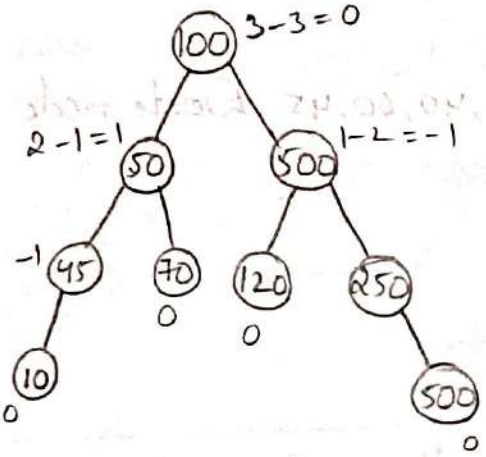




Insert 10



Insert 500



B-Trees :—

B-Tree is a self-balanced search tree in which every node contains more than one value and has more than two children.

B-Tree was developed by Bayes and Mc creight. The name height balanced M-way search tree. Later it was named as B-tree.

B-Tree of order n is an M-way search tree and hence maybe empty.

If non-empty then the following properties are that is all internal nodes other than the root node must have atleast $m/2$ non-empty child nodes and atmost m non-empty child nodes.

=> The root node must have atleast two child nodes and atmost m childnodes.
=> All external nodes are at the same level.
=> All nodes except root must have atleast $(m/2)-1$ keys, and maximum of $(m-1)$ keys.
=> The maximum number of keys in root node is 1 and all other nodes has the minimum number of $(m/2)-1$.
=> When a key is to be inserted in to a full node then the node is splitted into two nodes and the key with the middle value is inserted as the parent node. Incase parent is root then a new node root is created,
=> Keys are arranged in increasing order.

Max children m
Min children m/2
Max keys m-1
Min keys $(m/2)-1$

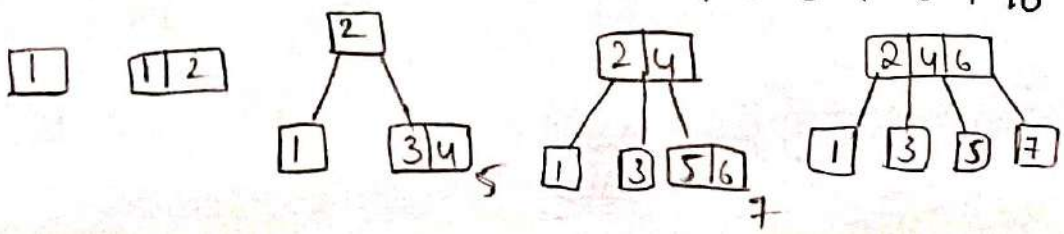
There are three orders in B-Tree

1. Order - 3
2. Order - 5
3. Order - 4

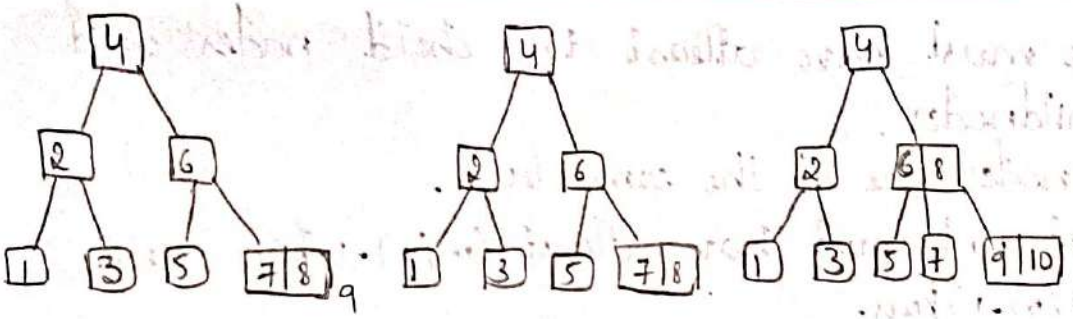
1. Order-3 :— An order-3 B-Tree might hold a maximum of six keys or a maximum of 7 keys.

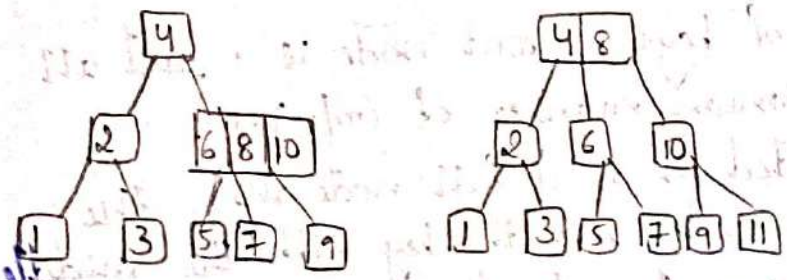
Max children 3
Min children $3/2 = 1.5$
Max keys $3-1=2$
Min keys $(3/2)-1 \Rightarrow 1.5-1 \Rightarrow 0.5$

Insert elements 1 to 11 (1 2 3 4 5 6 7 8 9 10 11)



5 7





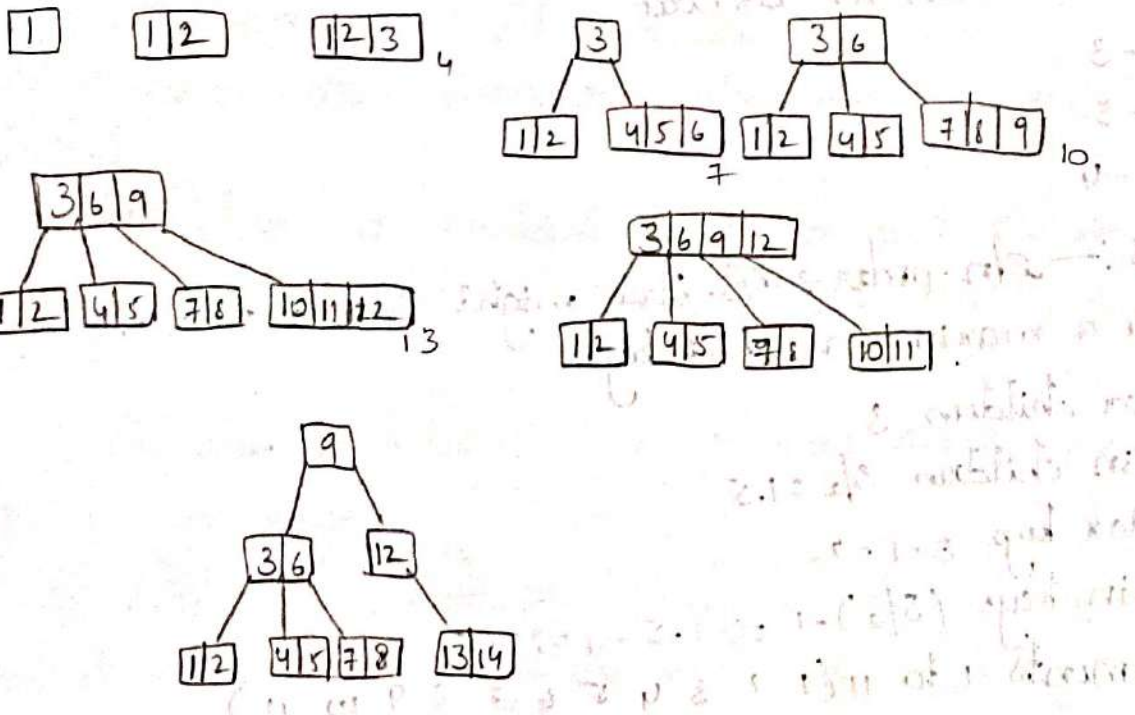
Order-4 :—

Max children 4

Min children $4/2 = 2$

Max keys $4-1 = 3$

Min keys $(4/2)-1 = 2-1 = 1$



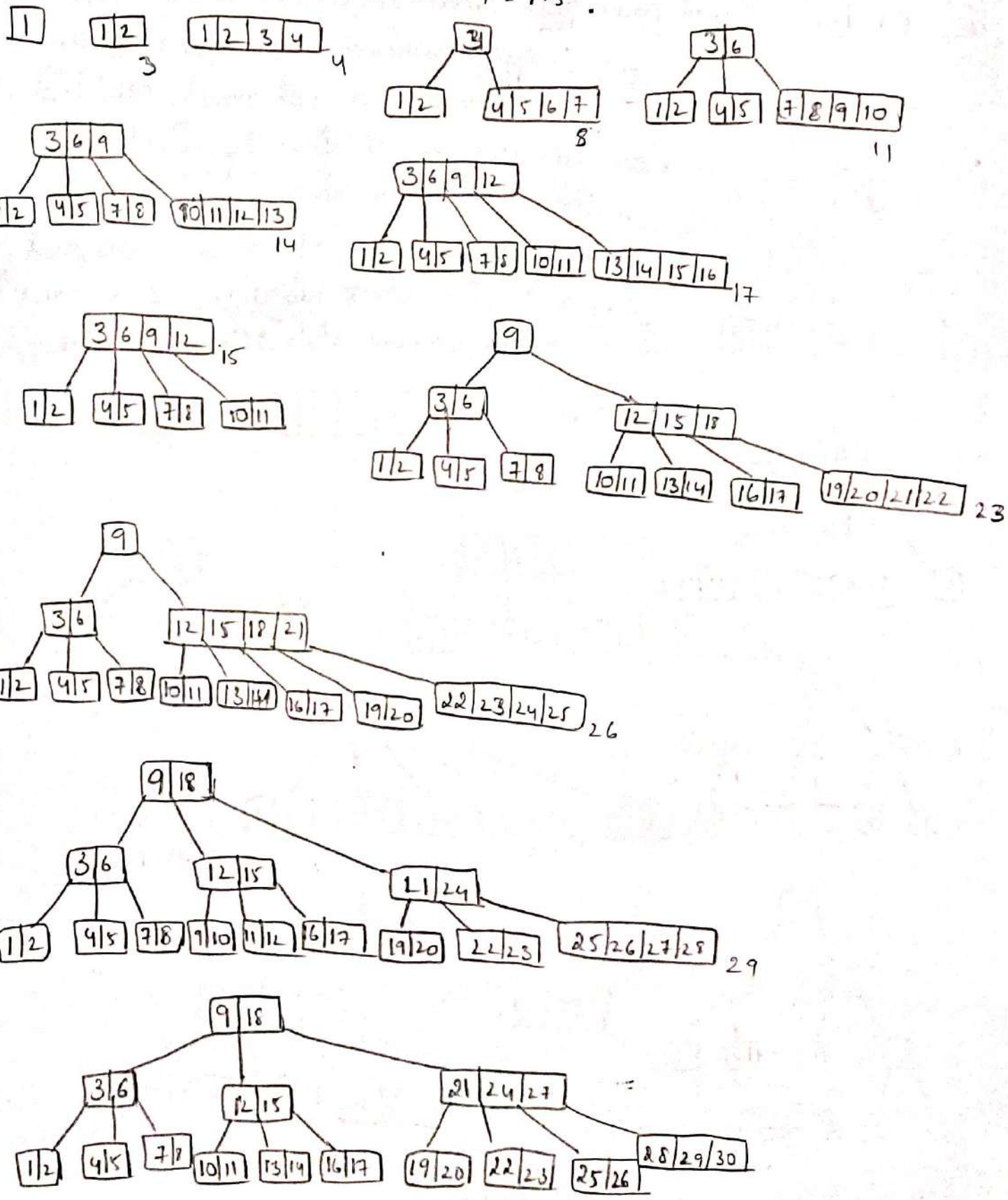
Order - 5 :—

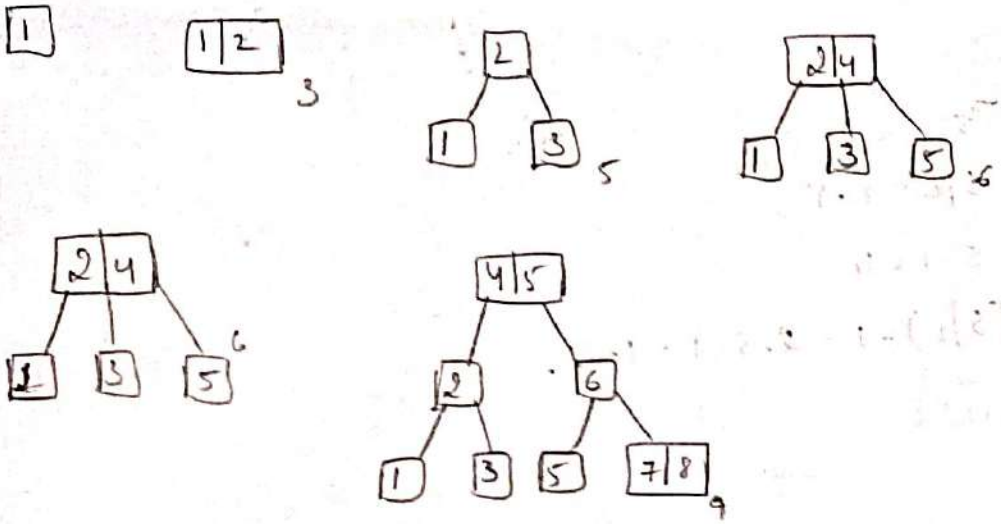
Max children 5

Min children $5/2 = 2.5$

Max keys $5-1 = 4$

Min keys $(5/2)-1 = 2.5-1 = 1.5$.




1
1 2
3
2
1 3
5
2 4
1 3 5
6
2 4
1 3 5
6
4 5
2 6
1 3 5 7 8
9
4
2 6 8
1 3 5 7 9
4
2 6 8
1 3 5 7 9 11
4 8
2 6 10
1 3 5 7 9 11 12
13
4 8
2 6 10 12
1 3 5 7 9 11 13 14
15
8
4 12
2 6 10 12
1 3 5 7 9 11 12 14

Deletion operation in B-Tree :—

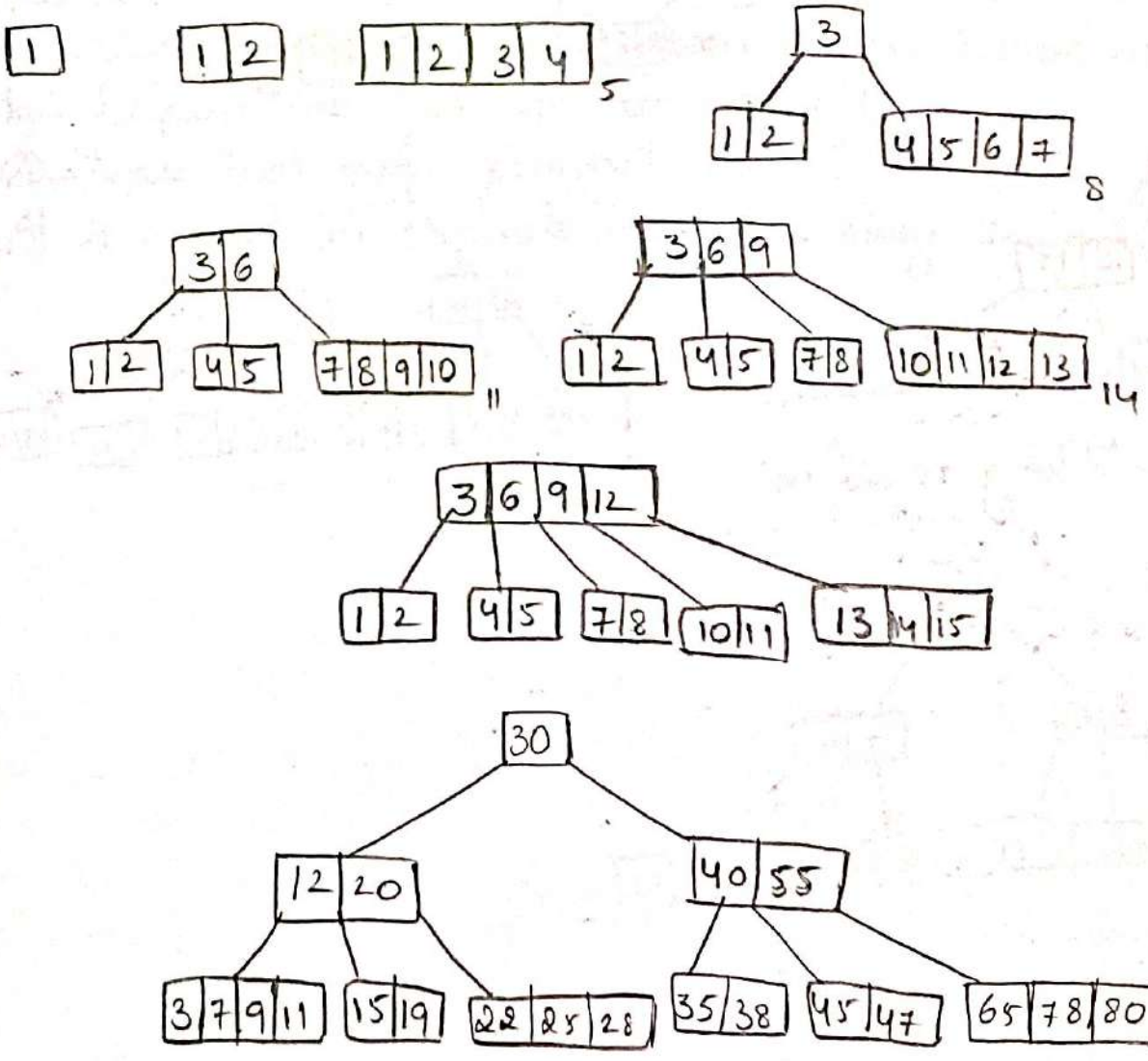
There are two po four cases and their are classified into two cases.

1. Deletion from the leaf node.
   - case (i) Node has more than minimum keys.
   - case (ii) Node has minimum keys.
2. Deletion from the non-leaf node.
   - Case (iii) Target node is in internal node.
   - Case (iv) Target node is in root node.

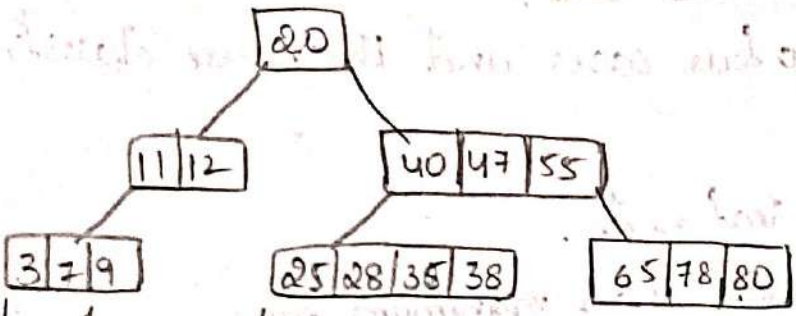
1. Deletion from the leaf node :—

Example-1 :— (i) order-5 elements 1 to 15.

(ii) Delete the nodes 7, 13.



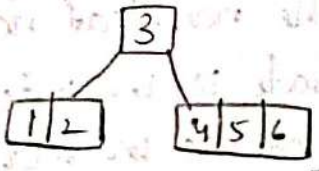
Delete the nodes 22, 15, 19, 45, 30



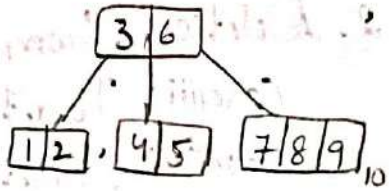
Insert 1 to 20 order - 4

Delete 16, 19, 12, 9

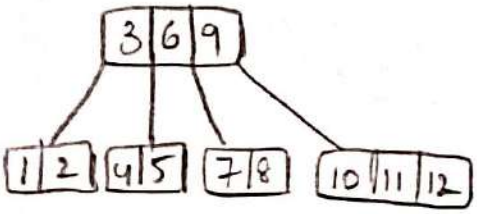
4



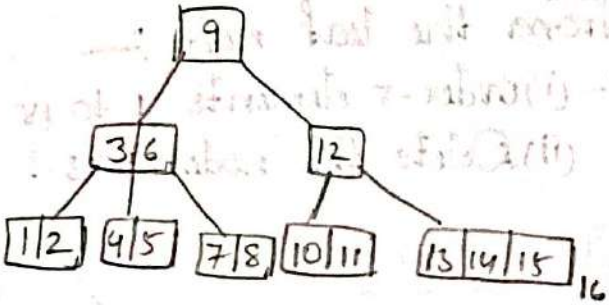
7



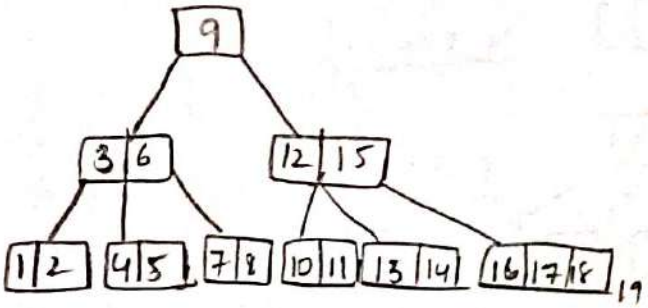
10



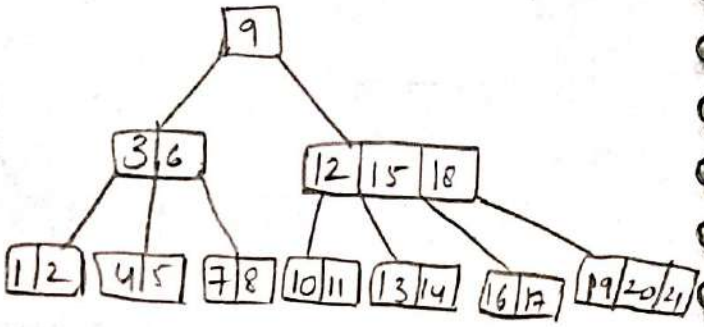
13



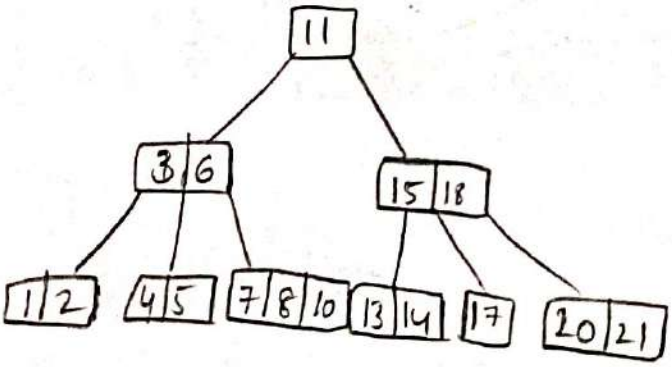
16



19



Now deleting 15 and 19



Applications of B-Tree :—

B-Trees are commonly used in applications where large amount of data need to be stored and retrieved efficiently. Some of the specific applications of B-Tree include is

1. Data base
2. File system
3. Operating system
4. Network routers
5. DNS (Domain name system) server
6. Compiler symbol tables.

Applications of AVL tree :—

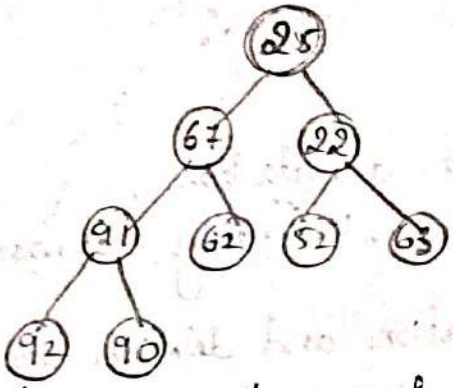
1. It is used to index huge records in a data base.
2. For all types in memory collections, including sets and dictionaries, AVL trees are used.
3. Database applications, where insertion and deletion are less common but frequent data look ups are necessary.
4. Software that needs optimized search.
5. It is applied in corporate areas and story line games.

Unit-II

Part-I

Heap Tree (Priority Queues): — Heap tree is a complete binary tree data structure that satisfies the heap property for every node, the value of its children is greater than or equal to its own value. Heaps are usually used to implement priority queues, where the smallest or largest element is always at the route of the tree.

