UNIT- 1:

Introduction to Algorithm Analaysis, Space and Time complexity analysis, Asymptotic Notations, AVL Trees - creation, insertion, deletion operations & applications. B Trees - creation, insertion, deletion operations & analysis.

Write a Algorithm for finding largest number among 3 numbers.

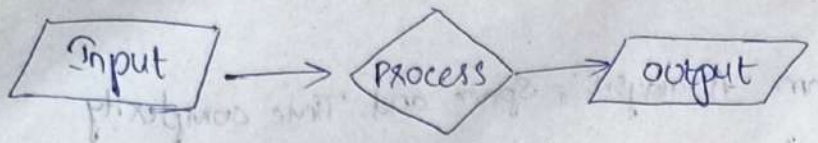
1: Start
2: Declare a,b,c
3: if (a>b)
   a is large
   else if(a>c)
   b is large
   else
4: if (a>c)
   'a' is large
   else
   'c' is large
5: if(b>c)
   'b' is large
   else
   'c' is large

```
Start
Declare a,b,c
if a>b
  if a>c
  Pf(" a is large")
else
  Pf (" c is large")
else
  b>c
  Pf(" b" is large)
```

Algorithm:-

→ The term Algorithm was introduced by "Donald Knuth".
→ Algorithm is a step by step procedure for solving a problem, it is a well defined instructions for performing specific computational tasks.
→ Algorithms are fundamental to computer science and to a play very imp. role in designing efficient solutions for various problems.

How Algorithm's work:-



Algorithm.

INPUT:

The algorithm receives zero or more input data.

Processing