Ex:— 25 67 22 91 62 52 63 92 90



There are two main types of heaps

1. MAX heap (maximum heap)
2. MIN heap (minimum heap)

1. MAX heap:— The root node contains maximum value, and the value decreases as you move down the tree.

2. MIN heap:— The root node contains minimum value and the value increases as you move down the tree.

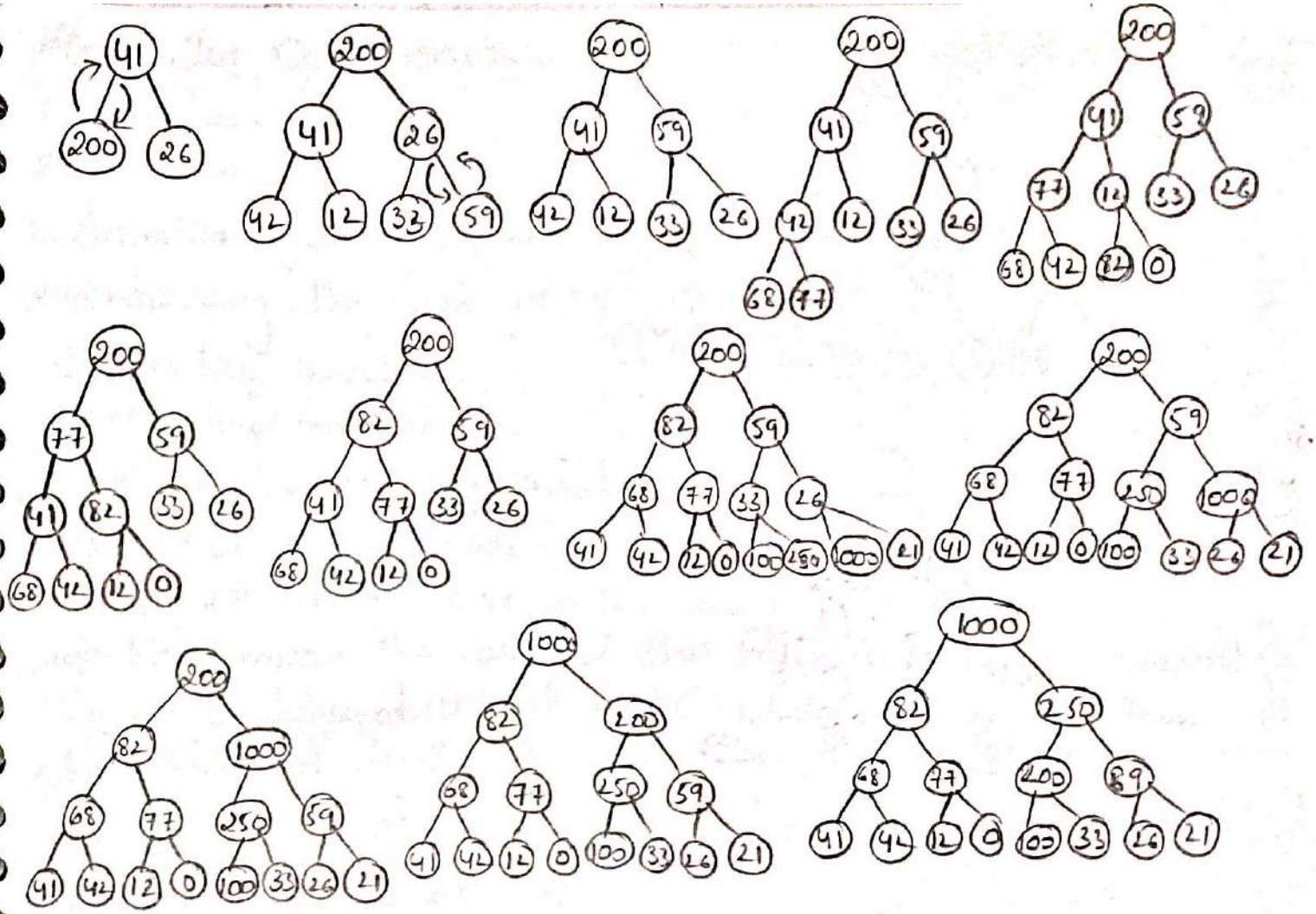
Example:— MAX HEAP

① 41 200 26 42 12 33 59 68 77 82

0 100 250 1000 21

⇒ 1000 250 200 100 91 82 77 68 59 42

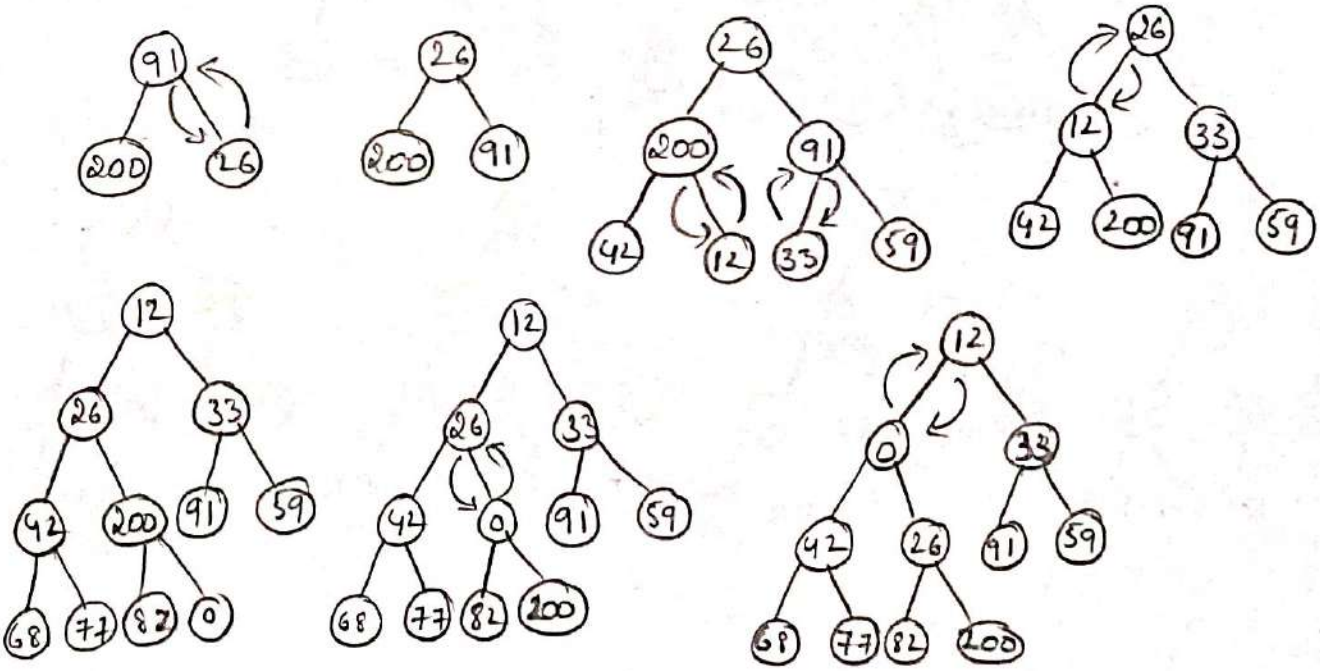
33 26 12 0

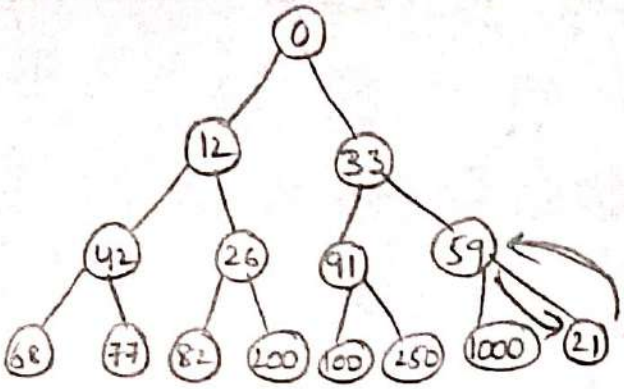


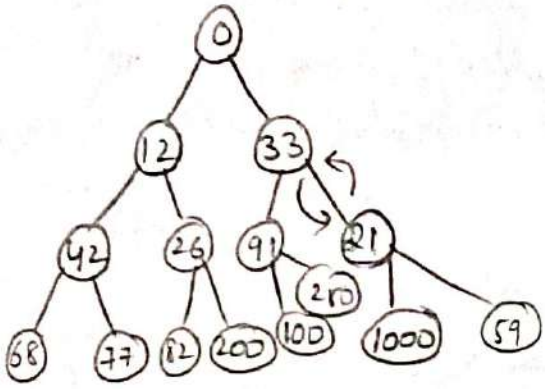
MIN HEAP :—

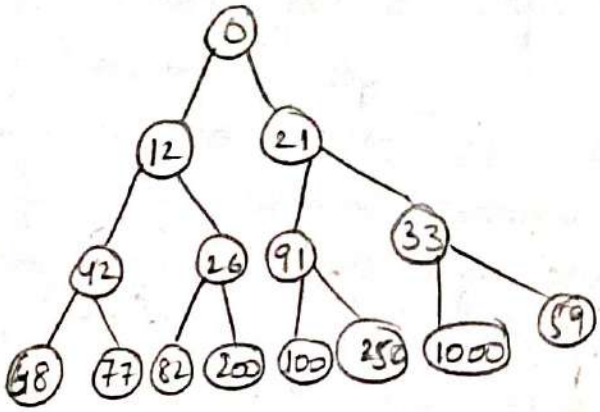
91 200 26 42 12 33 59 68 77 82 0 100

250 1000 21




0
12
33
42
26
91
59
68
77
82
200
100
250
1000
21


0
12
33
42
26
91
21
250
68
77
82
200
100
1000
59


0
12
21
42
26
91
33
59
68
77
82
200
100
250
1000

Heap-Tree Operations :—

1. Insertion
2. Deletion

1. Insertion :— Add a new element to the heap while maintaining the heap property.

(i) Min heap insertion

(ii) Max heap insertion

Algorithm for Max heap insertion :—

Step-1 :— Create a new node at the end of heap tree.

Step-2 :— Assign new value to the node.

Step-3 :— Compare the value of this child node with its parent.

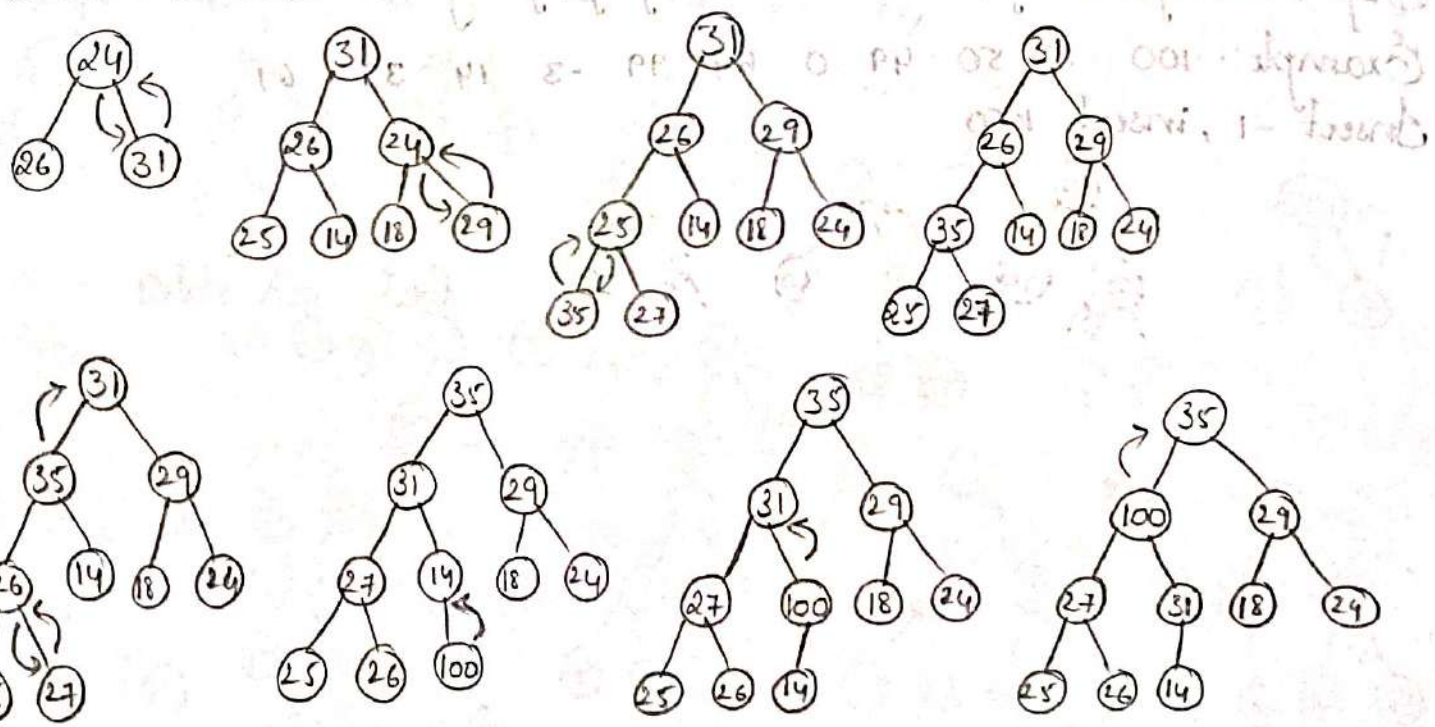
Step-4 :— If value of parent is less than child, then swap them.

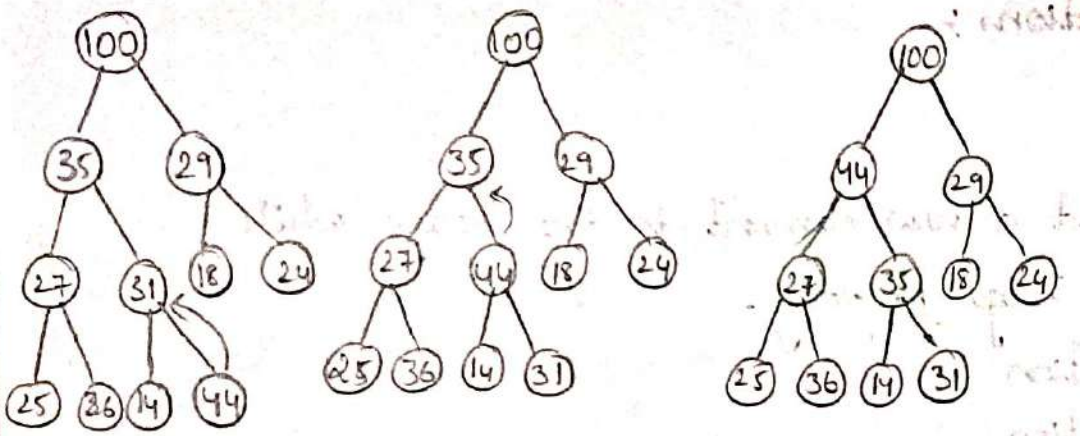
Step-5 :— Repeat step ③ and ④ until heap property holds.

Example-1 :— 24 26 31 25 14 18 29 35 27

Insert a new element 100

Insert a new element 44





Algorithm for Min heap insertion:—

Step 1:— Create a new node at the end of heap tree.

Step 2:— Assign new value to the node.

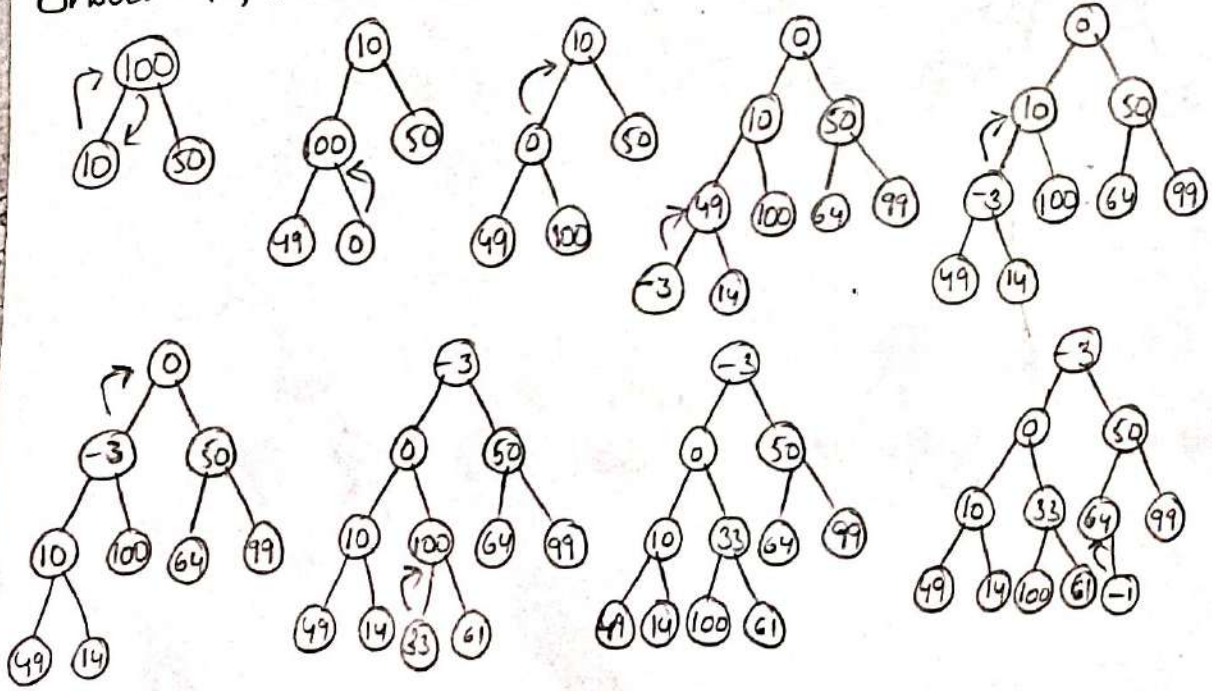
Step 3:— Compare the value of this child node with its parent.

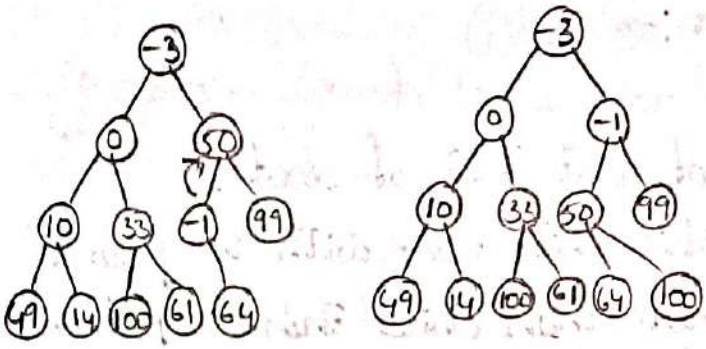
Step 4:— If value of parent is more than child, then swap them.

Step 5:— Repeat step ③ and ④ until heap property holds.

Example:— 100 10 50 49 0 64 99 -3 14 33 61

Insert -1, insert 1000





Deletion :— Remove the Minimum or Maximum element from the heap tree and return it.

Algorithm for Max heap deletion :—

Step-1 :— Remove root node.

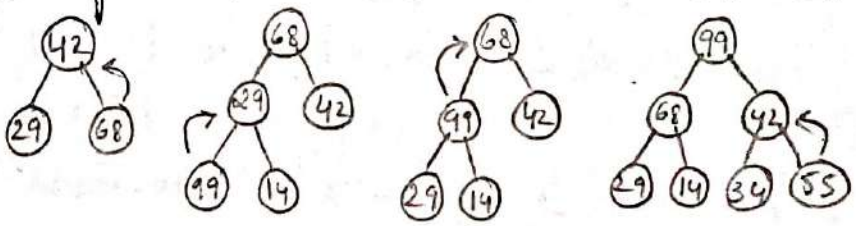
Step-2 :— Move the last element of last level of root.

Step-3 :— Compare the value of this child node with its parent.

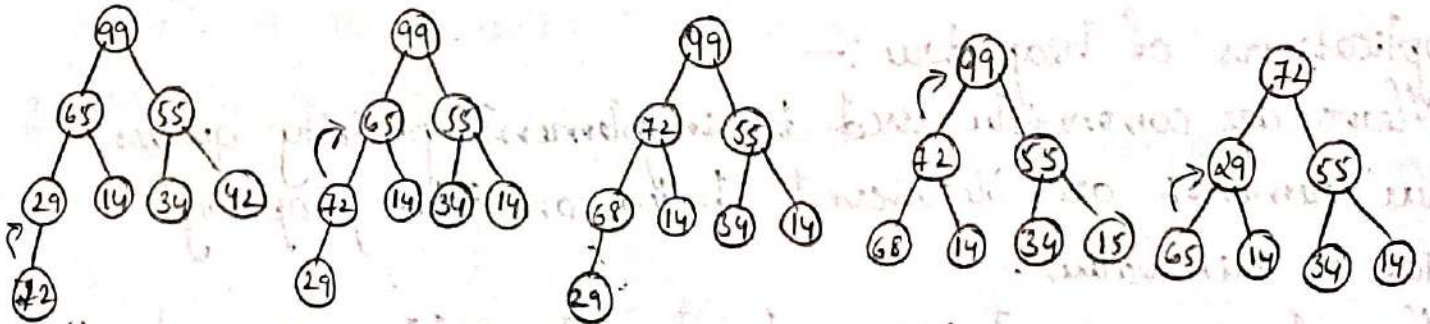
Step-4 :— If value of parent is less than child then swap them.

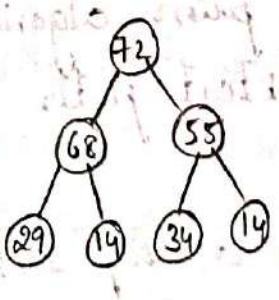
Step-5 :— Repeat step ③ and ④ until heap property holds.

Example :— 42 29 68 99 14 34 55 72



Delete root node





Algorithm for Min heap deletion :—

Step-1 :— Remove root node.

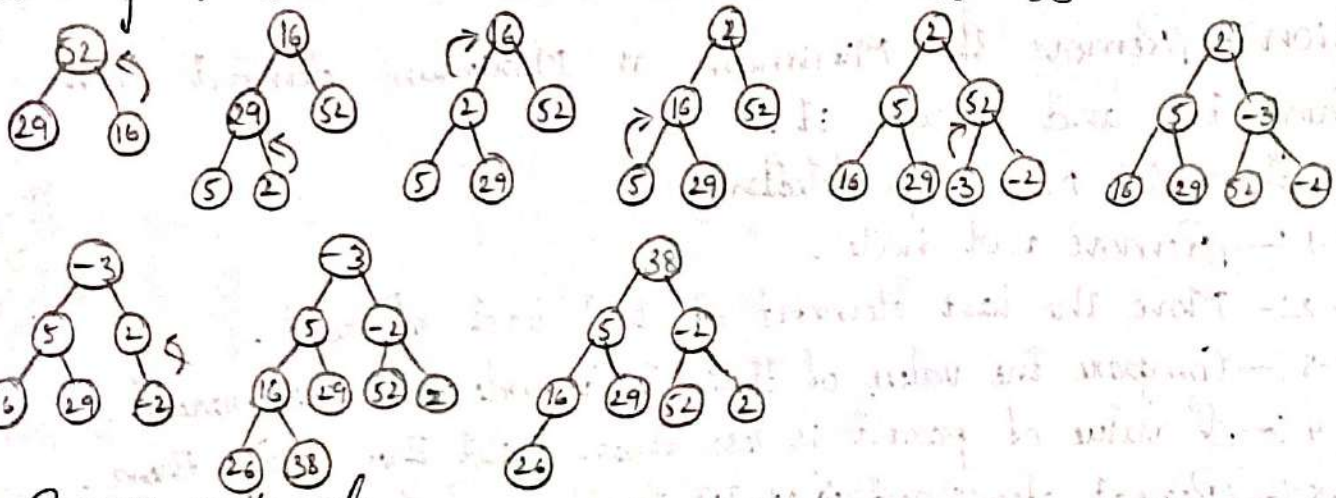
Step-2 :— Move the last element of last level of root.

Step-3 :— Compare the value of this child node with his parent

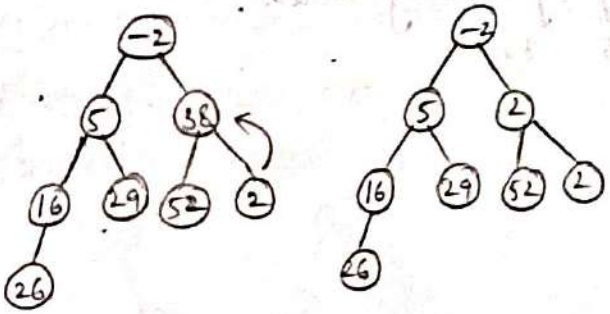
Step-4 :— If value of parent is more than child then swap them.

Step-5 :— Repeat step3 and 4 until heap property holds.

Example :— 52 29 16 5 2 -3 -2 26 38



Remove root node



Applications of heap tree :—

1. Heaps are commonly used to implement priority queues, where elements are retrieved based on their property (Max or Min value).
2. Heap tree are used in graph algorithms like prims algorithm and Dijkstra's algorithm for finding the shortest path and minimum spanning tree.
3. Heaps sort is a sorting algorithm that uses a heap sort an array in ascending or descending order.

Part-II

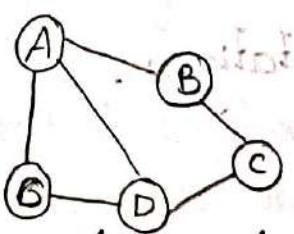
Graphs:— Graph is a Non-linear data structure it consist of finite set of node vertex and edges. The vertex are the elements and edges are ordered pair of connections between the nodes or vertex.

$$G = (V, E)$$

Terminologies :—

1. Graph representation
2. Vertex (or) Node
3. Edge
4. Adjacent Nodes
5. Path
6. Undirected graph
7. Directed graph
8. Weighted graph

1. **Graph Representation :—** Generally a graph is represented as a pair of sets that is V, E.
   V = Set of vertex (or) Node (or) Vertices
   E = Set of Edges.
2. **Node (or) Vertex :—** The elements of a graph are connected through edges.
   Ex :— A B C D E
3. **Edge :—** A path (or) a line between two vertices in a graph.



4. **Adjacent Nodes :—** Two nodes are called adjacent if they are connected through an edge.
   Ex :— A to B, A to D, A to C

5. Path :— It is a sequence of edges between two vertices It is essentially a traversal starting at one node and ending at another node.

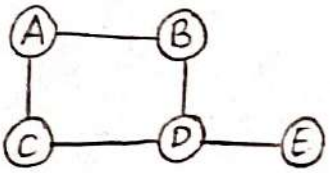
Ex:— A to B · B to E , C to D D to E

A to D D to E

6. Undirected graph :— An undirected graph is one where the edges do not specify a particular direction the edge are bidirectional.

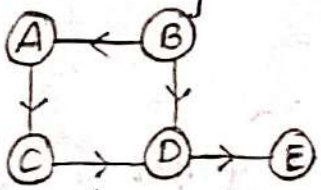
A to B , B to A
C to D , D to C
B to D , D to B
D to E , E to D

Ex:—



7. Directed graph :— A directed graph is one where the edges can be traversed in a specific direction only.

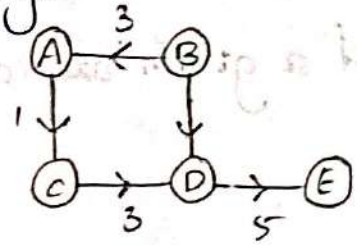
Ex:—



A to C to D to E
B to A to C to D to E

8. Weighted graph :— A weighted graph is one where the edges are associated with a weight. This is generally the cost to traverse the edge

Ex:—



AC + CD + DE
1 + 3 + 5 = 9
BA + AC + CD + DC
3 + 1 + 3 + 5 = 12.

Representations :—

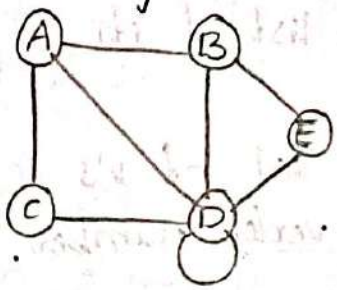
There are two graph representations.

1. Adjacency matrix
2. Adjacency List

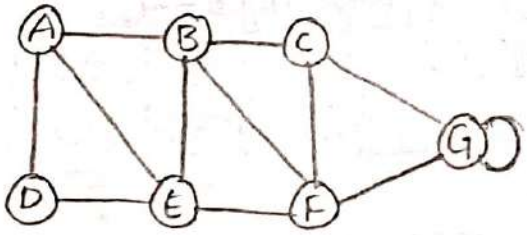
1. Adjacency Matrix :— In this representation graph can be represented using a matrix of size. total number of vertices by total number of vertices, means if a graph with four vertices can be represented using a matrix of 4×4 matrix size. In this matrix rows and columns both represent vertices.

* This matrix is filled with either one or zero.
* One represents there is an edge from row vertex to column and zero represented using a matrix of 4x4 matrix vertex and zero represents there is no edge from row vertex to column vertex.
* Adjacency matrix: Let G=(V,E) with 'n' vertices n≥1. The adjacency matrix of G is a two-dimensional nxn matrix, a, A(i,j)=1.

Example :—



	A	B	C	D	E
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

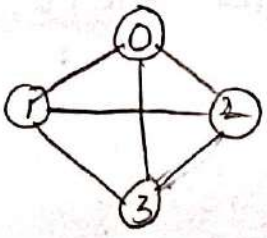


	A	B	C	D	E	F	G
A	0	1	0	1	1	0	0
B	1	1	1	0	1	1	0
C	0	1	0	0	0	1	1
D	1	0	0	0	1	0	0
E	1	1	0	1	0	1	0
F	0	1	1	0	1	1	1
G	0	0	1	0	0	1	1

2. Adjacency list :—

* In this representation every vertex of a graph contains list of adjacent vertices.
* The n rows of the adjacency matrix are represented as n chains
* The nodes in chain i represent the vertices that are adjacent to vertex i.
* It can be represented in two forms in one form array is used to store n vertices and chain is used to store it adjacencies.

Example :—

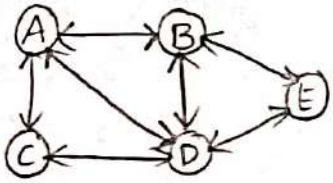


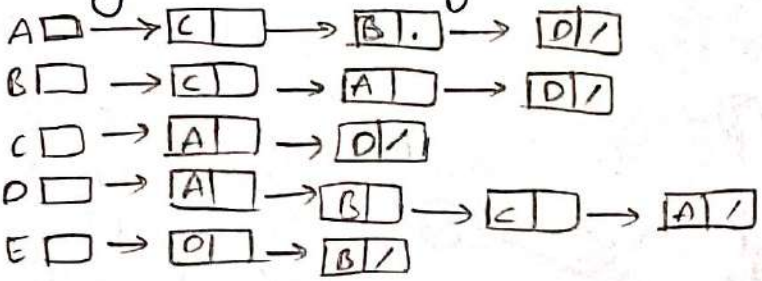
Consider the following directed graph representation implemented using linked list.

1. Graph representation using linked list.
2. Graph representation using array.

1. Graph representation using linked list :—

* In this representation, we keep a list of neighbours for each vertex in the graph.
* It means that each vertex in a graph has a list of its neighbours.
* Each array element points to a single linked list of v's neighbours and the array is indexed by the vertex number.





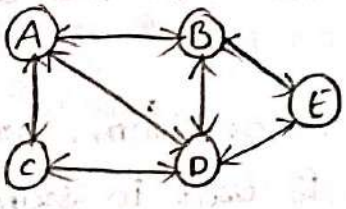
2. Graph Representation using array :—

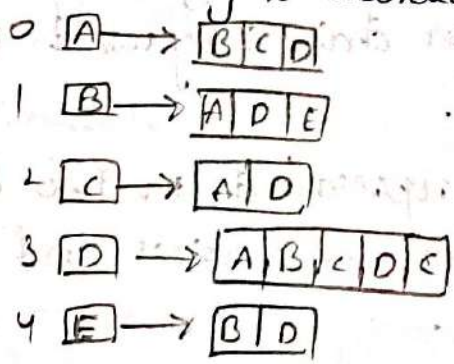
* An adjacency matrix represent a graph as a two dimensional array.
* Each vertex is assigned a distinct index in

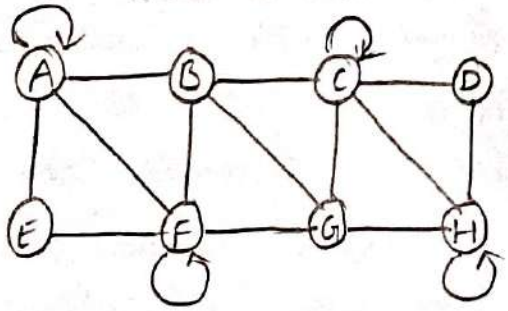
---

* If the graph is represented by the 2D array m.
* Then the edge from vertex i to vertex j is recorded at m[i][j]

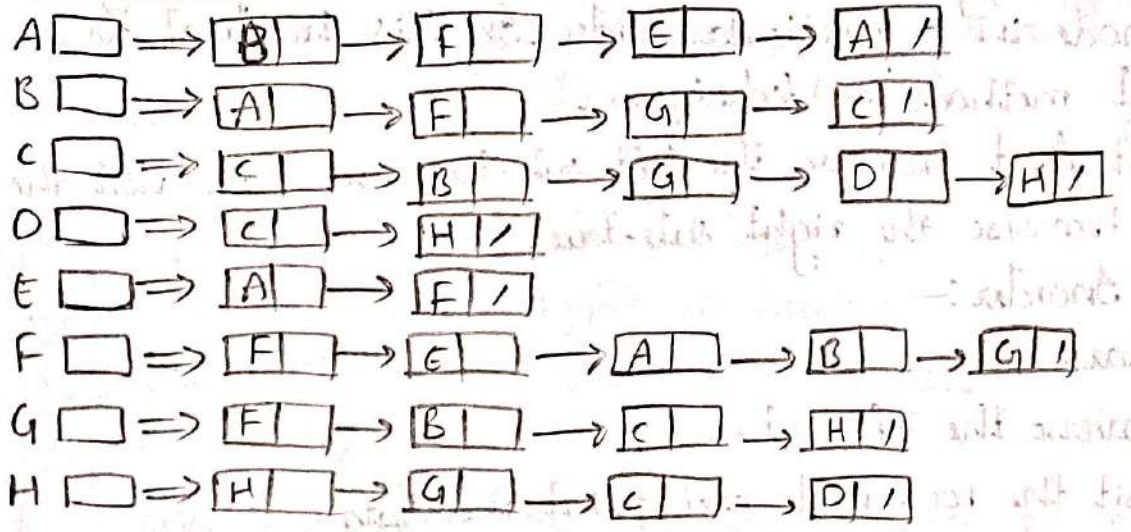
Ex :—



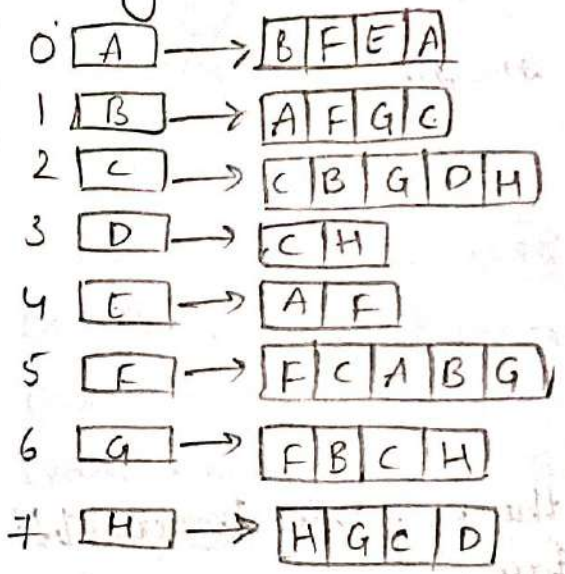




Linked list:-



Array:-



Basic Search and Traversal :- There are two types of search and three types of traversals.

1. Search techniques
   (i) BFS
   (ii) DFS
2. Traversal techniques
   (i) Inorder
   (ii) Preorder
   (iii) Postorder

Traversal:— Traversing a tree means visited on out putting the value of each node in a particular order. In this traversal we use the traversal methods (or) techniques.

(i) Inorder:— At first traverse the left sub-tree and then visit the root and then traverse the right sub-tree.

Algorithm for Inorder:—

Step-1 :— Start

Step-2 :— Traverse the left sub-tree.

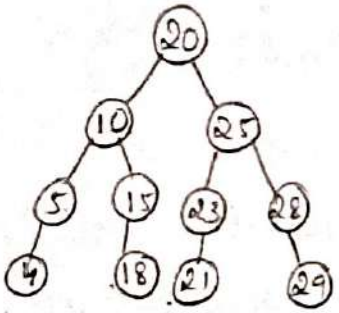
Step-3 :— Visit the root node and print the data.

Step-4 :— Traverse the right sub-tree.

Step-5 :— Stop.

Ex:— 20 10 25 5 15 23 28 4 18 21 29

L R Ri



4 5 10 15 18 20 21 23 25 28 29

(ii) Pre-order Traversal:— At first visit the root then traverse left sub-tree and then traverse the right-sub tree.

Algorithm for pre-order traversal:—

Step-1 :— Start

Step-2:— Visit the root and print the data

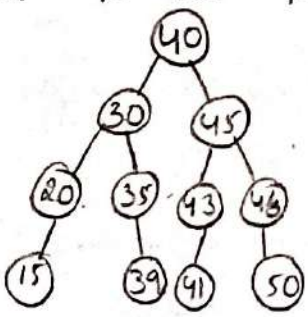
Step-3 :— Traverse left subtree

Step-4 :— Traverse the right sub-tree

Step-5 :— Stop.

Ex:— 40 30 45 20 15 35 39 43 41 46 50

R L Ri



40 30 15 20 35 39 45 41 43 46 50

(iii) Post-order traversal:— At first traverse left subtree then traverse right sub-tree and then visit the root.

Algorithm for post order traversal:—

Step-1 :— Start

Step-2 :— Traverse the left-sub tree

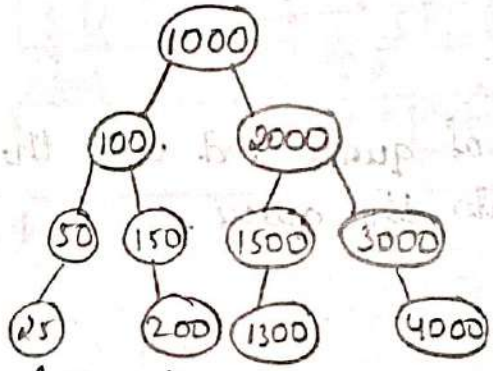
Step-3 :— Traverse the right sub tree

Step-4 :— Visit the root and print the data

Step-5 :— Stop.

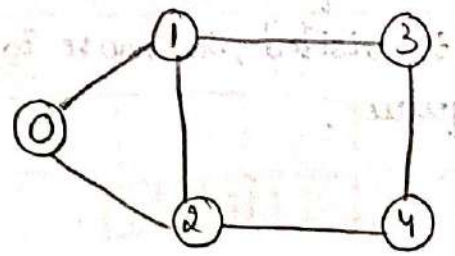
Ex:— 1000 100 2000 50 150 1500 3000 25 200 13000 14000

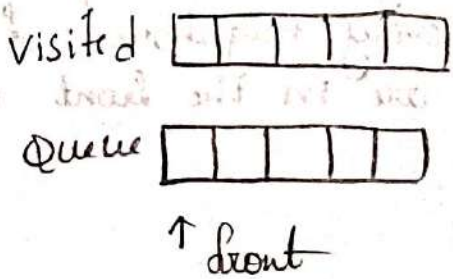
L Ri R



BFS (Breadth First Search):— It is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level.

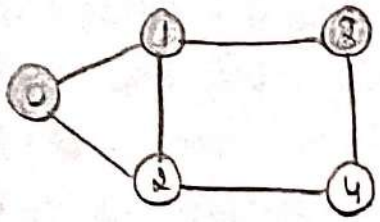
* If start at a specific vertex and visit all of its neighbours before moving on to the next level of neighbours.
* BFS is commonly used in algorithms for path finding connected components, and shortest path problems in a graph.
* In BFS we are using a "queue".





Step-1:— Initially queue and visited arrays are empty.

Step-2:— Push node 0 in to queue and mark it visited.

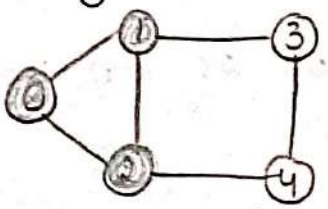


visited | 0 | 1 | | | |

Queue | 0 | | | | |

↑ front

Step-3:— Remove node 0 from the front of Queue and visit the unvisited neighbours and push them in to the queue.

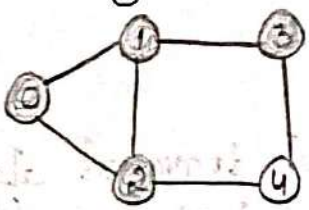


visited | 0 | 1 | 2 | | |

Queue | 1 | 2 | | | |

↑ front

Step-4:— Remove node 1 from the front of queue and visit the unvisited neighbours and push them into the queue.

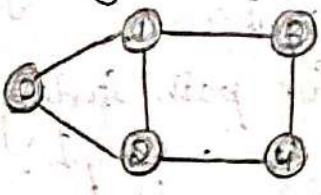


visited | 0 | 1 | 2 | 3 | |

Queue | 2 | 3 | | | |

↑ front

Step-5:— Remove node 2 from the front of queue and visit the unvisited neighbours and push them in to queue.



visited | 0 | 1 | 2 | 3 | 4 |

Queue | 3 | 4 | | | |

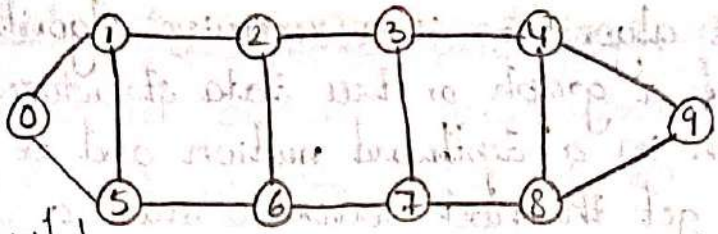
↑ front

Step-6:— Removed node 3 from the front of queue and visit the unvisited neighbours and push them in to queue. As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.

visited | 0 | 1 | 2 | 3 | 4 |

Queue | 4 | | | | |

↓ front



Visited

Queue

1. Visited

0									

0									

↑ front Queue

2. visited

0	1	5							

1	5								

↑ front Queue

3. visited

0	1	5	2						

5	2								

↑ front Queue

4.

0	1	5	2	6					

Visited

2	6								

↑ front Queue

5. visited

0	1	5	2	6	3				

6	3								

↑ front Queue

6. visited

0	1	5	2	6	3	7			

3	7								

↑ front Queue

7. visited

0	1	5	2	6	3	7	4		

7	4								

↑ front Queue

8. visited

0	1	5	2	6	3	7	4	8	

4	8								

↑ front Queue

9. visited

0	1	5	2	6	3	7	4	8	9

8	9								

↑ front Queue

10. visited

0	1	5	2	6	3	7	4	8	9

9									

↑ front Queue

11.

0	1	5	2	6	3	7	4	8	9

Visited

↑ front

Queue

DFS (Depth First Search):— The algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depthward motion and we uses a stack to remember to get the next verten to start a search, when a dead end occurs in any iteration.

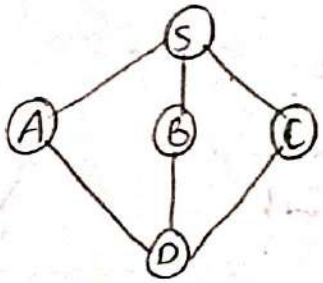
Rule 1:— Visit the adjacent unvisited verten, mark it as visited, display it, push it in a stack.

Rule 2:— If no adjacent verten is found, pop-up (or) remove a verten from the stack it will remove all the vertices from the stack, which do not have adjacent vertices.

Rule 3:— Repeat rule 1 and rule 2 until the stack is empty.

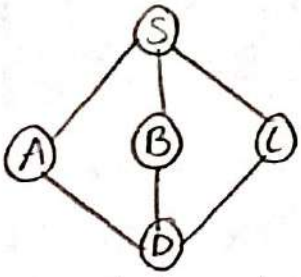
Example-1

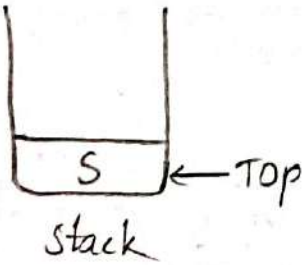
Step 1:—



Initialize the stack.

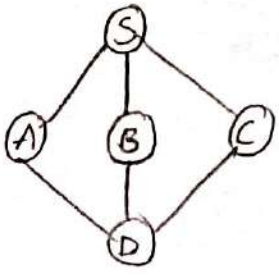
Step 2:—

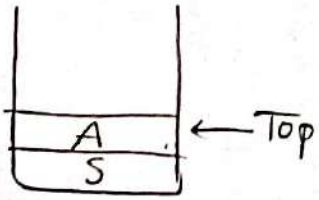




Mark 's' as visited and put it on to the stack explore any unvisited adjacent node from 's'. We have three nodes and we can pick any of them. For this ex we shall take the node in an alphabetical order.

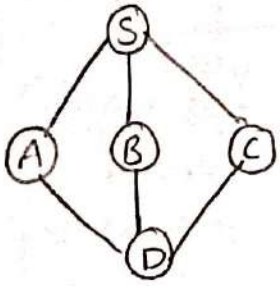
Step-3:—

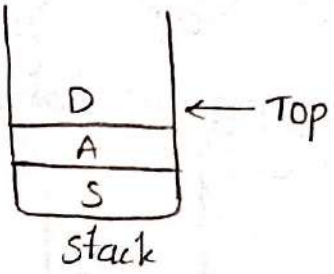




Mark 'A' as visited and push it on to the stack. explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are connected for unvisited nodes only.

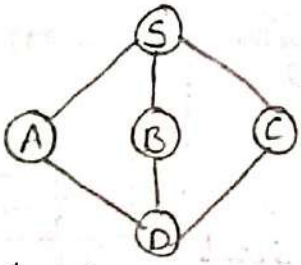
Step-4 :—

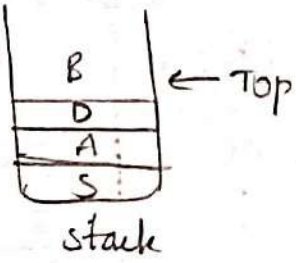




Visit D and mark it as visited and push on to the stack here we have B and C nodes which are adjacent to D and both are unvisited however we shall again choose in an alphabetical order.

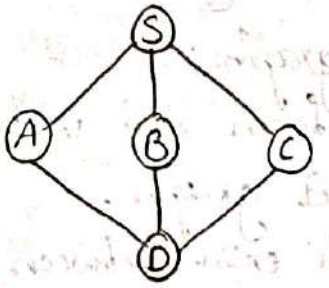
Step 5 :—

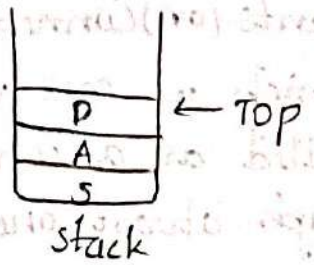




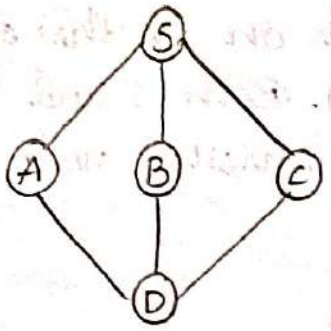
We choose B mark it as visited and push on to the stack here B does not have an unvisited adjacent node. So, we pop B from the stack.

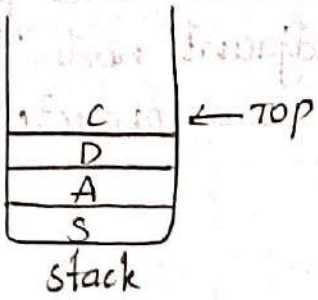
Step-6 :— We chuck the stack top for return to the previous node and check if it has any unvisited nodes. Here we find D to be on the top of the stack.



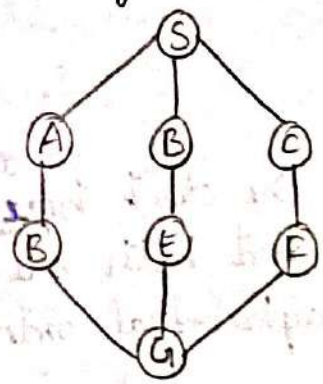


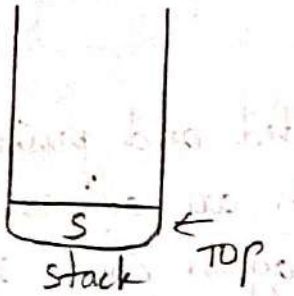
Step 7 :— Only unvisited adjacent node is from D to C. Now, we visit C, mark it as visited and push it on to the stack.

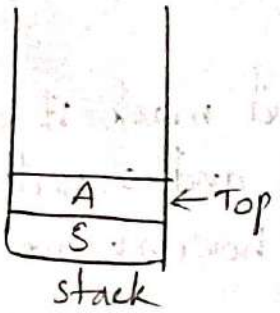


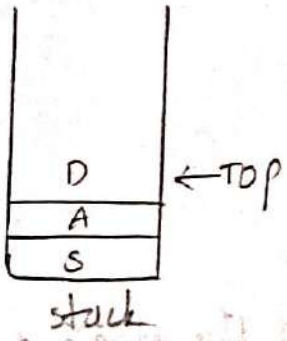


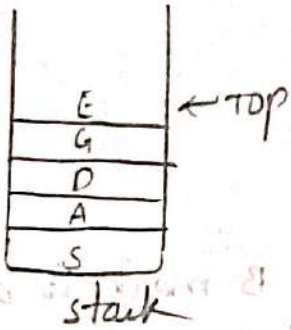
Example-2 :—

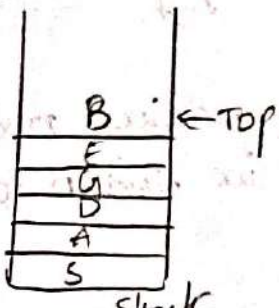


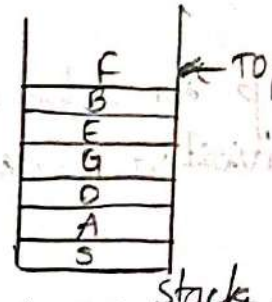


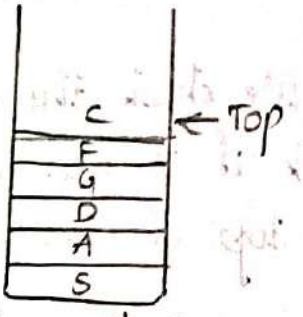












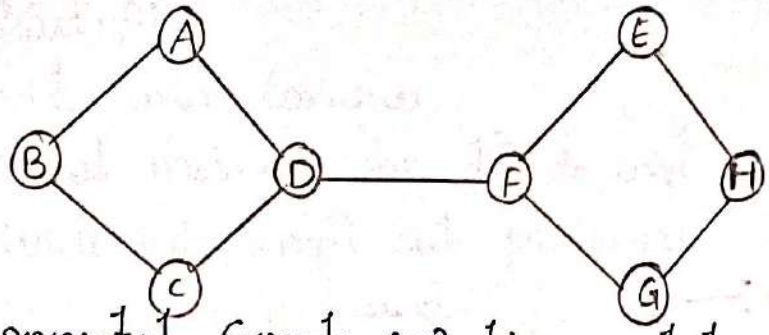
Connected components (or) Connected graph :—

. A Graph in which we can visit from one vertex to any other vertex is called as a connected graph.

. In connected graph atleast one path exist between every pair of vertices i.

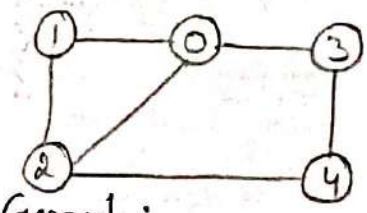
. Here in this graph we can visit from one vertex to any other vertex There exist atleast one path between every pair of vertices.

. Therefore, It is a connected graph.



* Biconnected Graph and biconnected components: An undirected graph is called biconnected. If there are two vertex disjoint paths between any two vertices.
* In a biconnected graph, there is a simple cycle through any two vertices.
* A graph is said to be biconnected if they are
  (i) It is connected that is it is possible to reach every vertex from every other vertex, by a simple path.
  (ii) Even after removing any vertex the graphs remains connected.

Example-1 :—



Applications of Graph:—

1. Social media analysis
2. Network Monitoring
3. Financial Trading
4. Internet of Things (IoT).
5. Autonomous vehicles
6. Disease surveillance.

Advantages of Graphs:—

1. Representing complex data
2. Efficient Data processing
3. Network Analysis.
4. Pathfinding.

5. Visualization
6. Machine learning
7. Webpages
8. Graph Transforming

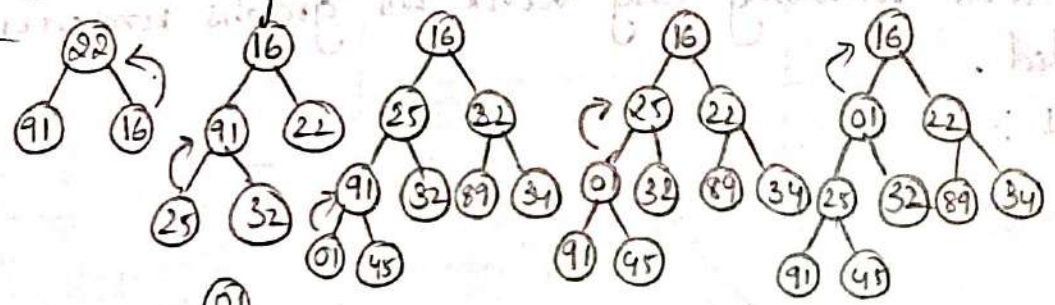
Disadvantages of Graph:—

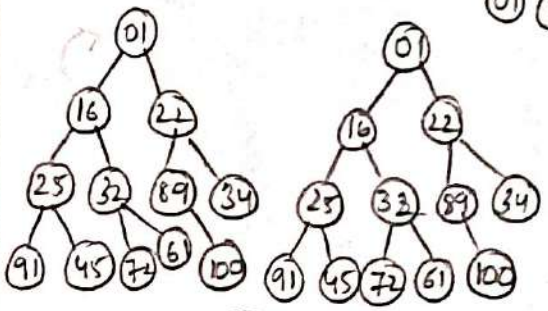
1. Limited Representation
2. Difficulty in Interpretation
3. Scalability issues
4. Data quality issues
5. Lack of standardization
6. Privacy concerns.

Ex-1:— 22 91 16 25 32 89 34 01 45 72 61 100

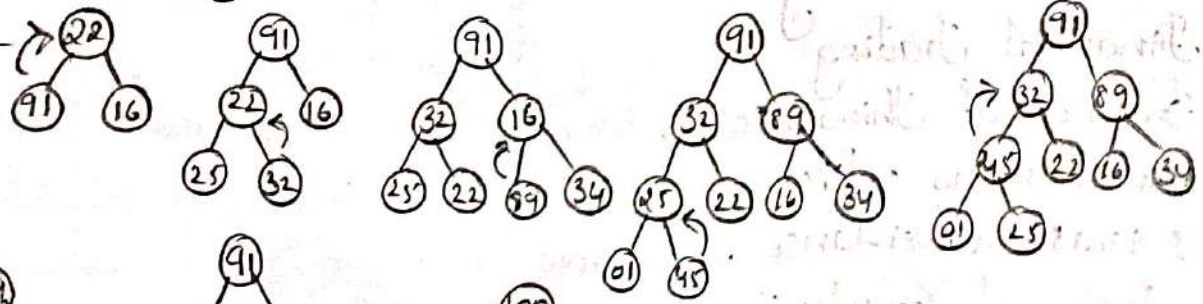
Min and Max heap tree

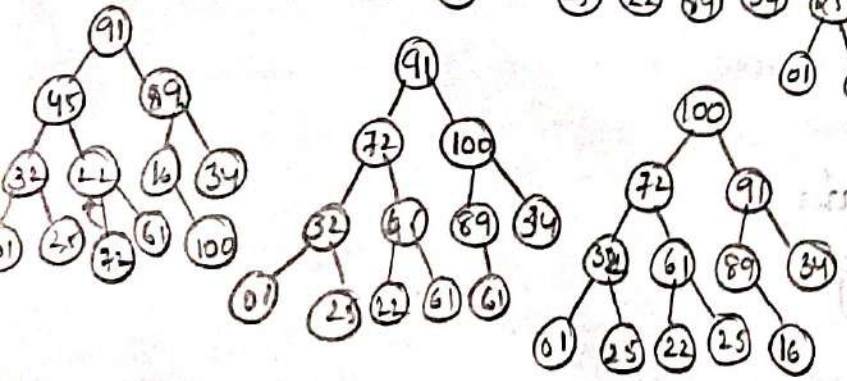
Min:—





Max:—





Part-III

1. Divide and Conquer

2. General method for divide and conquer.

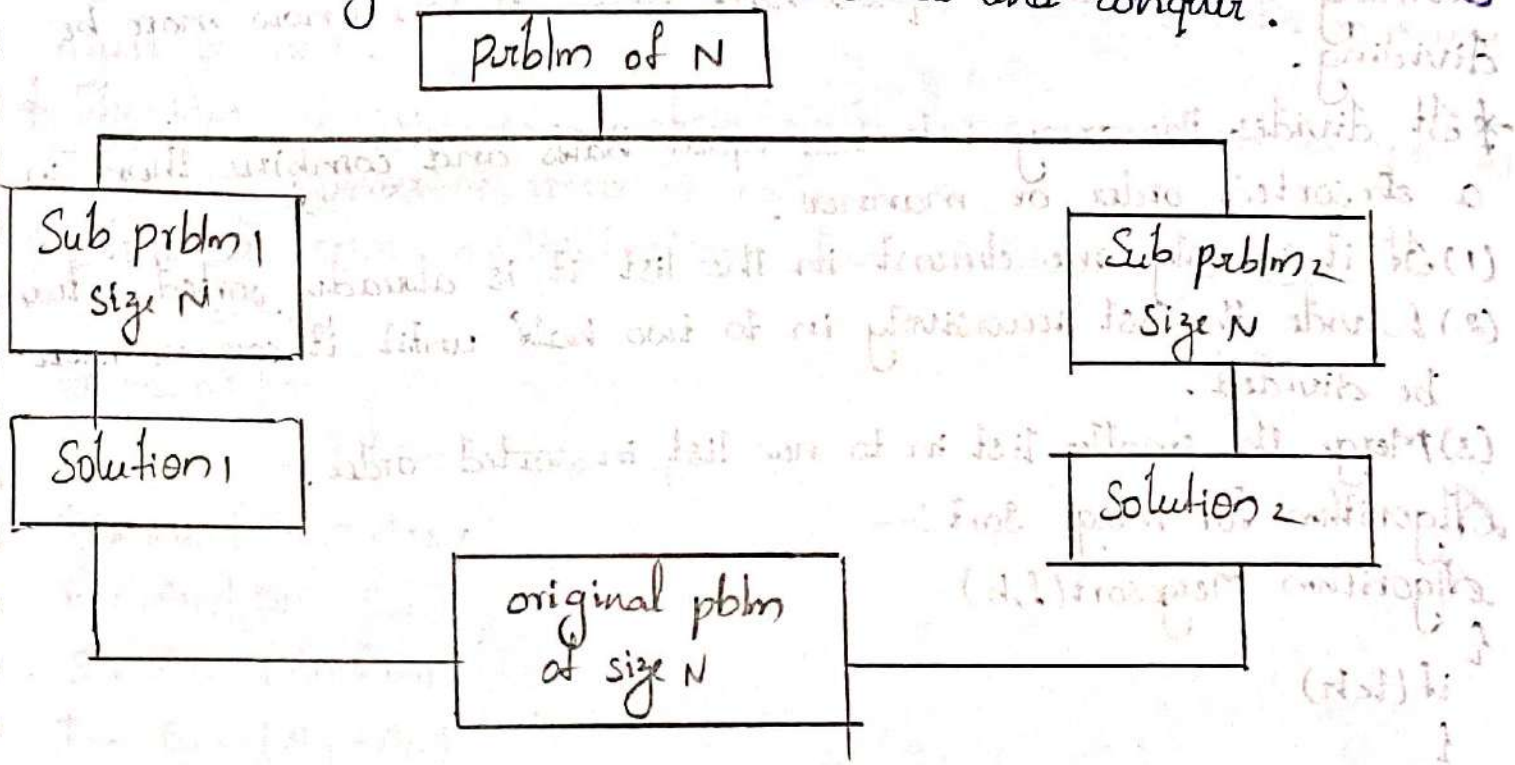
→ Divide into small sub problems.

→ Solve them independently.

→ Combine all solve n (sub problems)

→ If sub problems are large enough then divide and conquer is applied.

→ Recursive algorithm are used in divide and conquer.



→ If a problem some size = N.

→ If a problem is large, divide the problem in to sub problems and solve these sub problems and combine the solution of the sub problem to get the solution for main problem.

Algorithm for divide and conquer :—

```
DA c(p)
{
  if(small(p))
  {
    s(p);
  }
```

```
else
{
divide pin to P1, P2 . . . : Pn
Apply DAC(P1), DAC(P2)
Combine DA(P1), DA(P2)
 }
}
```

Merge Sort :—

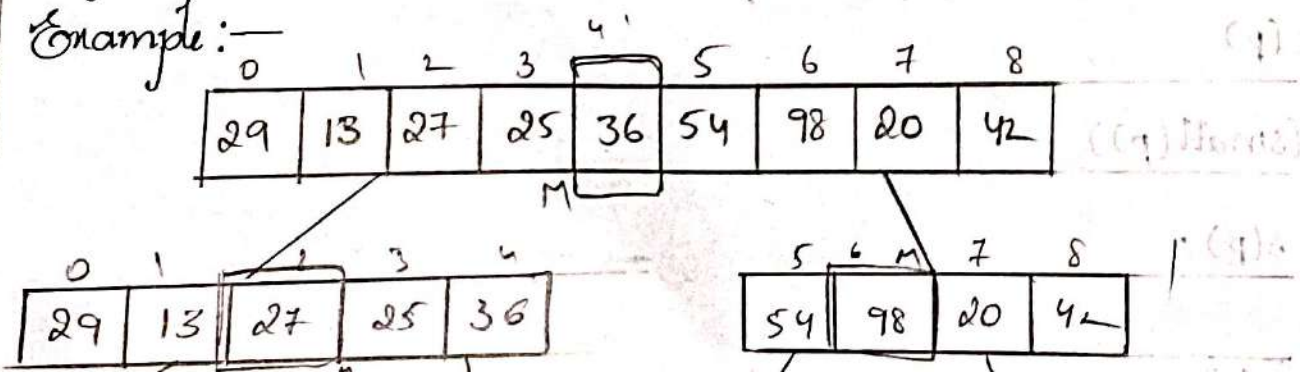
* Merge Sort is a seqpa recursive algorithm that can keep on dividing the list in to equial half until it can know more be dividing.
* It divides the array into two equal halfs and combine them in a sorted order or manner.

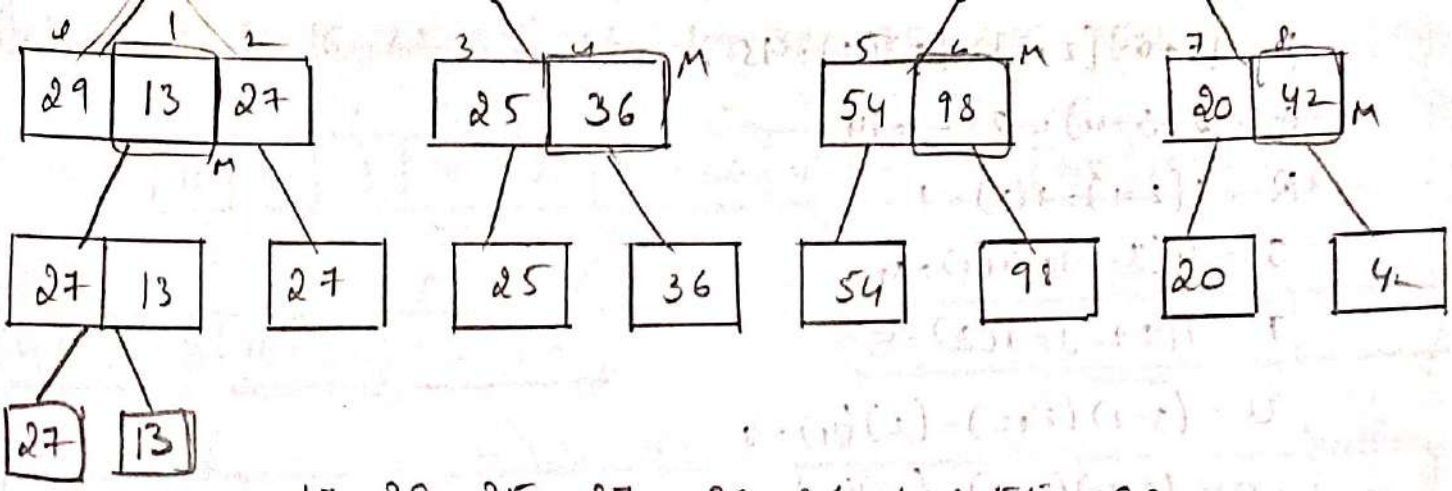
(1) If it is only one element in the list it is already sorted, return.
(2) Divide the list recursively in to two half until it can no more be divided.
(3) Merge the smaller list in to new list in sorted order.

Algorithm for merge Sort :—

```
Algorithm Mergesort(l,h)
{
  if(l<h)
  {
   Mid(l+h)/2;
   MergeSort(l,mid);
   Mergesort(mid+1,h);
   Mergesort(l,mid,h);
  }
}
```

Example :—





13 20 25 27 29 36 42 54 98

Strassen's Matrix Multiplication :—

* It is performed only on square matrixes order of both matrices must be nxn.
* The idea of strassen's matrix multiplication method is used to reduce the number of recursive calls to 7.
* There are seven multiplications, four additions and four subtraction.

Formulas :—

$P = [A_{11} + A_{22}] . [B_{11} + B_{22}]$

$Q = B_{11} . [A_{21} + A_{22}]$

$R = A_{11} . [B_{12} - B_{22}]$

$S = A_{22} . [B_{21} - B_{11}]$

$T = B_{22} . [A_{11} + A_{12}]$

$U = [A_{21} - A_{11}] . [B_{11} + B_{12}]$

$V = [A_{12} - A_{22}] . [B_{21} + B_{22}]$

Ex:— $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $B = \begin{bmatrix} 2 & 2 \\ 3 & 1 \end{bmatrix}$ $C = ?$

$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

$A_{11} = 1$ $B_{11} = 2$ $C_{11} = ?$

$A_{12} = 2$ $B_{12} = 2$ $C_{12} = ?$

$A_{21} = 3$ $B_{21} = 3$ $C_{21} = ?$

$A_{22} = 4$ $B_{22} = 1$ $C_{24} = ?$

$P = [1+4][2+1] = [5][3] = 15$.

$Q = 2[3+4] = 2[7] = 14$

$R = 1[2-1] = 1(1) = 1$

$S = 4[3-2] = 4(1) = 4$

$T = 1[1+2] = 1(3) = 3$

$U = (3-1)(2+2) = (2)(4) = 8$

$V = (2-4)(3+1) = (-2)(4) = -8$

$C_{11} = P+S-T+V \Rightarrow 15+4-3+(-8) \Rightarrow 15+4-3-8 \Rightarrow 19-11 \Rightarrow 8$

$C_{12} = R+S \Rightarrow 1+3 = 4$

$C_{21} = Q+S \Rightarrow 14+4 = 18$

$C_{22} = P+R-Q+U \Rightarrow 15+1-14+8 \Rightarrow 16+8-14 = 10$.

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \Rightarrow \begin{bmatrix} 8 & 4 \\ 18 & 10 \end{bmatrix}.$$

Ex-2 :—

$$A = \begin{bmatrix} 1 & 3 \\ 5 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 4 \\ 3 & 1 \end{bmatrix} \quad C = ?$$

$A_{11} = 1$ $B_{11} = 2$ $P = [1+4][2+1] = [5][3] = 15$

$A_{12} = 3$ $B_{12} = 4$ $Q = 2[5+4] = 2(9) = 18$

$A_{21} = 5$ $B_{21} = 3$ $R = 1[4-1] = 1(3) = 3$

$A_{22} = 4$ $B_{22} = 1$ $S = 4(3-2) = 4(1) = 4$

$T = 1(1+3) = 1(4) = 4$.

$U = [5-1][2+4] = (4)(6) = 24$

$V = [3-4][3+1] = (-1)(4) = -4$

$C_{11} = P+S-T+V = 15+4-4+(-4) \Rightarrow 19-8 = 11$

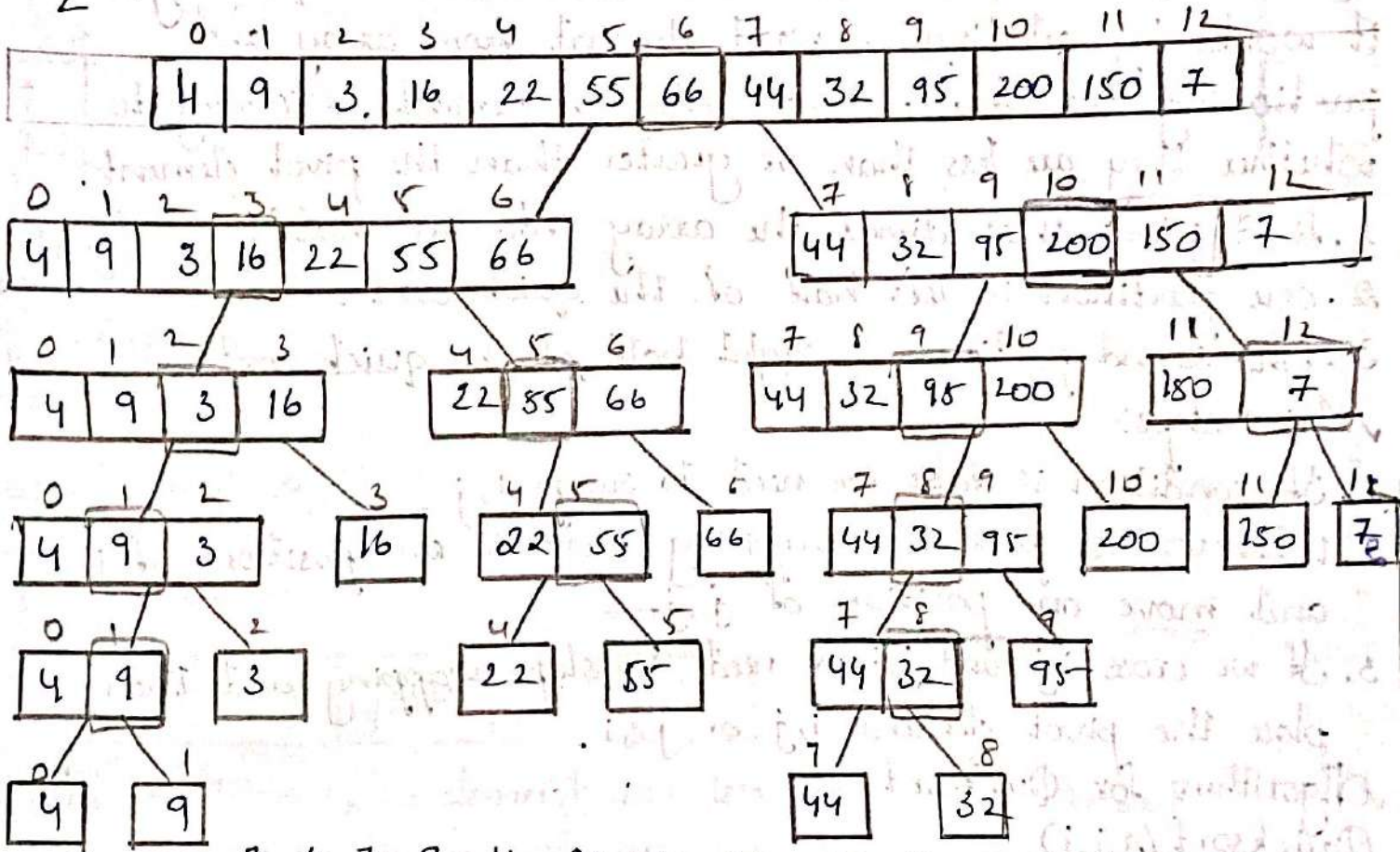
$C_{12} = R+T \Rightarrow 3+4 = 7$

$C_{21} = Q+S \Rightarrow 18+4 = 22$

$C_{22} = P+R-Q+U \Rightarrow 15+3-18+24 \Rightarrow 18-18+24 = 24$

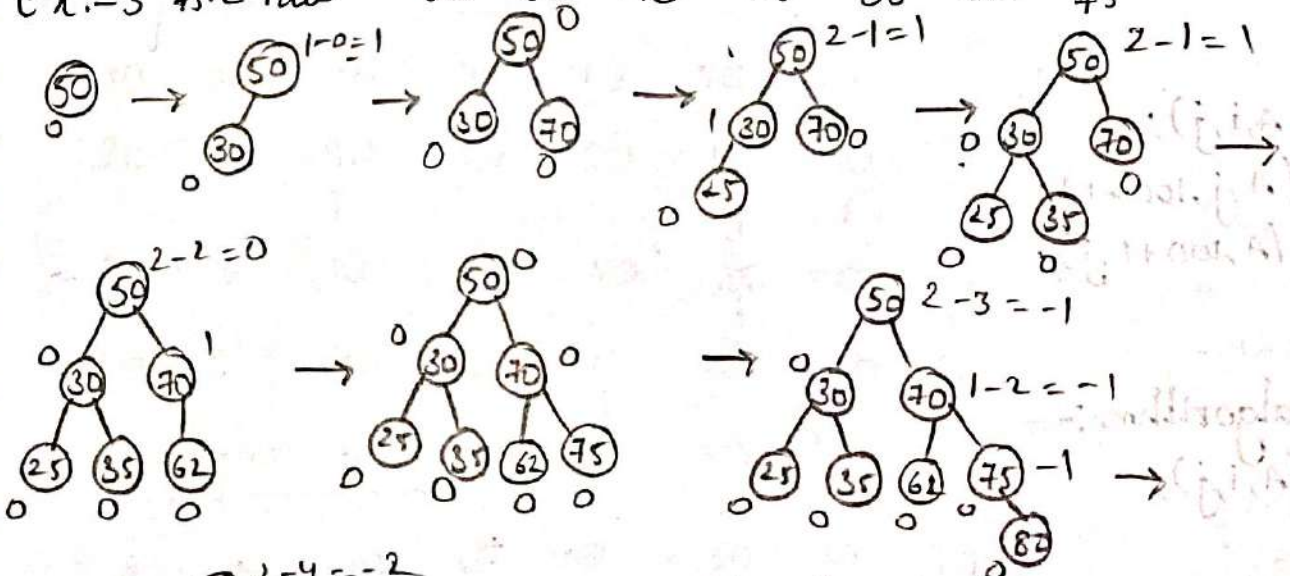
$$C = \begin{bmatrix} 11 & 7 \\ 22 & 24 \end{bmatrix}$$

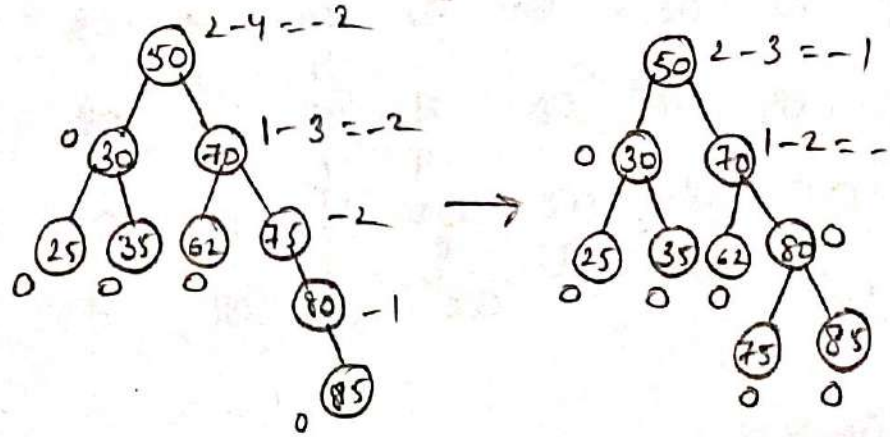
Ex:-2 4 9 3 16 22 55 66 44 32 95 200 150 7 . . .



3 4 7 9 16 22 32 44 55 66 95 150 200.

Ex:-3 AVL Tree 50 30 70 25 35 62 75





Quick Sort :— Quick sort is a divide and conquer algorithm it works by selecting a pivot element from array and partioning the other elements into sub elements. According to whether they are less than or greater than the pivot element.

1. find pivot that divide the array into two half.
2. one partition is left half of the quick sort.
3. The second partition is right half of the quick sort.

Main steps:—

1. If condition is false we need to swap i, j.
2. Whenever we perform swapping move to one position of i++ and move one position of j, j---
3. If we cross i,j and j,i we need to stop swapping and then place the pivot element i,j or j & i.

Algorithm for Quicksort:—

```
Quicksort (A,i,j)
{
if(i<j)
{
partition(A,i,j);
Quicksort(A,j,100-1);
Quicksort(A,100+1,j);
}
}
```

Partition algorithm:—

```
Partition(A,i,j);
{
pivot=a[i];
i=0;
j=n-1;
while(i<j)
{
when(i<j)

while(a[j]<=pivot)
```

```
i++;
}
while(a[j]>pivot)
{
j--;
}
if(i<j)
{
swap(a[i],a[j]);
}
}
swap(a[pivot],a[j]);
return j;
}
```

partition 1	pivot	partition 2

1. In partition 1, the elements are less than pivot element.
2. In partion 2, the elements are greater than pivot element.

Example :—

10 80 30 90 40 50 60

20 5 90 70 50 10 30 — 20<30 T

i j p

20 5 10 70 50 15 90 30 — 5<30 T

j — 10<15 T

20 5 10 15 50 70 90 30 — 90<30

ij — 70<30 T

20 5 10 15 30 50 70 90

i p j — i<p<j

20 5 10 15 30 50 70 90

i j p

20 5 20 15 30 50 70 90

i j p

10 5 15 20 50 70 90

i p j

5 10 15 20 50 70 90

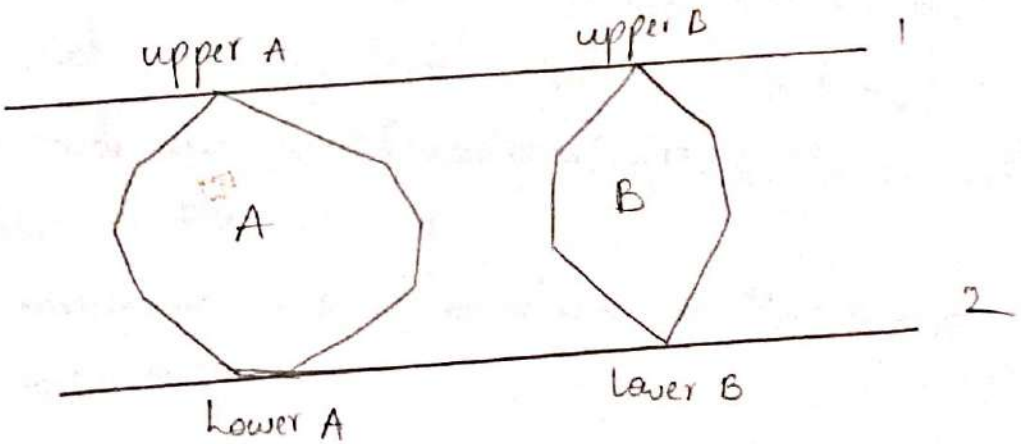
Convex hull:— The convex hull of a set of points in a Euclidean space is the smallest convex polygon that encloses all of the points. In two dimensions(2D), the convex hull is a convex polygon, and in three dimensions(3D), the convex hull is a convex polyhedron.

Convex hull using divide and Conquer algorithm:—

Algorithm: Given the set of points for which we have to find the convex hull. Suppose we know the convex hull of the left half points and the right half points, then the problem now is to merge these two convex hulls and determine the convex hull for the complete set.

* This can be done by finding the upper and lower tangent to the right and left convex hulls. This is illustrated here tangents between two convex polygons.

Let the left convex hull be a and the right convex hull be b. Then the lower and upper tangents are named as 1 and 2 respectively, as shown in the figure. Then the red outline shows the final convex hull.



* Now the problem remains, how to find the conver hull for the left and right half. Now recursion comes into the picture, we divide the set of points until the number of points in the set is very small, say 5, and we can find the conver hull for these points by the brute algorithm.
* The merging of these halves would result in the conver hull for the complete set of points.
* But the conver hull for 3 points is the complete set of points. This is correct but the problem comes when we try to merge a left convex hull of 2 points and right convex hull of 3 points, then the program gets trapped in an infinite loop in some special cases.
* So, to get rid of this problem I directly found the conver hull for 5 points by $O(N^3)$ algorithm, which is somewhat greater but does not affect the overall complexity of the algorithm.

Chapter-03

GREEDY METHOD

Greedy method :-

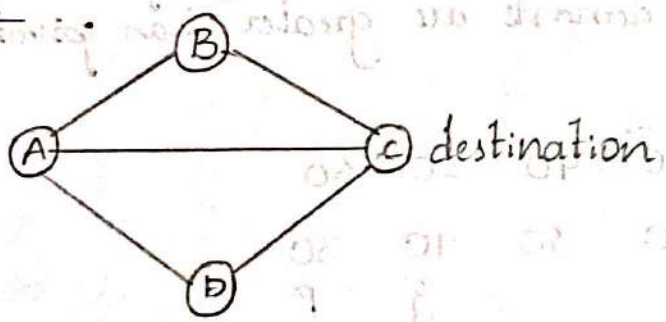
* It is used for solving optimisation problems.
* Optimisation means the problem which requires min and max results.

General method for greedy :-

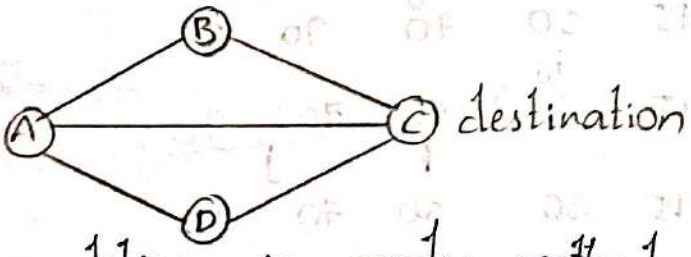
When compare to all algorithm approaches, the simple and straight forward approach is greedy method.

* Greedy method is a technique in which the decision is taken on the basis of current available information worrying about the effect current decision in future. So there is no guarantee that we will get best solution.

Example 1 :-



Example 2 :-



There are two conditions in greedy method

1. Feasible solution
2. Optimal solution.

Feasible solution :- The solution which satisfy our given condition is known as feasible solution.

Optimal solution :- The solution which is feasible solution gives best solution.

Components :—

There are five components in greedy method.

1. Candidate set.
2. Selection function.
3. Objective function.
4. Solution function.
5. Feasibility function.

1. Candidate Set :— This solution is created from this set.
2. Selection function :— This is used to choose best candidate to be added to solution.
3. Objective function :— This is used to assign a value to the solution.
4. Solution function :— It is used to indicate whether a complete solution has been reached.
5. Feasibility function :— It is used to determine whether a candidate can be used to contribute to the solution.

Algorithm for general method :—

```
Algorithm Greedy(a, n)
{
for i=1 to n do → in an array we can include n number of
                                                elements.
{
x = selection ; → Select each element in an array.
if feasible(x) then → If you find any feasible solution.
{
Solution = solution + x ; → we need to include in our solution.
}
}
}
```

Job Sequencing with deadlines:—

* Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline.
* It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is one.
* Maximize the total profit if only one job can be scheduled at a time.

Algorithm for job sequencing with deadlines:—

Step-1:— Find the maximum deadline value from the input set of jobs.

Step-2:— Once the deadline is decided, arrange the jobs in descending order of their profits.

Step-3:— Select the jobs with highest profit, there time periods not exceeding the maximum deadline.

Step-4:— The selected set of jobs are the output.

Example:—

Jobs	J_1	J_2	J_3	J_4	J_5
deadline	2	2	1	3	4
profit	20	60	40	100	80

Step-1:— Find the maximum deadline, from the given deadlines.

Step-2:— Arrange the jobs in descending order of their profits.

(i) The maximum deadline is 4 Therefore all the last must end before 4.

(ii) Choose the job with highest profit J_4, It takes up 3 hours/parts/periods/time of the maximum deadline.

(iii) Therefore the next job must have the time.

(iv) Total profit = 100.

Step-3:— The next job with highest profit is J_5 but time taken J_5 is 4, which exceeds the deadline by 3. Therefore, it cannot added to the output set.

Step-4 :— The next job with highest profit is J_2, but time taken by 2, which exceeds the deadline by 3, Therefore it cannot be added to the output set.

Step-5 :— The next job with highest profit is J_3, but the time taken by 1, which exceeds the deadline by 3, Therefore it can be added to the output set.

Total profit = 100 + 40 = 140

Step-6 :— Since, the maximum deadline is met, the algorithm comes to an end. The output set of jobs scheduled with the deadlines are J_4, J_3 with the maximum profit of 140.

Example-2 :—

Jobs	J_1	J_2	J_3	J_4	J_5
profit	30	40	10	5	5
deadline	1	4	1	1	1

$n = 6$

Jobs	J_2	J_1	J_3	J_4	J_5
profit	40	30	10	5	5
deadline	4	1	1	1	1

slot 1	slot 2	slot 3
J_1	J_2	J_3

40 + 30 + 10 = 80

Knapsack Problem :—

* Given the weights and profits of n items in the form of profit, weight put these items in a knapsack of capacity W to get the maximum total profit in the knapsack.
* In fractional knapsack, we can break items for maximizing the total value of the knapsack.

Algorithm for knapsack :—

Step-1 :— Consider all the items with their weight and profit mentioned respectively.

Step-2:— Calculate p/w of all the items and sort the items in decreasing order based on their p/v values.

Step-3:— Without exceeding the limit, add the items into the knapsack.

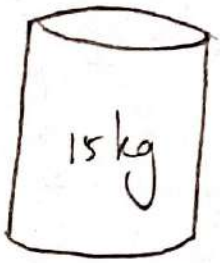
Step-4:— In the knapsack can still store items and some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.

Step-5:— Hence, giving it the name fractional knapsack problem.

Example-1:— $n=7$, $M=15$ kg

Objects	1	2	3	4	5	6	7
Profit	10	5	15	7	6	18	3
weight	2	3	5	7	1	4	1
p/w	5	1.6	3	1	6	4.5	3

$$x_i = (1 \quad 2/3 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1)$$
$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7$$



$15 - 1 = 14$
$14 - 2 = 12$
$12 - 4 = 8$
$8 - 5 = 3$
$3 - 1 = 2$
$2 - 2 = 0$

$$\Sigma_i x_i w_i = 2\times1 + 3\times\frac{2}{3} + 5\times1 + 7\times0 + 1\times6 + 4\times1 + 1\times1$$
$$= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$$

$$\Sigma_i x_i p_i = 10\times1 + 5\times\frac{2}{3} + 15\times1 + 7\times0 + 6\times1 + 18\times1 + 3\times1$$
$$= 10 + \frac{10}{3} + 15 + 0 + 6 + 18 + 3$$
$$= 13.33 + 15 + 6 + 18 + 3$$
$$= 54.6$$

Example-2

$n=5$, $m=10$ kg

Object	1	2	3	4	5
Weight	3	3	2	5	1
Profit	10	15	10	20	8
p/w	3.33	5	5	4	8

$10 - 1 = 9$

$9 - 3 = 6$

$6 - 2 = 4$

$4 - 4 = 0$

$\frac{4}{5} \times 5 = 4$

$x_1 = (0 \quad 1 \quad 1 \quad 4/5 \quad 1)$

$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5$

$\Sigma_i x_i w_i = 0 \times 3 + 1 \times 3 + 1 \times 2 + \frac{4}{5} \times 5 + 1 \times 1$

$= 0 + 3 + 2 + 4 + 1 = 10$

$\Sigma_i x_i p_i = 0 \times 10 + 1 \times 15 + 1 \times 10 + \frac{4}{5} \times 20 + 1 \times 8$

$= 0 + 15 + 10 + 16 + 8 = 49$

Step 1:— Given $n=5$ and calculate p/w for all the items.

Step 2:— Arrange all the items in decreasing order based on p/w.

Step 3:— Without exceeding the knapsack capacity, insert the items in the knapsack with maximum profit.

→ However, the knapsack can still hold 4kg weight, but the next item having 5kg weight will exceed the capacity.

→ Therefore only 4kg weight of the 5kg will be added in the knapsack.

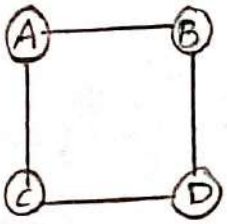
→ Hence the knapsack hold the weight 10, with maximum profit 49.

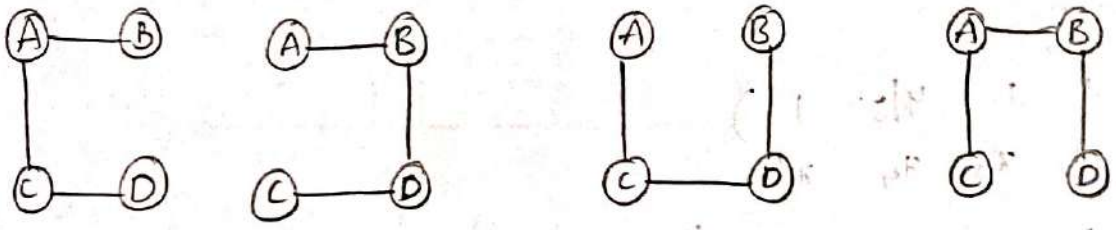
Minimum Cost Spanning Tree:—

What is Spanning tree:— A spanning tree is a subgraph of the given graph.

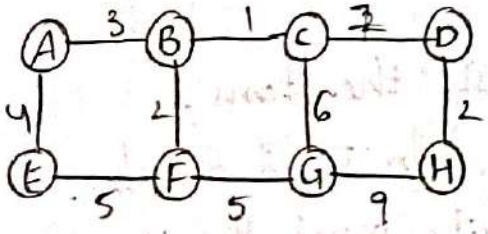
* Spanning tree satisfies three properties:—
  1. If graph contain all the vertices in a graph.
  2. If graph contain n vertices then the spanning tree should contain (n-1) edges.
  3. Spanning tree should not contain any cycle.

Example:—

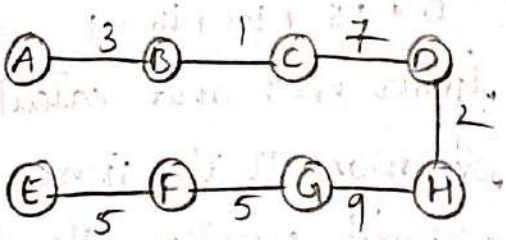




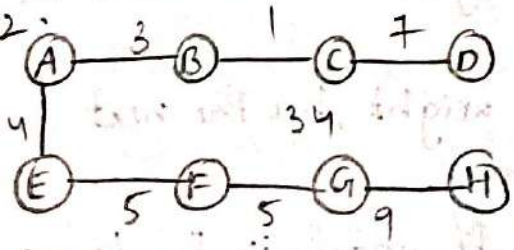
What is Minimum Cost Spanning tree:— A spanning tree in which the cost is minimum than the remaining spanning tree then it is called MCST.



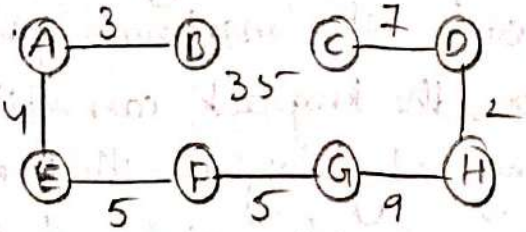
1.

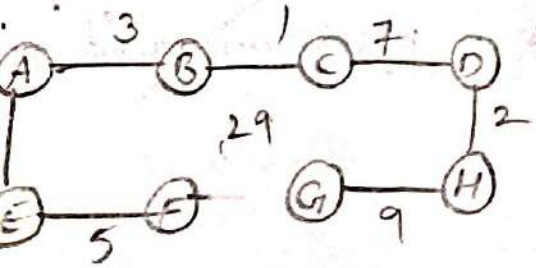


2.

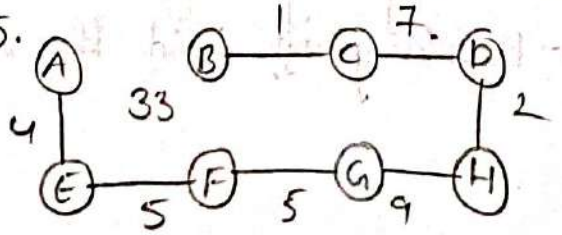


3.

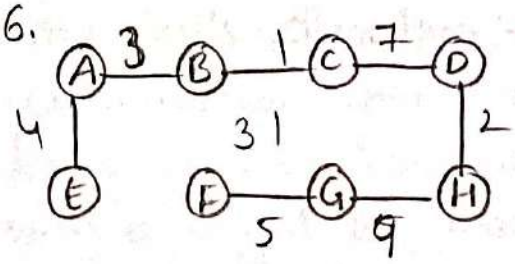




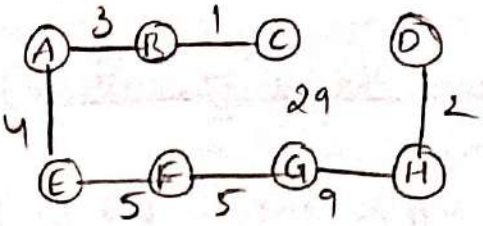
5.



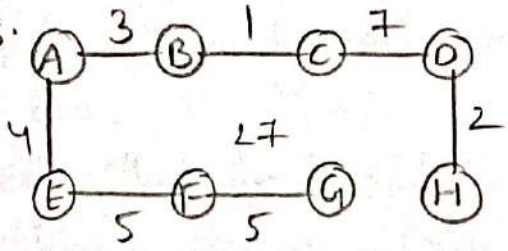
6.



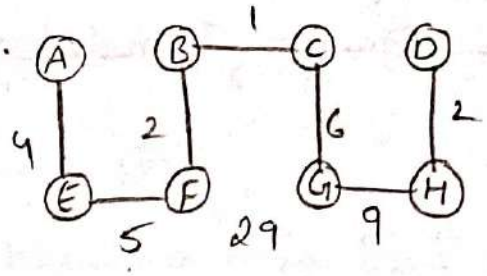
7.



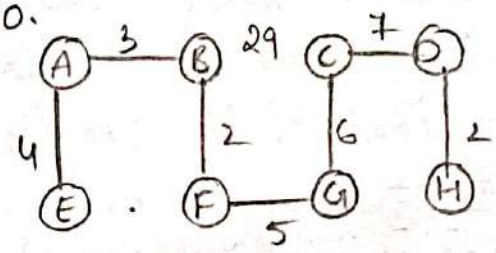
8.



9.



10.



In minimum cost spanning tree, there are two algorithms.

1. Prims algorithm
2. Krushkal's algorithm.

Prim's algorithm :- It is mainly useful to calculate the minimum cost spanning tree.

Algorithm for prims :-

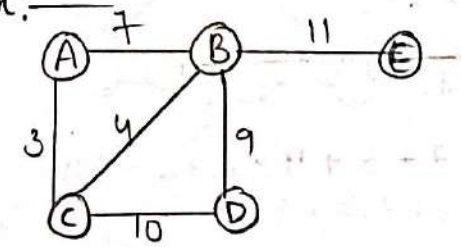
Step 1 :- Start with any vertex of a graph.

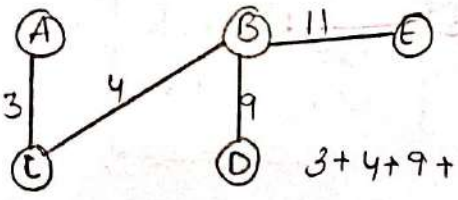
Step 2 :- Find the edges associated with that vertex and add minimum edge spanning tree if it is not from any cycle.

Step 3 :- If it form a cycle by adding that edge we can delete that edge.

Step 4 :- we have to repeat step 2 and 3 till all vertices are covered.

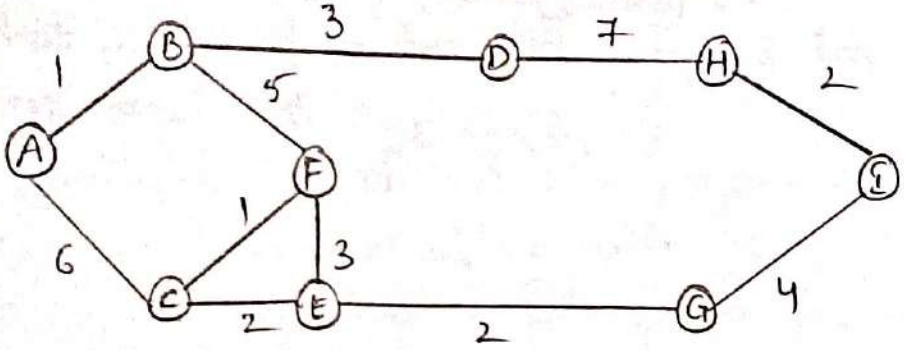
Ex :-

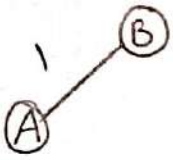


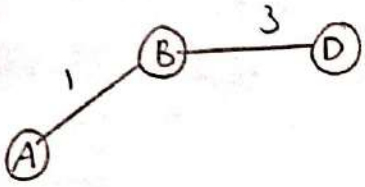


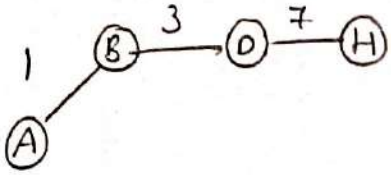
3+4+9+11=27

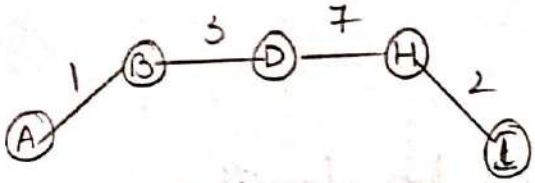
Example:—

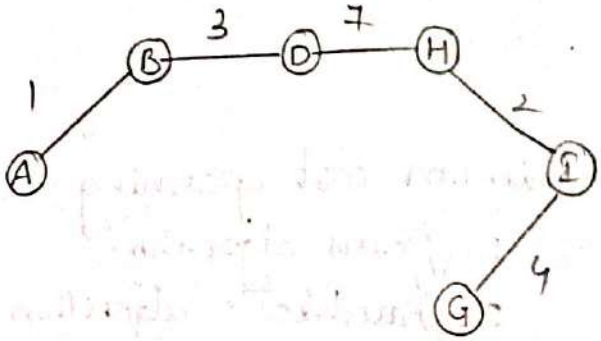


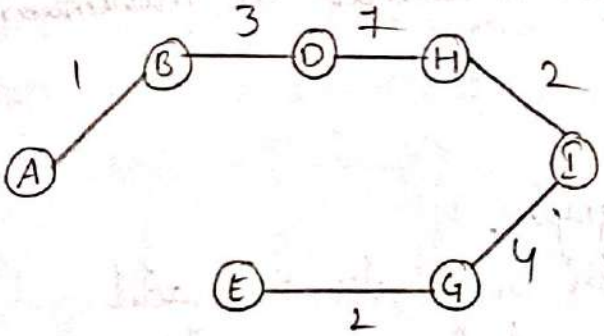


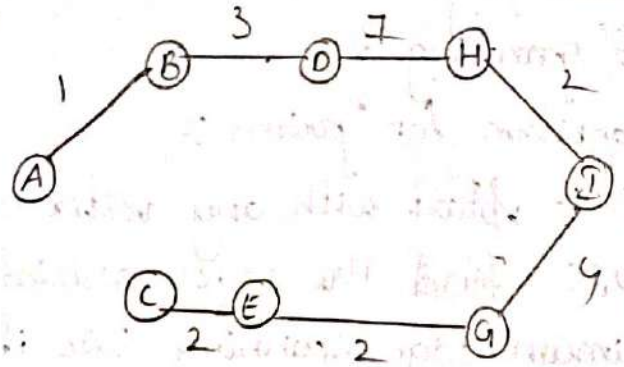


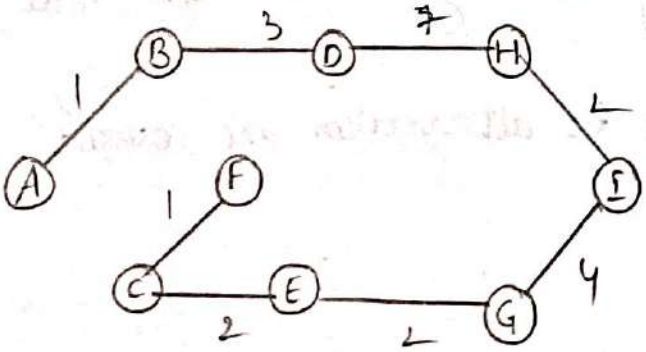


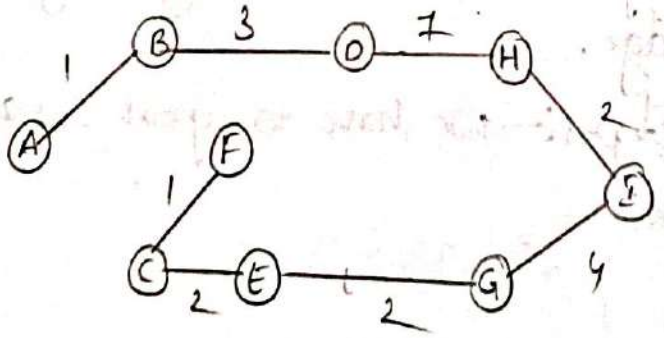












1+3+7+3+4+2+2 = 23

Krushkal's Algorithm :—

A minimum spanning tree or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every otha spanning tree.

* It is mainly useful inorder to calculate the minimum cost spanning tree.

Algorithm for krushkal's :—

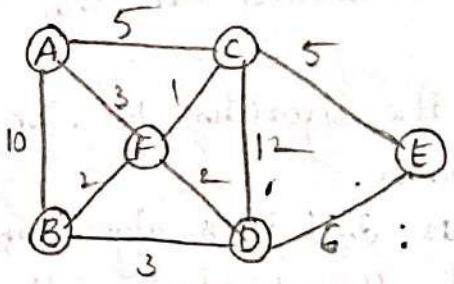
Step 1 :— Arrange all the edges in increasing order based upon the cost.

Step 2 :— Select minimum cost yet from the edge list and add the edge to the spanning tree if it not forming any cycle.

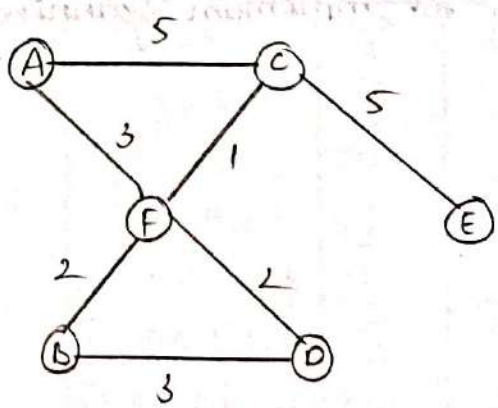
Step 3 :— If it form a cycle by adding that edge we can remove or delete.

Step 4 :— We have to repeat step 2 and 3 until all the vertices are covered.

Example :—

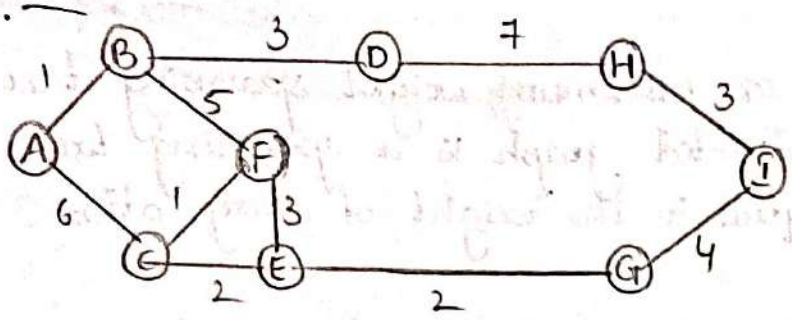


Edge	Cost
C - F ✓	1
B - F ✓	2
F - D ✓	2
A - F ✓	3
B - D ✗	3
A - C ✗	5
C - E ✓	5
D - E ✗	6
A - B ✗	10
C - D ✗	12

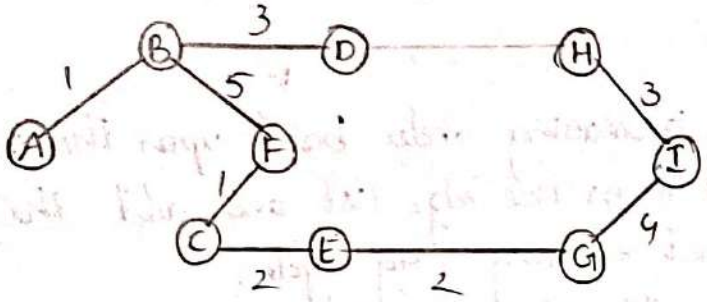


$5 + 3 + 2 + 1 + 2 = 13$

c:—



Edge	Cost
A–B ✓	1
C–F ✓	1
C–E ✓	2
C–G ✓	2
B–D ✓	3
E–F ✗	3
I–H ✓	3
I–G ✓	4
B–F ✓	5
A–C ✓	6
D–H ✗	7



1+3+5+1+2+2+4+3 = 21

Single Source Shortest Path:—

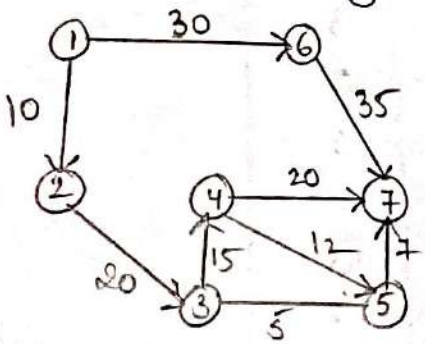
We have a graph G=(V,E), graph is a directed weighted graph.

* we have to select one vertex as source vertex and we need to find out the shortest path from source vertex to remaining all the vertices of a graph.

* In this, single source shortest path we use the algorithm to solve the shortest path in the single source shortest path.

* Here we are using the algorithm named as: dijkstra's algorithm.

Dijkstra's algorithm:— It is used to identify the shortest path from starting node to destination node in a weighted graph. This is quite similar to prims algorithm for minimum spanning tree and it works on a greedy method.

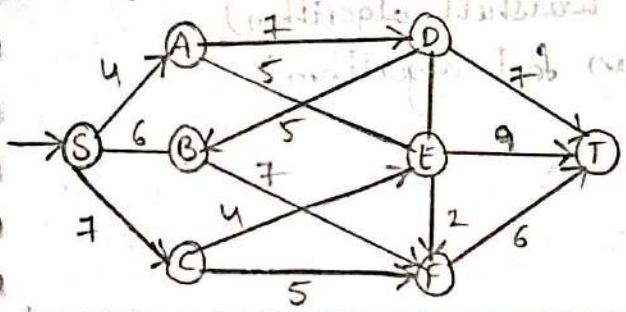


Set is a collection of elements which are enclosed in curly braces.

Selected vertex	Visited set	d(2)	d(3)	d(4)	d(5)	d(6)	d(7)
1	{1}	(10)	∞	∞	∞	30	∞
2	{1,2}	(10)	(30)	∞	∞	30	∞
3	{1,2,3}	(10)	(30)	45	35	(30)	∞
6	{1,2,3,6}	(10)	(30)	45	(35)	(30)	65
5	{1,2,3,6,5}	(10)	(30)	45	(35)	(30)	(42)
7	{1,2,3,6,5,7}	(10)	(30)	45	(35)	(30)	(42)
4	{1,2,3,6,5,7,4}	(10)	(30)	(45)	(35)	(30)	(42)

1	Cost
2	10
3	30
6	45
5	35
7	30
4	42

Example-2:—



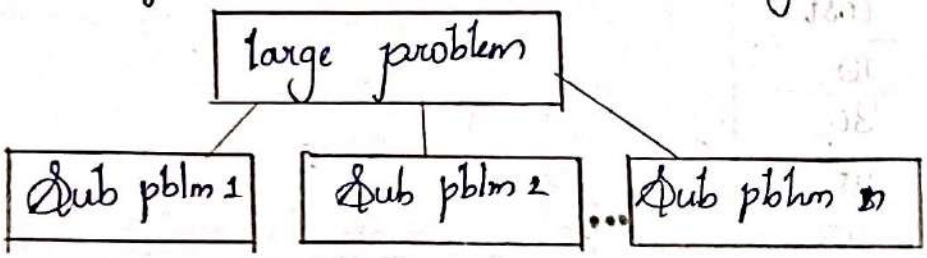
Source(s)	Cost
A	4
B	16
C	7
D	14
E	11
F	13
T	19

Selected vertex	Visited set	d(A)	d(B)	d(C)	d(D)	d(E)	d(F)	d(T)
S	{S}	(4)	6	7	∞	∞	∞	∞
A	{S,A}	(4)	6	7	11	9	∞	∞
B	{S,A,B}	(4)	(6)	(7)	11	9	13	∞
C	{S,A,B,C}	(4)	(6)	(7)	(11)	11	12	∞
D	{S,A,B,C,D}	(4)	(6)	(7)	(11)	(11)	12	18
E	{S,A,B,C,D,E}	(4)	(6)	(7)	(11)	(11)	(13)	20
F	{S,A,B,C,D,E,F}	(4)	(6)	(7)	(11)	(11)	(13)	19
T	{S,A,B,C,D,E,F,T}	(4)	(6)	(7)	(11)	(11)	(13)	(19)

Part-II

Dynamic Programming

* Dynamic programming solve problem by combining the solution to sub-problem. It is mainly used for optimization problem.
* Each solution contain value, the following are the steps that the dynamic programming follows
  (i) It breaks the complex problem into sample sub problems.
  (ii) It find optimal solution to these sub problems.
  (iii) It stores the result of sub problems in memory. The process of storing the sub problems is known as memorization.



Example :— 1 + 1 + 2 + 3 + 5 + 5 —

Applications of dynamic programming :—

1. All pairs shortest path (Floyd Warshell algorithm)
2. Single Source Shortest path (Bellman ford algorithm)
3. Optimal binary search
4. 0/1 knapsack
5. Travelling sales person.

All pairs shortest path :—

* Floyd Warshell algorithm is a dynamic programming algorithm which is used to compute the all pair shortest path in directed or undirected graph and shortest paths in a directed or undi where V is the no. of vertices in a graph.
* The all pair shortest path algorithm is also known as floyd warshell algorithm is used to find all pair shortest path problem from a given weighted graph.
* As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to another other nodes in the graph.

Algorithm for all pairs shortest path:—

Step-1:—Remove all the self loops and parallel edges from the graph, in the given graph there are neither self edges nor parallel edges.

Step-2:—

(i) Write the initial distance matrix.

(ii) It represent the distance between every pair of edges vertices in the form of given weight.

(iii) For diagonal elements, distance value = 0.

(iv) For vertices having a direct edge between them, distance value = weight of that edge.

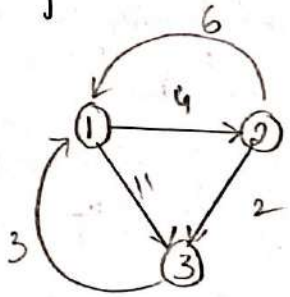
(v) For vertices having no direct edge between them, distance value = ∞.

Step-3:—Using floyd warshall algorithm, write the following four matrices.

Formula:—

$$A^*[i,j] = \min\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$$

Example:—

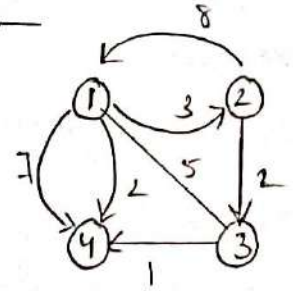


$$A^0 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$A^k[i,j] = \min\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$$

$$A^1 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Ex:—



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & ? & \infty \\ 5 & \infty & 0 & ? \\ 2 & \infty & \infty & 0 \end{bmatrix} \Rightarrow A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix}$$

$A^1[2,3] = \min\{A^0[2,3], A^0[2,1] + A^0[1,3]\}$

$= \min\{2, 8+\infty\} = \min\{2\}$

$A^1[2,4] = \min\{A^0[2,4], A^0[2,1] + A^0[1,4]\}$

$= \min\{\infty, 8+7\}$

$= \min\{\infty, 15\} = \min\{15\}$

$$A_1 = \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

$A^1[3,2] = \min\{A^0[3,2], A^0[3,1] + A^0[1,2]\}$

$= \min\{\infty, 5+3\} =$

$= \min\{\infty, 8\} = \min\{8\}$

$A^1[3,4] = \min\{A^0[3,4], A^0[3,1] + A^0[1,4]\}$

$= \min\{1, 5+7\}$

$= \min\{1, 12\} = \min\{2\}$

$$A_3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

$$A_4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

$A^1[4,2] = \min\{A^0[4,2], A^0[4,1] + A^0[1,2]\}$

$= \min\{\infty, 2+3\}$

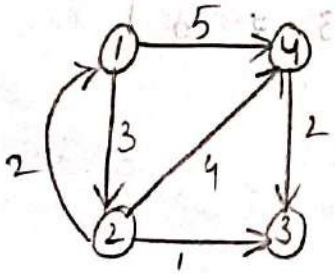
$= \min\{\infty, 5\} = \min\{5\}$

$A^1[4,3] = \min\{A^0[4,3], A^0[4,1] + A^0[1,3]\}$

$= \min\{\infty, 2+\infty\} = \min\{\infty\}$

$$\therefore A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 1 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 1 \end{bmatrix}, A_3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}, A_4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 2 \\ 3 & 8 & 0 & 1 \\ 0 & 5 & 7 & 0 \end{bmatrix}$$



$$A^0 = \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 1 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

$A^k[i,j] = \min\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & ? & ? \\ \infty & ? & 0 & ? \\ \infty & ? & ? & 0 \end{bmatrix}$$

$A^1[2,3] = \min\{A^0[2,3], A^0[2,1] + A^0[1,3]\}$

$= \min\{1, 2+\infty\} = \min\{1\}$

$A^1[2,4] = \min\{A^0[2,4], A^0[2,1] + A^0[1,4]\}$

$= \min\{4, 2+5\}$

$= \min\{4, 7\} = \min\{4\}$

$A^1[3,2] = \min\{A^0[3,2], A^0[3,1] + A^0[1,2]\}$

$= \min\{\infty, \infty+3\}$

$= \min\{\infty\}$

$A^1[3,4] = \min\{A^0[3,4], A^0[3,1] + A^0[1,4]\}$

$= \min\{\infty, \infty+5\}$

$= \min\{\infty\}$

$A^1[4,2] = \min\{A^0[4,1], A^0[4,1] + A^0[2,2]\}$

$= \min\{\infty, \infty+3\} = \min\{\infty\}$

$A^1[4,3] = \min\{A^0[4,3], A^0[4,1] + A^0[1,3]\}$

$= \min\{2, \infty+\infty\} = \min\{2\}$

$$\therefore A^1 = \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 1 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 0 & 3 & ? & ? \\ 2 & 0 & 1 & 4 \\ ? & \infty & 0 & ? \\ 0 & \infty & 0 & 0 \end{bmatrix}$$

$A^2[1,3] = \min\{A^1[1,3], A^1[1,2] + A^1[2,3]\}$

$= \min\{\infty, 3+1\}$

$= \min\{4\}$.

$A^1[1,4] = \min\{A^1[1,4], A^1[1,2] + A^1[2,4]\}$

$= \min\{5, 3+4\}$

$= \min\{5\}$

$A^2[3,1] = \min\{A'[3,1], A'[3,2] + A'[2,1]\}$

$= \min\{\infty, \infty + 2\} = \min\{\infty\}$

$A^2[3,4] = \min\{A'[3,4], A'[3,2] + A'[2,4]\}$

$= \min\{\infty, \infty + 4\}$

$= \min\{\infty\}$

$A^2[4,1] = \min\{A'[4,1], A'[4,2] + A'[2,1]\}$

$= \min\{\infty, \infty + 1\}$

$= \min\{\infty\}$

$A^2[4,3] = \min\{A'[4,3], A'[4,2] + A'[2,3]\}$

$= \min\{2, \infty + 1\}$

$= \min\{2\}$.

$\therefore A^2 = \begin{bmatrix} 0 & 3 & 4 & 5 \\ 2 & 0 & 1 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$

$\therefore A^3 = \begin{bmatrix} 0 & ? & 4 & ? \\ ? & 0 & 1 & ? \\ \infty & \infty & 0 & \infty \\ ? & ? & 2 & 0 \end{bmatrix}$

$A_3[1,2] = \min\{A^2[1,2], A^2[1,3] + A^2[3,2]\}$

$= \min\{3, 4 + \infty\}$

$= \min\{3\}$

$A_3[1,4] = \min\{A^2[1,4], A^2[1,3] + A^2[3,4]\}$

$= \min\{5, 4 + 4\} = \min\{5\}$

$A_3[2,1] = \min\{A^2[2,1], A^2[2,3] + A_2[3,1]\}$

$= \min\{2, 1 + \infty\} = \min\{2\}$

$A_3[2,4] = \min\{A^2[2,4], A^2[2,3] + A^2[3,4]\}$

$= \min\{4, 1 + \infty\} = \min\{4\}$

$A_3[4,1] = \min\{A'[4,1], A^2[4,3] + A^2[3,1]\}$

$= \min\{\infty, 2 + \infty\} = \min\{\infty\}$

$A_3[4,2] = \min\{A^2[4,2], A^2[4,3] + A^2[3,2]\}$

$= \min\{\infty, 2 + \infty\} = \min\{\infty\}$

$\therefore A^3 = \begin{pmatrix} 0 & 3 & 4 & 5 \\ 2 & 0 & 1 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 0 \end{pmatrix}$

$$A_4 = \begin{bmatrix} 0 & ? & ? & 5 \\ ? & 0 & ? & 4 \\ ? & ? & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

$A_4[1,2] = \min\{A^3[1,2], A^3[1,4] + A^3[4,2]\}$

$= \min\{3, 5+\infty\} = \min\{3\}$

$A_4[1,3] = \min\{A^3[1,3], A^3[1,4] + A^3[4,3]\}$

$= \min\{4, 5+2\} = \min\{4\}$

$A_4[2,1] = \min\{A^3[2,1], A^3[2,4] + A^3[4,1]\}$

$= \min\{2, 4+\infty\} = \min\{2\}$

$A_4[2,3] = \min\{A^3[2,3], A^3[2,4] + A^3[4,3]\}$

$= \min\{1, 4+2\} = \min\{1\}$

$A_4[3,1] = \min\{A^3[3,1], A^3[3,4] + A^3[4,1]\}$

$= \min\{\infty, \infty+\infty\}$

$= \min\{\infty\}$

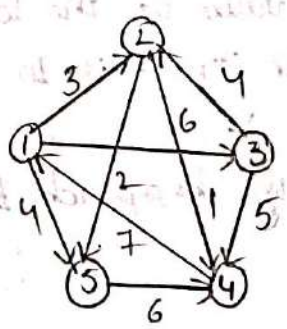
$A[3,2] = \min\{A^3[3,2], A^3[3,4] + A^3[4,2]\}$

$= \min\{\infty, \infty+\infty\}$

$= \min\{\infty\}$

$$\therefore A^4 = \begin{bmatrix} 0 & 3 & 4 & 5 \\ 2 & 0 & 1 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

Ex:—



$$A^0 = \begin{bmatrix} 0 & 3 & 6 & \infty & 4 \\ \infty & 0 & \infty & 1 & 2 \\ \infty & 4 & 0 & 5 & \infty \\ 7 & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Optimal Binary Search Tree:—

The optimal Binary Search Tree problem is a classic problem in computer science. It involves finding the optimal binary search tree for a given set of keys and their frequencies.

Problem Definition:—

Given

* A set of keys $(k_1, k_2, \ldots, k_n)$
* A set of frequencies $(f_1, f_2, \ldots, f_n)$ for each key.
* A set of internal nodes costs $(c_1, c_2, \ldots, c_n)$

Dynamic programming solution:—

The optimal binary search tree problem can be solved by using dynamic programming. The idea is to break down the problem into smaller sub-problems and solve each sub-problem only once. Here's a step-by-step explanation of the dynamic programming solution:

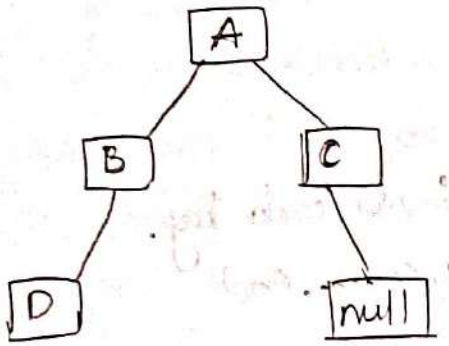
1. Create a 2D table cost of size $(n \times n)$ to store a minimum cost of searching for each sub-tree.
2. Initialize the cost table with infinity for all entries.
3. Fill the cost table using the following recurrence relation:

   cost[i,j] = min(cost[i,k] + cost[k+1,j] + sum(frequencies[i..j]))

   where i and j are the indices of the sub-tree, k is the index of the root node, and sum(frequencies[i..j]) is the sum of frequencies of all keys in the sub-tree.
4. The minimum cost of searching for the entire tree is stored in cost[1,n].

Example:—

Suppose we have 4 keys (A, B, C, D) with frequencies (0.4, 0.3, 0.2, 0.1) respectively. The internal node costs are (1, 1, 1, 1) respectively. Using the dynamic programming solution, we can obtain the following optimal binary search tree.



The minimum cost of searching for the entire tree is 2,3.

0/1 knapsack :—

The 0/1 knapsack algorithm is a dynamic programming approach where items are either completely included or not at all. It consider all combinations to find the maximum total value.

1. As the name suggest, items are indivisible here, we cannot take the fraction of any item.
2. We have to take either on item completely or leave it completely.
3. It is solved using dynamic program approach.

1. knapsack weight capacity = W 2. Number of items each having some weight and profit are value = n

Step-1 :— Draw a table say alphabetical T with (n+1) number of rows and (w+1) number of columns.

Step-2 :— Fill all the boxes of 0^{th} row and 0^{th} columns with 0's.

Step-3 :— Start filling the table row wise top to bottom from left to right.

Use the following formula

$$T[i,j] = \max\{T(i-1,j), value_i + T(i-1,j) - weight_i\}$$

Here T[i,j] = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

Steps :—

1. This step leads to completely filling the table, the value of the last box represent the maximum possible value that can be inset in to the knapsack.
2. To identify the items that must be insert in to the knapsack to obtain their maximum profit.
   (i) Consider the last column of the table.
   (ii) Start scanning the entries from bottor to top.
   (iii) On encountering an entry whose value is not same as the value stored in the ~~entry~~ entry immediately above it, mark the row label of that entry.

(iv) After all the entries are scanned, the marked labels represent the items that must be insert in to the knapsack.

Example 1:- Solve for 0/1 knapsack problem using dynamic programming for the instance $n=3$, $m=6$ profits $(P_1, P_2, P_3)=(1,2,5)$ and weight $(w_1, w_2, w_3)=(2,3,4)$ size $m=4$, $n=8$ $(P_1, P_2, P_3, P_4)=(1,2,5,6)$, weight $(w_1, w_2, w_3)=(2,3,4,5)$.

$m=4$

$n=8$

$(P_1, P_2, P_3)=(1,2,5,6)$

$(w_1, w_2, w_3)=(2,3,4,5)$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	(8)

P	w
1	2
2	3
5	4
6	5

$x_i = \{x_1\ x_2\ x_3\ x_4\}$

$0\ 1\ 0\ 1$

$x_i w_i = 0\times 2 + 1\times 3 + 0\times 4 + 1\times 5$

$= 0+3+5+0$

$= 8$

$x_i p_i = 0\times 1 + 1\times 2 + 0\times 5 + 1\times 6$

$= 2+6$

$= 8$

String Editing:— The string editing problem is a classic problem in computer science. It involves finding the minimum number of operations required to transform one string into another.

Problem Definition:—

Given,

* Two strings str1 and str2 of lengths m and n respectively.
* A set of operations with associated costs.

Dynamic programming solution:—

The string editing problem can be solved using dynamic programming. The idea is to break down the problem into sub-problems and solve each sub-problem only one.

Here's a step-by-step explanation of the dynamic programming solution:

1. Create a 2D table dp of size $(m+1) \times (n+1)$ to store the minimum number of operations required to transform the first i characters of str1 into the first j characters of str2.
2. Initialize the dp table with the following base cases:
   * dp[i][0] = i
   * dp[0][j] = j
3. Fill the dp table using the following recurrence relation:

   dp[i][j] = min(dp[i-1][j-1] + cost(substitution), dp[i-1][j] + cost(deletion), dp[i][j-1] + cost(insertion))

   where cost(substitution), cost(deletion), and cost(insertion) are the costs of substitution, deletion, and insertion operations respectively.
4. The minimum number of operations required to transform str1 into str2 is stored in dp[m][n].

Example:—

Suppose we have two strings str1 = "kitten" and str2 = "sitting". The costs of substitution, deletion, and insertion operations are 1, 1 and 1 respectively.

Using the dynamic programming solution, we can obtain the following minimum number of operations required to transform str1 into str2:

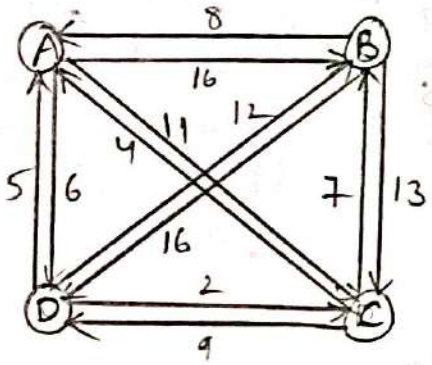
dp[6][7] = 3

The minimum number of operations required to transform str1 into str2 is 3.

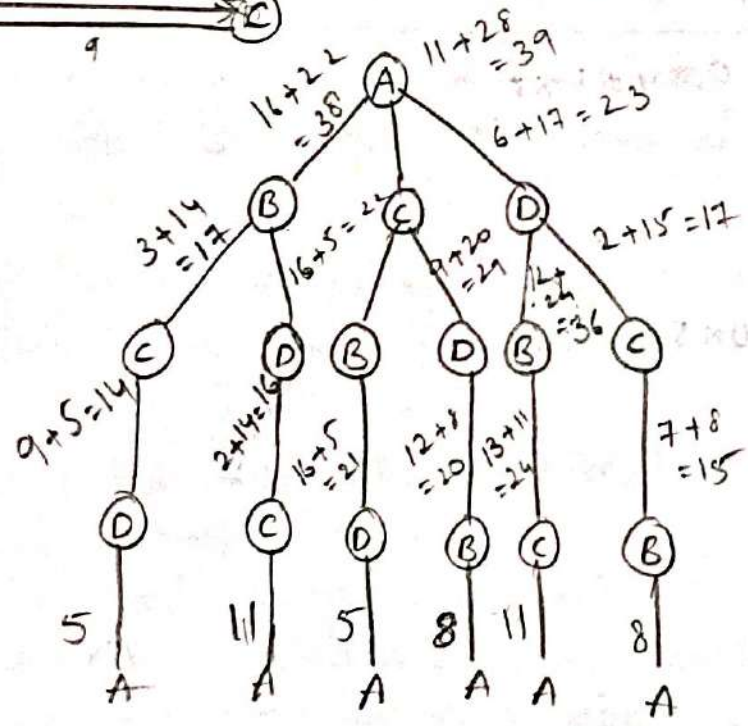
Travelling Sales Person problem:—

1. Notorious competitional problem.
2. It is a dynamic programming involves finding the most efficient root visiting cities once and returning.
3. Dynamic programming optimize solution by breaking the problem into sub-problems, avoiding redundant competitions for a more efficient solution.
4. Dynamic programming break it into sub-problems storing optimal solution to avoid recalculating, ensuring an efficient solution for smaller root before taking larger ones.
5. However, instead of using brute force, using the dynamic programming approach will obtain the solution in lesser time though there is no polynomial time algorithm.

Example:— In the following example, we will illustrate the steps to solve the travelling sales person problem.



	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0



$$g[i,S] = \min\{w(i,j) - g[j,[S-j]]\}$$

$g(A,\{B,C,D\}) = \min(\omega(A,B) + g(B,\{C,D\}))$

$= 16 + ?$

$\min(\omega(A,C) + g(C,\{B,D\}))$

$= 11 + ?$

$\Rightarrow \omega(A,D) + g(D,\{B,C\})$

$= 6 + ?$

$g(B,\{C,D\}) = \omega((B,C) + g(C,\{D\})$

$= 13 + 14 = 27$

$\omega(B,D) + g(D\{C\})$

$= 16 + 6 = 22$

$g(C\{D\}) = \omega(C,D) + g(D,\{\phi\})$

$= 14$

$(D,C), C\{\phi\} = 6 + ?$

$2 + 4 = 6$

$\therefore g(A,\{B,C,D\}) = \min(\omega(A,B) + g(B,\{C,D\}))$

$= 16 + 22 = 38$

Unit-IV

Back Tracking :— It is like trying different paths when you hit a dead end, you back track to the last choice and try a different root.

1. It is a problem solving technique.
2. It is used for solving recursive problems.
3. It uses multi solutions, not a single solution.

Going back and coming until condition is satisfied.

There are some applications we are using the backtracking approach. They are:, 1. N-Queen's problem
2. Sum of Subset
3. Graph colouring
4. 0/1 knapsack problem.

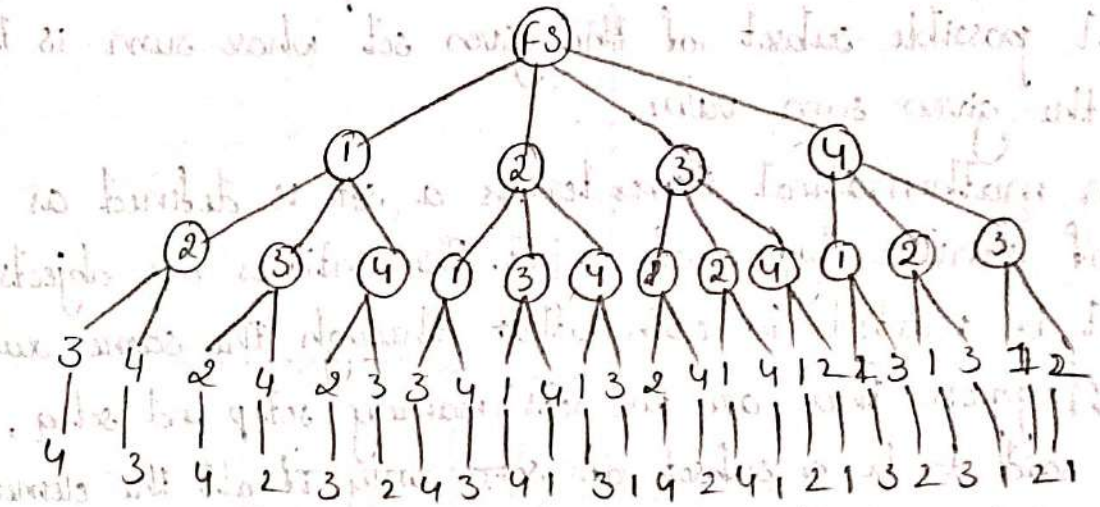
N-Queen's problem :— It is a classic example of back tracking problem, the problem involves placing N-Queen's on an N×N chess board, such that no queen attack another queen. There are three conditions to satisfy the N-Queen's problem.

(i) No two queens are in same row.
(ii) No two queens are in same column.
(iii) No two queens are in same diagonal.

4-Queen's :—

2 4 1 3

3 1 4 2



	Q_1		
			Q_2
Q_3			
		Q_4	

		Q_1	
Q_2			
			Q_3
	Q_4		

Algorithm for N-Queen's :—

Step-1 :— Place the first queen in the top left cell of the chessboard.

Step-2 :— After placing a queen in the first cell, mark the position as a part of the solution and then recursively check if this will lead to a solution.

Step-3 :— Null, If placing the queen does not lead to a solution, then go to the first step and place queens in other cell. Repeat until all the cells are tried.

Step-4 :— If placing queens return a lead to solution return true.

Step-5 :— If all queens are placed return true.

Step-6 :— If all rows are tried and no solution is found, return false.

Sum of subset problem:—

In this, there is a given set with a non-negative integer element and another sum value is also provided. Our task is to find all possible subset of the given set whose sum is the same as the given sum value.

Set:— In mathematical terms a set is defined as a collection of similar types of object. The entities are objects of a set must be related to each other through the same rule.

Subset:— Suppose there are two sets namely set p and set q, the set p is said to be a subset of set q only if all the elements of set p also belong to the set q & vice versa need to be true.

Algorithm for sum of subset:—

Step-1:— First take an empty set.

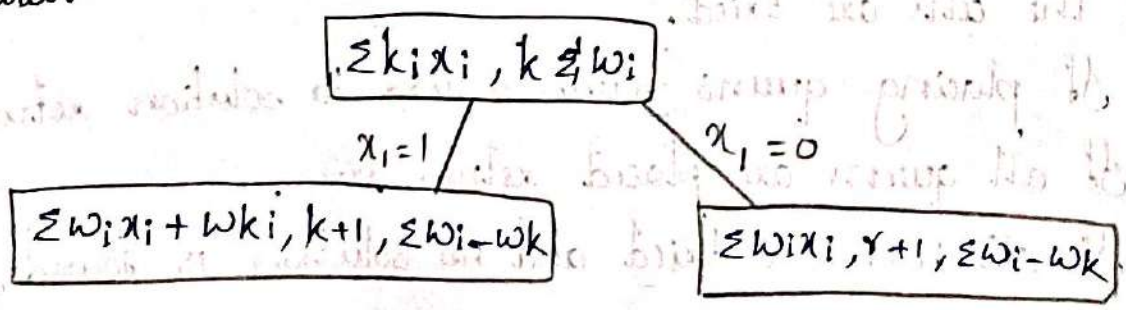
Step-2:— Include the next element which is at 0 to the empty set.

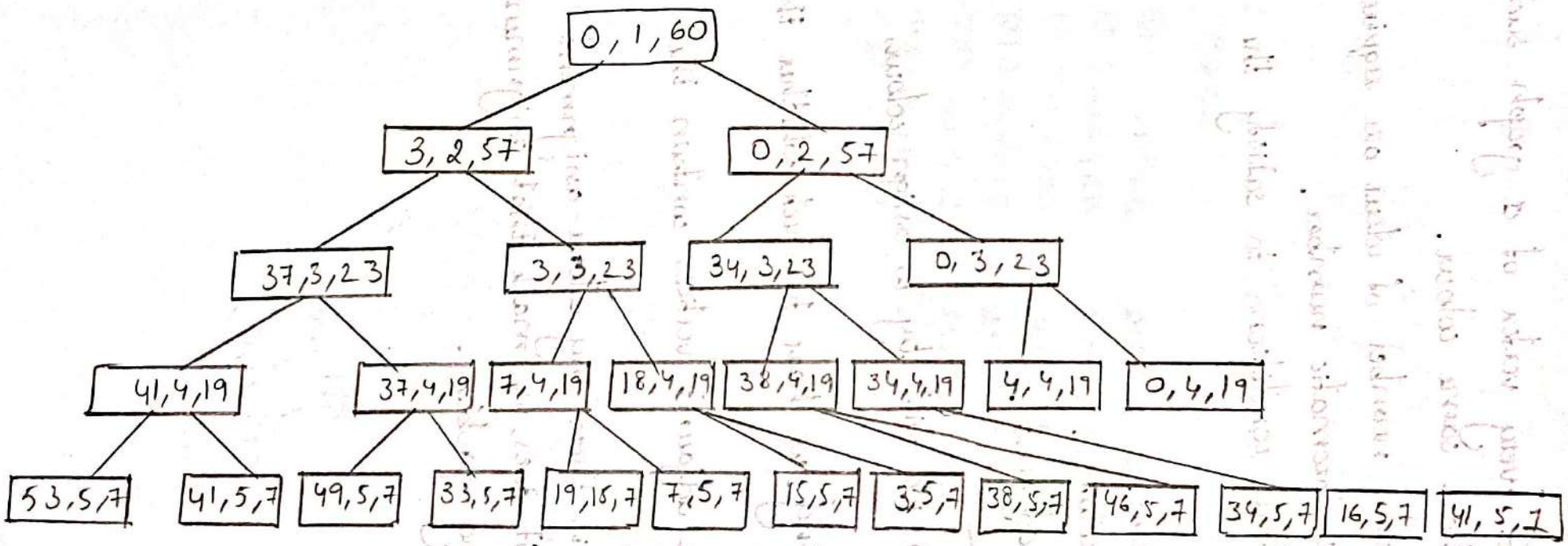
Step-3:— If the subset is equal to the sum value, mark it as the part of solution.

Step-4:— If the subset is not a solution & it is less than the sum value, add the next element to the subset until a valid solution is found.

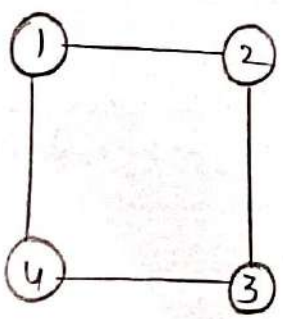
Step-5:— Now move to the next element in the set & check for another solution until all combinations have been tried.

Formula:—

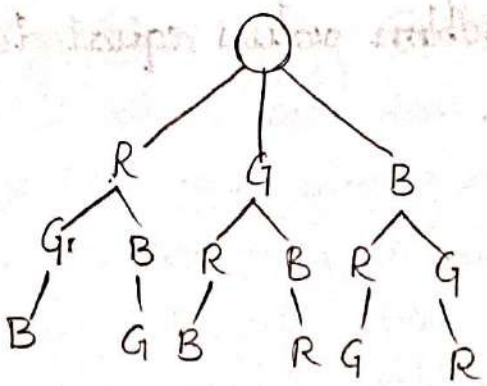



0, 1, 60
3, 2, 57
0, 2, 57
37, 3, 23
3, 3, 23
34, 3, 23
0, 3, 23
41, 4, 19
37, 4, 19
7, 4, 19
18, 4, 19
38, 4, 19
34, 4, 19
4, 4, 19
0, 4, 19
53, 5, 7
41, 5, 7
49, 5, 7
33, 5, 7
19, 15, 7
7, 5, 7
15, 5, 7
3, 5, 7
38, 5, 7
46, 5, 7
34, 5, 7
16, 5, 7
41, 5, 7

Graph colouring :—
Consider G(V,E) colour each and every vertex of a graph such that no two adjacent vertex have same colour.
Chromatic number :— The minimum number of colour are required to colour a graph is called its chromatic number.
In graph colouring there are three conditions to satisfy the given problem.
1. No two colours are in same row.
2. No two colours are in same column.
3. No two colours are in same adjacent.
Algorithm for graph colouring :—
Step-1 :— Starting from vertex 0, we will try to assign colours one by one to different nodes.
Step-2 :— However, before assigning, we have to check whether the colour is safe or not.
Step-3 :— Colour is not safe when adjacent vertices contain the same colour.
Step-4 :— Next, we will check if is there any colour assignment that satisfies the constrain. If it does, we mark that assignment as the solution to the m-colouring problem.
Example :—



Red Green Blue
m = (R, G, B)
m = 3



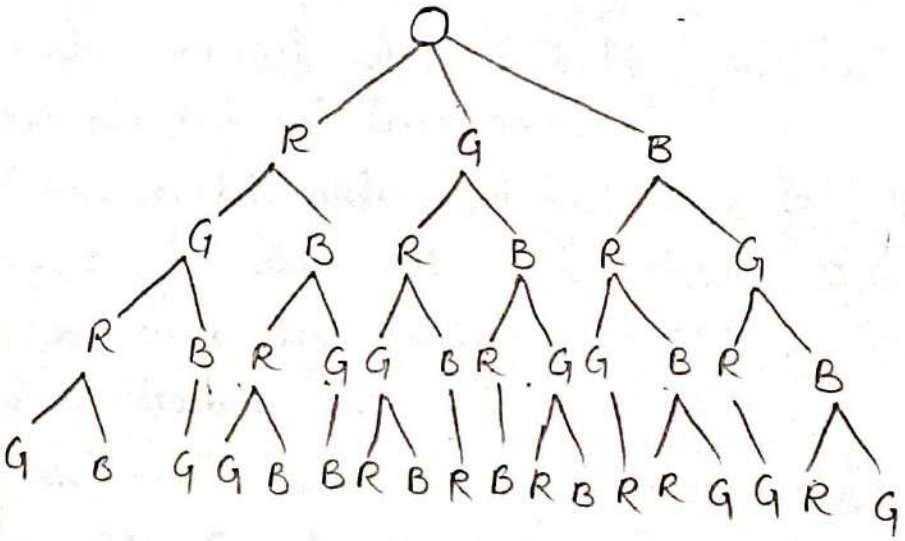
State space tree

3 1 2 1
BRGR

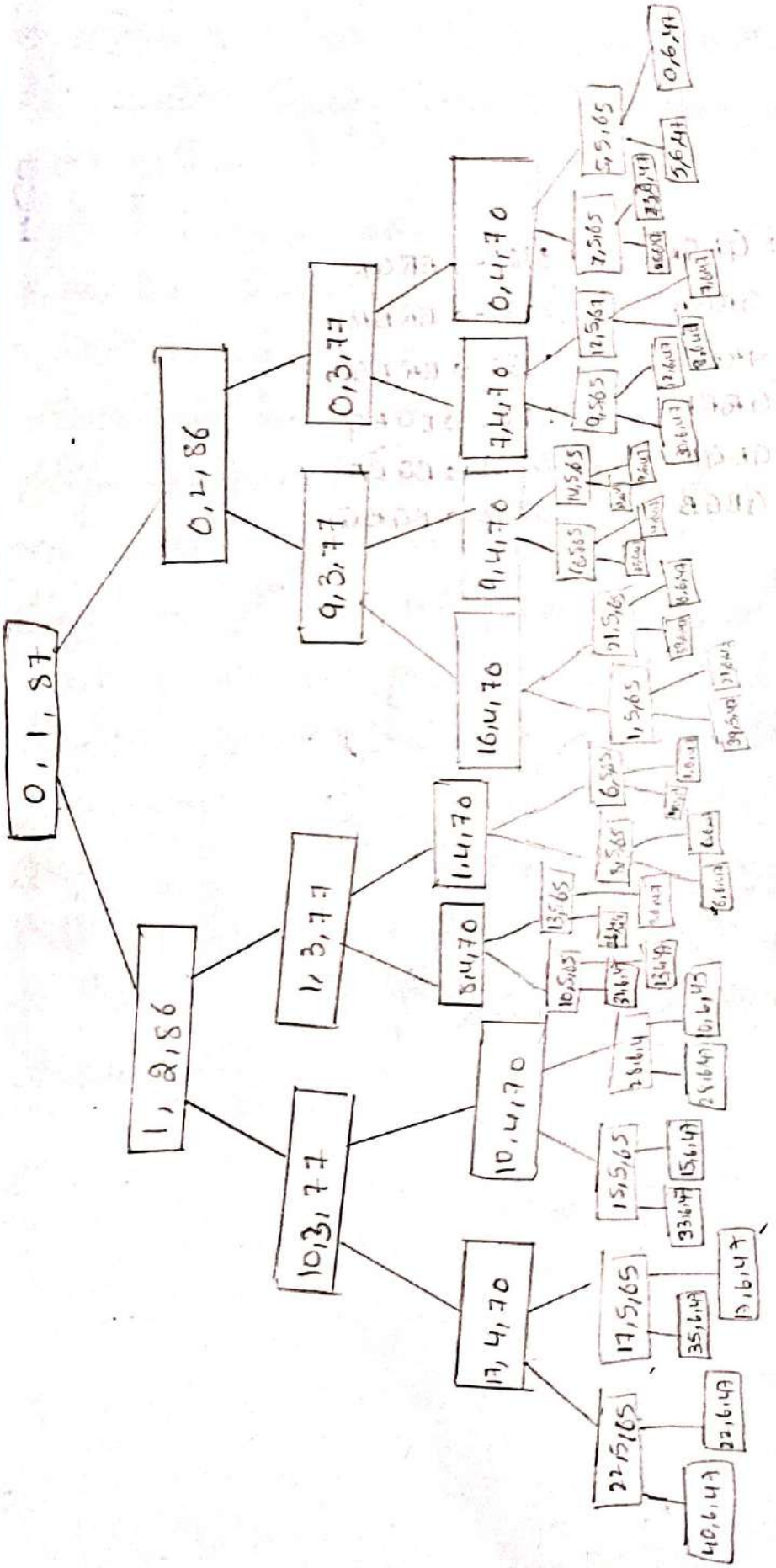
1212 → RGRG
1213 → RGRB
1232 → RGBG
1312 → RBRG
1313 → RBRB
1323 → RBGB

2121 → GRGR
2123 → GRGB
2131 → GRBR
2313 → GBRB
2321 → GBGR
2323 → GBGB

3121 → BRGR
3131 → BRBR
3132 → BRBG
3212 → BGRG
3231 → BGBR
3232 → BGBG

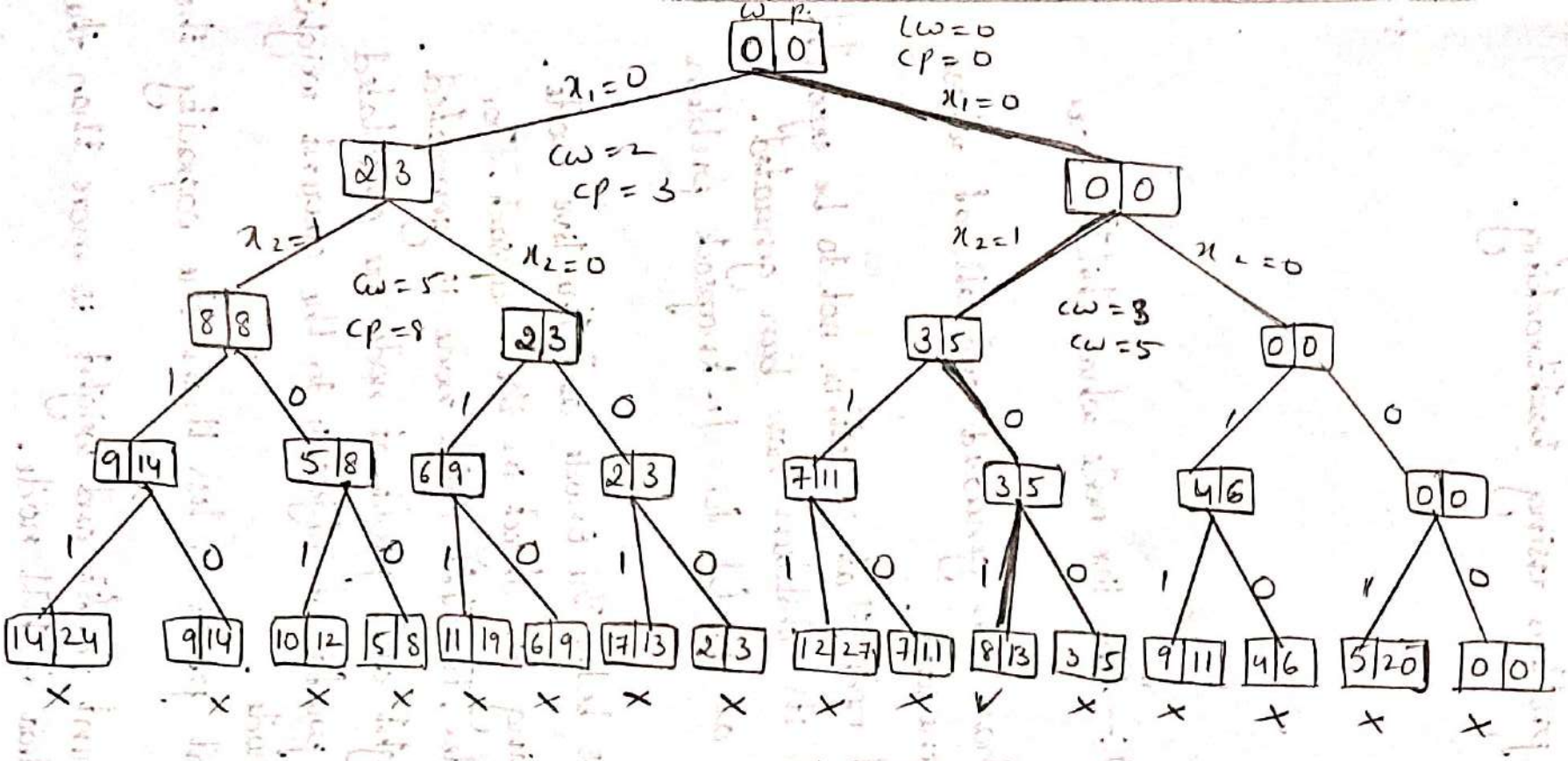


Explain the sum of subset problem values equal to
1, 9, 7, 5, 18, 12, 20, 15, m = 35.



0/1 knapsack problem using Backtracking :—
Given N items where each item has some weight and profit associated with it and also given a bag with capacity k_1, the task is to put the items in to the bag such that the sum of profits associated with them is the maximum possible.
Here in this 0/1 knapsack problem using backtracking.
Some conditions :—
(i) cw = current weight
(ii) cp = current profit
(iii) E node
(iv) Live node
(v) Dead node (or) kill node.
Enode (node being Expanded) :— The live node whose children are currently being generated.
Live node :— A node which has been generated and all of whose children are not yet been generated.
Dead node (or) kill node :— A node that is either not to be expanded further, or for which all of its children have been generated.
There are some steps which are useful for 0/1-knapsack problem using backtracking.
1. Start :— The root node is both the Enode and the live node.
2. Expand :— Consider including an object in the knapsack. For example if object 4 is consider, the left child node is generated.
3. Update :— The current weight and profit at node are calculated by adding the weight and profit of object 4 to the current weight and profit of the parent node.
4. Terminate :— If the current profit is less than the capacity then it should be expanded.
5. Kill (or) Dead :— If the current profit and weight is more than the capacity then it should be dead or kill node.

Example-1 :— Weight - 2 3 4 5, Size-4, Profit-3 5 6 10 and capacity -8 kg.



$0 + 3 + 0 + 5 = 8$ kg.

Part-II

Branch and Bound :— It is a method for solving optimization problems by breaking them down into smaller sub problems and using a bonding function to eliminate sub problems that cannot contain the optimal solution. The branch and bound method is a statespace search algorithm used for optimization problems. It involves partitioning a problem into sub problems (Branching) and solving these subproblems to the optimal level, using bounds to eliminate the need to consider sub-optimal solutions (Bounding).

There are different search techniques in branch and bound

(i) LC search (least cost search)

(ii) BFS

(iii) DFS

LC search :— It uses a least cost ~~search~~ solution to compute the bound values at each node. Nodes are added to the list of live node as soon as they get generated. The node with the least value of a cost function selected as a next Enode.

1. It is used to find optimal solution.
2. It is similar to backtracking but there we are using DFS.
3. In branch and bound we use BFS.
4. Branching is a process of generating subproblems.
5. The statespace tree is useful for to get all possible paths.

Applications of Branch and Bound :—

1. 0/1 - knapsack using B and B.
2. Travelling Sales person problem.

0/1 knapsack using branch and bound:—
It is same as FIFO and it is a LCBB (lower cost bound and branch). Here the knapsack mc bag container. Here to place the objects into knapsack. So, that the profit is minimum in the knapsack we contain the profit and weight and mainly capacity. is used to solve the problem.

Main points (or) keys (or) terms:—

* Here the upper bound cannot take the fraction.
* In lower bound we can use the fraction.
  We have to convert the given profits into negative profit by putting negative sign of each and every profit.
* Lower bounds are same then compare to upper bound.

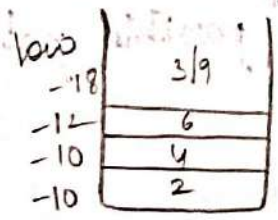
Example 1:— Draw the position of statespace tree generated by LCBB for knapsack instances $n=4$ profits = {10, 10, 12, 18}, weight = {2, 4, 6, 9} and capacity = 15 kg.

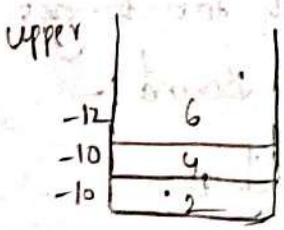
profits = {-10, -10, -12, -18}
weights = {2, 4, 6, 9}
capacity = 15 kg.

$n=4$

Negative profits = {-10, -10, -12, -18}

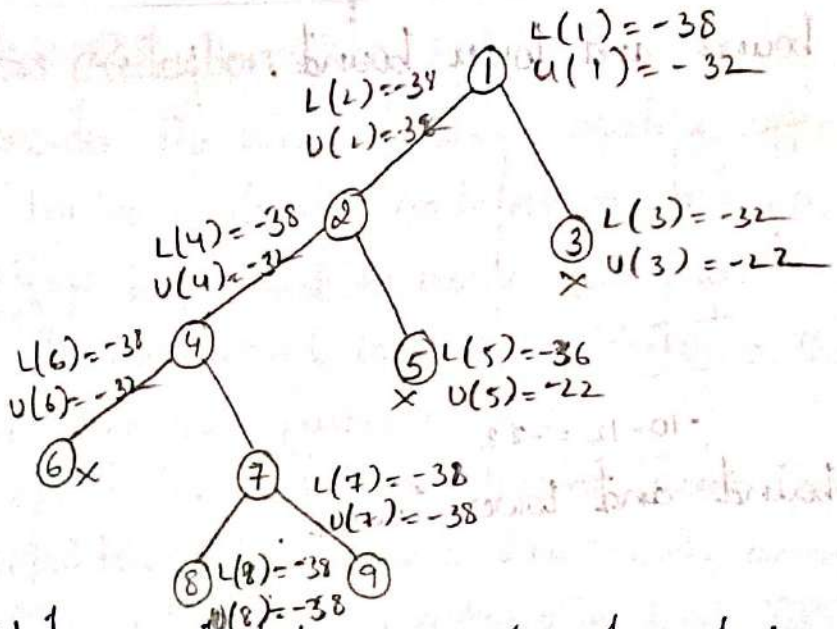
* Node 1, calculate upper bound and lower bound.





$-10-10-12-18 \times 3/9 = -38$

$-10-10-12 = 32$



* Node 2, calculate upper bound and lower bound.

low

-18	3/9
-12	6
-10	4
-10	2

-10-10-12-18 = -38

upper

-12	6
-10	4
-10	2

-10-10-12 = 32

Node 2, $x_1 = 1$

* Node 3, calculate upper bound and lower bound

low

-18	3/9
-12	6
-10	4

-10-12-18 × 5/9 = -32

upper

6	-12
4	-10

-10-12 = -22

Node 3 $x_1 = 0$

* Node 4, calculate upper bound and lower bound.

low

-18	7/9
-12	6
-10	4
	2

-10-10-12-18 × 3/9 = -38

upper

6	
-12	4
-10	2

-10-10-12 = -32

Node 4 $x_1 = 1, x_2 = 1$

* Node 5, calculate upper bound and lower bound.

low

-18	7/9
-12	6
-10	2

-10-12-18 × 7/9 = -36

upper

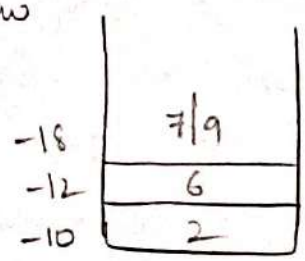
-12	6
-10	2

-10-12 = -22

Node 5 $x_1 = 1, x_2 = 0$

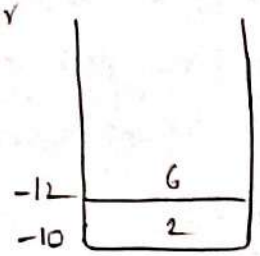
* Node 6, calculate upper bound and lower bound.

low



$-10-12-18\times 7/9 = -36$

upper

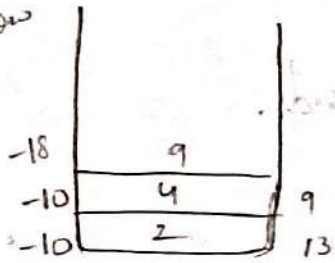


$-10-12 = -22$

Node 6 $x_1=1, x_2=1, x_3=?$

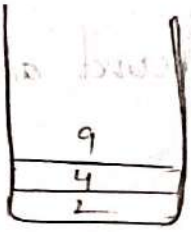
* Node 7, calculate upper bound and lower bound.

low



$-10-10-18 = -38$

upper



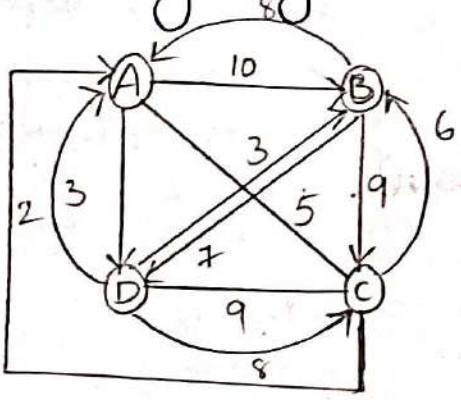
$-10-10-18 = -38$

Node 7 $x_1=1, x_2=1, x_3=0$

Travelling Sales Person Problem Using Branch and Bound:—

Here are the salesman means rules.

1. Visit each city atleast once.
2. Do not visit any city more than once.
3. Starting city and ending city must be same.



	A	B	C	D
A	∞	10	5	3
B	8	∞	9	7
C	1	6	∞	9
D	2	3	3	∞

Rules:—

1. To reduce a matrix, perform the row reduction and column reduction of the matrix respectively.
2. A row or a column is said to be reduced if it contains atleast one entry zero in it.

Row Reduction:—

Consider the rows of above matrix one-by-one.

If the row already contains an entry 0, then

(i) there is no need to reduce that row.

If the row doesn't contain an entry 0, then

(i) Reduce that particular row.

(ii) Select the least value element from that row.

(iii) Subtract that element from each element of that row.

(iv) This will create an entry 0 in that row, thus reducing that row.

Column Reduction:—

Consider the columns of above row reduced matrix one-by-one.

If the column already contains an entry 0, then there is no need to reduce that column.

If the column does not contain an entry 0, then

(i) Reduce that particular column.

(ii) Select the least value element from that column.

(iii) Subtract that element from each element of that column.

(iv) This will create an entry 0 in that column, thus reducing that column

All rows (or) columns represent to infinity (or) atleast one row (or) one column represents to zero.

Row reduction:—

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 10 & 5 & 3 \\ 8 & \infty & 9 & 7 \\ 1 & 6 & \infty & 9 \\ 2 & 3 & 8 & \infty \end{bmatrix} \end{array}\begin{array}{c} \\ 3 \\ 7 \\ 1 \\ 2 \end{array}$$

$$\begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ R_4 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 0 = 3 \end{array} \\ \begin{bmatrix} \infty & 7 & 2 & 0 \\ 1 & \infty & 2 & 0 \\ 0 & 5 & \infty & 8 \\ 0 & 1 & 6 & \infty \end{bmatrix} \\ \begin{array}{cccc} C_1 & C_2 & C_3 & C_4 \end{array} \end{array}$$

column reduction

$$M_1 = \begin{bmatrix} \infty & 6 & 0 & 0 \\ 1 & \infty & 0 & 0 \\ 0 & 4 & \infty & 8 \\ 0 & 0 & 4 & \infty \end{bmatrix}$$

M_1 = Total cost = RR + CR

= 13 + 3

= 16

A → B A rows as ∞ , B rows as ∞ , A → B as ∞

	A	B	C	D	
A	∞	∞	∞	∞	–
B	∞	∞	0	0	–
C	0	∞	∞	8	–
D	0	∞	4	∞	–

Total cost = PMG + RC + (A - R)

= 16 + 0 + 6

= 22 .

A → C A rows as ∞ , C column as ∞ , C - A as ∞

	A	B	C	D	
A	∞	∞	∞	∞	–
B	1	∞	0	0	–
C	∞	4	∞	8	–
D	0	∞	4	8	–

(-4)

$$\begin{bmatrix} 1 & \infty & \infty & \infty \\ 1 & \infty & 0 & 0 \\ \infty & 0 & \infty & 4 \\ 0 & \infty & 4 & 6 \end{bmatrix}$$

T_C = PMC + RC + (A - C)

= 16 + 4 + 0

= 20

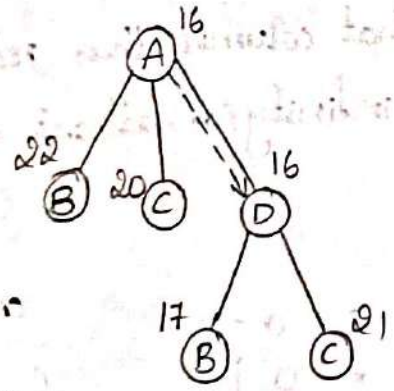
A → D A rows as ∞ , D column as ∞ , D - A as ∞

A	∞	∞	∞	∞	–
B	1	∞	0	∞	–
C	0	4	∞	∞	–
D	∞	0	4	∞	–

T_C = PMC + RC + (A - D)

= 16 + 0 + 0

= 16



$(A \to D) \to B$ A & D rows as ∞, B columns as ∞

B-A as B-D as ∞.

	A	B	C	D	
A	∞	∞	∞	∞	✓
B	∞	∞	0	∞	✓
C	0	∞	∞	∞	✓
D	∞	∞	∞	∞	✓

PMC + R + B-A + B-D

=> 16 + 0 + 1 + ∞

=> 17

$(A \to D) \to C$ A & D rows as ∞, D columns as ∞

C-A as ∞ C-D as ∞

	A	B	C	D	
A	∞	∞	∞	∞	
B	1	∞	∞	∞	(1)
C	∞	4	∞	∞	(4)
D	∞	∞	∞	∞	

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	0	∞	∞
D	∞	∞	∞	∞

PMC + R + C-A + C-D

=> 16 + 5 + 0 + 0

=> 21

Unit-V [Part-I]

NP HARD AND NP-COMPLETE PROBLEMS

Basic Concepts:—

1. Tractability
2. Decision problem
3. Optimization problem
4. Polynomial Time Complexity

1. Tractability:—

Some problems are tractable that is the problem are solvable in a reasonable amount of time called polynomial time, some problem are interactable that is as problem grow large, we are unable to solve them in reasonable amount of time called polynomial time.

2. Decision problem:—

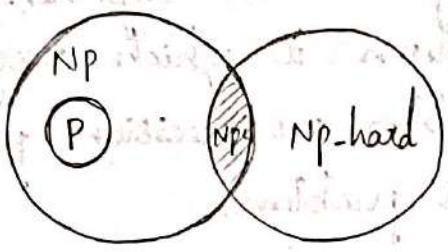
Competitional problem with produces output of yes or no, one or zero are decision problems.

3. Optimization problem:—

Competitional problems where we tried to maximize or minimize some values that is identifying optimal solution to problem.

4. Polynomial Time Complexity:—

An algorithm is of polynomial complexity, if there exist a polynomial $p()$ such that the computing time is $O(p(n))$ for every input size of n. Polynomial time is the worst case running time required to an algorithm to process an input of size n is $O(n^k)$ for some constant k.



Class-P problems:—

1. Class-P problem are the set of decision problem solvable by deterministic algorithm in polynomial time.
2. A deterministic algorithm is (essential) one that always computes the correct answer.

Ex:— Fractional knapsack, MST, Single Source Shortest Path.

Class-NP problems:—

1. NP-problems are set of decision problems solvable by non-deterministic algorithm in polynomial time.
2. A non-deterministic algorithm is one that can this the "the right answer or solution".

Ex:— Hamiltonian cycle (Travelling sales person), CNF (conjunctive normal form).

NP-Complete problems (NPC):—

1. A problem x is a NP-class problem and also NP-complete if and only if every other problem in NP can be reducable. (solvable) using non-deterministic algorithm in polynomial time.
2. The class of problems which are NP-hard and belong to NP.
3. The NP complete problem are always decision problems only.

Ex:— Vertex covering problem, TSP

NP-Hard problem:—

1. A problem x is a NP-class problem and also NP-Hard if and only if every other problem can be reducable using non-deterministic algorithm in exponential time.
2. The class of problems to which every NP-problem reduces.
3. The NP-Hard problem are decision problems and sometimes may be optimization problems.

Ex:— Integer linear programming.

Cook's theorem :—

1. We know that, class p problems are the set of all decision problems solvable by deterministic algorithms in polynomial time similarly class NP problems are set of all decision problems solvable by non-deterministic algorithms in polynomial time.
2. Since deterministic algorithms are a special case of non-deterministic algorithms P and NP.
3. Cook formulated the following question; is there any single problem in NP such that if we showed it to be in P then that would imply that P=NP ? This lead to cooks theorem as; satisfiability is in P if and only if P=NP.

Cook's theorem proof :—

Consider Z denotes a deterministic polynomial algorithm.
A denotes a non-deterministic polynomial algorithm
I denotes input instance of algorithm
n denotes length of input instance
Φ denotes a formula
m denotes length of formula.

1. Now the formula Φ is satisfiable if and only if non-deterministic algorithm A has successfull termination with input I.
2. If the time complexity A is $p(n)$ for some polynomial $p(n)$, then the time needed to construct the formula Φ by algorithm A is given by $o(p^3(n)\log n)$.

3. Therefore complexity of non-deterministic algorithm A is $o(p^3(n)\log n)$. (Np)

(i) Similarly, the formula Φ is satisfiable if and only if the deterministic algorithm z have a successfull termination with input Σ.

(ii) If the time complexity of z is $q(m)$ for some polynomial $q()$, then the time needed to construct the formula Φ by algorithm z is given by $o^3(p^3(n)\log n) + q(p^3(n)\log n)$

Therefore complexity of deterministic algorithm z is

$$o(p^3(n)\log n + q(p^3(n)\log n))(p)$$

(i) If satisfiability is in p, then $q(m)$ is a polynomial function and the complexity of z becomes $o(r(n))$ for some polynomial $r(n)$.

(ii) Hence, p is satisfiable, then for every non-deterministic algorithm A in Np can obtain a deterministic algorithm z in p.

(iii) So, the above construction shows that if satisfiability is in p, then $p = Np$.

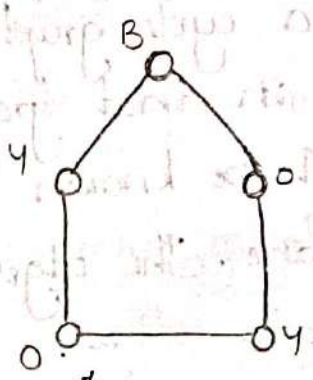
Chromatic Number Decision Problem (CNDP):—

Chromatic number is the minimum number of colors required to properly colour any graph.

(or)

Chromatic number is the minimum number of colors required to color any graph such that no two adjacent vertices of it are assigned the same color.

Example:—



Color(m) = 3

B, Y, O

chromatic number = 3

In this graph,

1. No two adjacent vertices are colowied with the same colour.
2. Minimum number of colours required to properly color the vertices = 3
3. Therefore, chromatic number of this graph is 3.
4. We cannot properly colour this graph with less than three colours.

Types of chromatic number of graphs:—

There are various types of chromatic number of graphs which are described as follows:

1. Cycle graph
2. Planner graph
3. Complete graph
4. Bipartite graph
5. Tree

1. Cycle graph :— A graph will be known as a cycle graph if it contains n edges and n vertices ($n >= 3$) which form a cycle of length n.

* There can be only 2 or 3 number of degrees of all the vertices in the cycle graph.
* The chromatic number in a cycle graph will be 2, if the number of vertices in that graph is even.
* The chromatic number in a cycle graph will be 3, if the number of vertices in that graph is odd.

2. Planar graph :— A graph will be known as planar graph if it is drawn on a plane. The edges of the planar graph must not accross each other.

Chromatic number :—

(i) In a planar graph, the chromatic number must be less than or equal to 4.

(ii) The planar graph can also be shown by all the above cycle graph.

3. Complete graph :— A graph will be known as complete graph if only one edge is used to join every two vertices every vertex in a complete graph is connected with every other vertex.

In this graph every vertex will be color with a different colour. that means in that complete graph two vertices do not contain the same colour.

Chromatic number :—
In a complete graph, the chromatic number will be equal to the number of vertices in that graph.

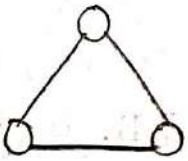
4. Bipartite graph :— A graph will be known as bipartite graph if it contains two sets of vertices A and B. The vertex of A can also join with the vertices of B. That means the edges cannot join the vertices with a set.

Chromatic number :—
In any Bipartite graph, the chromatic number is always equal to 2.

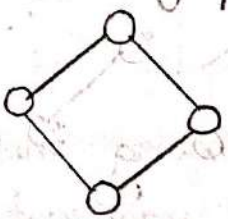
5. Tree :— A connected graph will be shown as a tree if there are no circuits in that graph. In a tree, the chromatic number will equal to 2. No matter how many vertices are in the tree. Every Bipartite graph is also a tree.

Chromatic number :—
In any tree, The chromatic number is equal to 2.

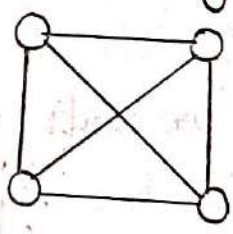
Cycle graph :—



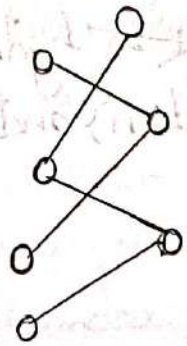
Planner graph :—



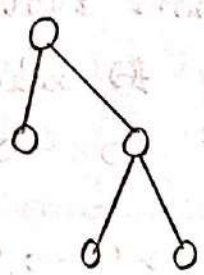
Complete graph :—



Bipartite graph :—



Tree :—



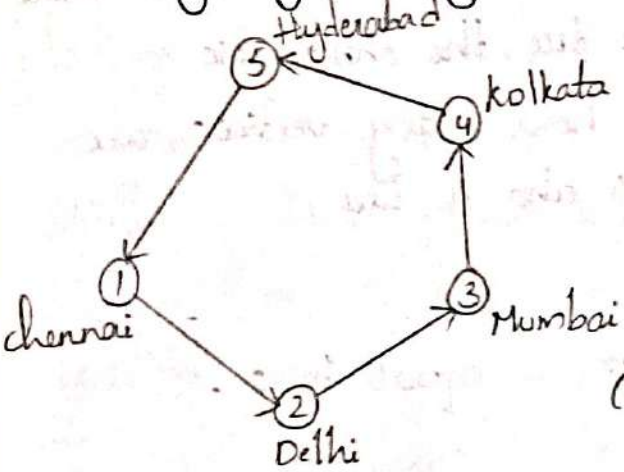
Travelling Sales Person Using Np Hard Graph decision problem

1. It is a solution where a salesman has to start from one place to and go to all other cities just once and then come back to their own place.
2. It is all about finding the minimum distance path.
3. The polynomial type hardness is called Np hard which defines the property of a class of problems.

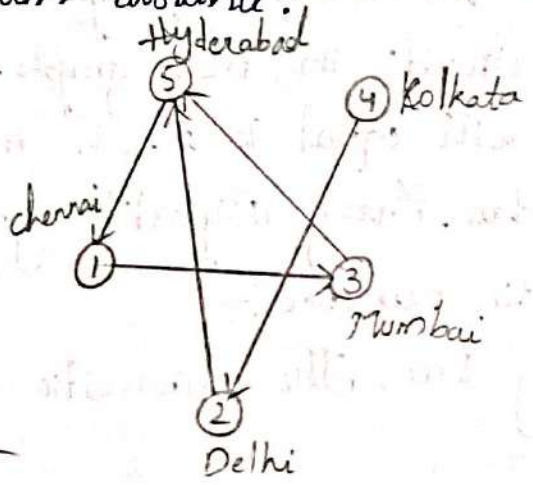
The subset sum is a simple example of Np hard problems..

Example :—

We have five cities to understand how salesman travel to each city by talking the minimum distance.



Fig(1)



$(n-1)!/2$

Fig(2)

Which one we will prefer among these two possibilities of figure. In this two figures it represent the minimum distance path.

In figure-1 the salesman travel from chennai to hyderabad it takes a lot of time to reach its own location, and total distance cover

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$$

In figure-2 the salesman directly reach the point at mumbai, which saves the time and also cover the total minimum distance

$$1 \rightarrow 3 \rightarrow 5 \rightarrow 1$$

The possibility is given by using the formula $(n-1)!/2$

Example-2 :— Can we find the possibilities of 32 cities problem.

$$\frac{(n-1)!}{2} \Rightarrow \frac{(32-1)!}{2} = \frac{31}{2} = 15.5.$$

B+ Trees :—

B+ Trees are extensions of B trees designed to make the insertion and searching operations more efficient.

The properties of B+ trees are similar to the properties of B-trees, except that the B-trees can store keys and records in all internal nodes and leaf nodes while B+ trees store records in leaf nodes and keys in internal nodes.

In B+ tree is that all the leaf nodes are connected to each other in a single linked list format and a data pointer is available to point to the data present in disc file.

This helps fetch the records in equal number of disc ~~place~~ acces.

Since the size of main memory is limited, B+ trees act as the data storage for the records that could not be stored in the main memory. For this, the internal nodes are stored in the main memory and the leaf nodes are stored in the secondary memory storage.

Properties of B+ trees :—

* Every node in a B+ tree, except root, will hold a maximum of m children and (m-1) keys, and a minimum of $\lceil \frac{m}{2} \rceil$ children and $\lceil \frac{m-1}{2} \rceil$ keys, since the order of the tree is m.
* The root node must have no less than two children and atleast one search key.
* All the paths in a B-tree must end at the same level. i.e., the leaf nodes must be at the same level.
* A B+tree always maintains stored data.

Basic operations of B+ tree :—

→ Insertion
→ Deletion
→ Searching

Example :— elements 10 12 15 18 20 25

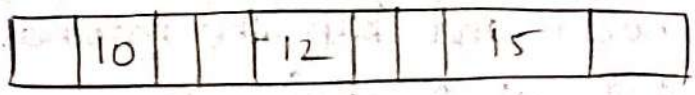
order 4

$m = 4$

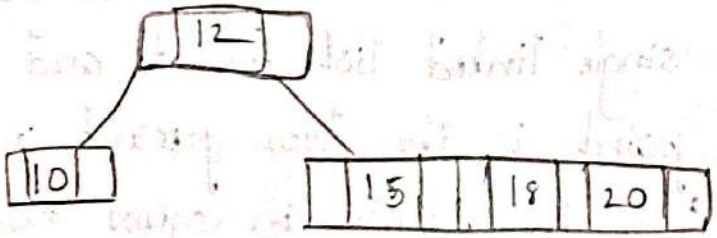
$\frac{m}{2} = 2$

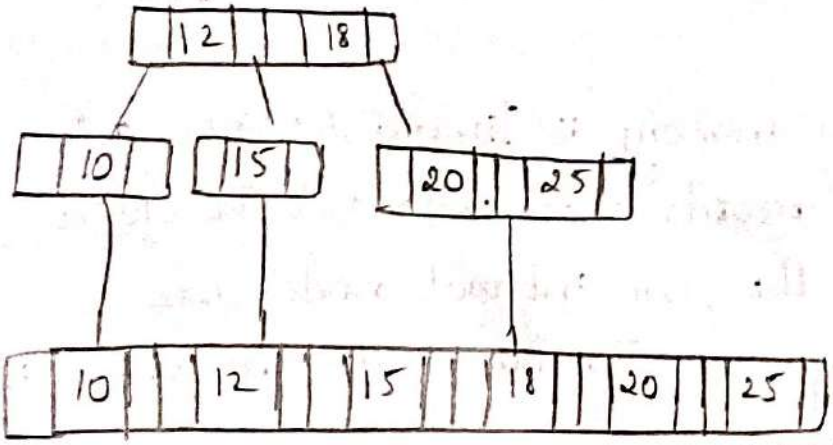
$m-1 = 3$

$\frac{m-1}{2} = 1.5$



18



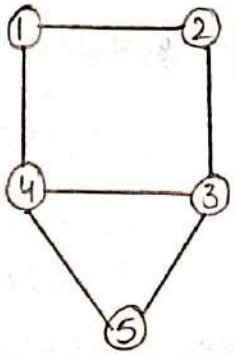


← disc file

Clique Decision Problem [CDP]:—

It is a sub graph of any graph and is complete the size of clique is the number of vertices in it.

Example:—



Problem:—

To prove clique decision problem is np-Hard.

Solution:—

To prove a problem is Np-Hard

1. Pick a problem L, already known as Np-Hard
2. Show that L_1 and L_2.
3. Thus, L_2 is Np-Hard.

Question: To prove that or clique decision problem is Np-Hard pick CNF-Satisfiability that is known Np-Hard problem. Prove that CNF = satisfiability $\propto$ CDP.

Answer:— Consider a proportional formula in CNF

CNF is conjugate normal form.

$$F = \wedge C_i$$

$$1 \le i \le k$$

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

C_1 C_2

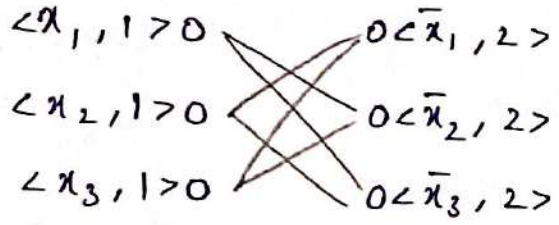
$\langle x_1, 1\rangle$ $\quad$ $\langle \bar{x}_1, 2\rangle$

$\langle x_2, 1\rangle$ $\quad$ $\langle \bar{x}_2, 2\rangle$

$\langle x_3, 1\rangle$ $\quad$ $\langle \bar{x}_3, 2\rangle$

$V = \{\langle \tau, i\rangle \mid \tau$ is a literal in clause $i\}$

$E = \{(\langle \tau, i\rangle, \langle s, j\rangle) \mid i \neq j$ and $\tau \neq \bar{s}\}$



Take a clique with vertices $\{\langle \bar{x}, 1\rangle, \langle \bar{x}_2, 2\rangle\}$

By setting $x_1 =$ true

$x_2 =$ false

$x_3 =$ true / false

$F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

$= (T \vee F \vee T) \wedge (F \vee T \vee F)$

$= T \wedge T$

$=$ True

Scheduling identical processors:—

The Scheduling identical processors problem is a classic Np-Hard problem in scheduling theory. It involves scheduling a set of tasks on a set of identical processors to minimize the maximum completion time (makespan).

Problem Definition:—

Given:—

* A set of n tasks ($T_1, T_2, \ldots, T_n$)
* A set of m identical processors ($p_1, p_2, \ldots, p_m$)
* A processing time (t_i) for each task T_i
* A release time (r_i) for each task T_i
* A deadline (d_i) for each task T_i.

Example:—

Suppose we have 4 tasks (T_1, T_2, T_3, T_4) and 2 identical processors (p_1, p_2). The processing times, release times, and deadlines are:

Task	Processing time	Release time	Deadline
T_1	3	0	16
T_2	2	1	5
T_3	4	2	8
T_4	1	3	4

The makespane is 6.

Job Shop Scheduling:— The Job shop scheduling problem is a classic Np-Hard problem in scheduling theory. It involves scheduling a set of jobs on a set of machines to minimize the maximum completion time (makespan).

Problem Definition:—

Given

* A set of n jobs ($J_1, J_2, \ldots, J_n$)
* A set of m machines ($M_1, M_2, \ldots, M_m$)
* A processing time (t_{ij}) for each job J_i on each machine M_j.
* A due date (d_i) for each job J_i.

Example:—

Suppose we have 3 jobs (J_1, J_2, J_3) and 2 machines (M_1, M_2).

The processing times are:

Job	M_1	M_2
J_1	3	2
J_2	2	4
J_3	1	3

The due dates are:

Job	Due date
J_1	6
J_2	5
J_3	4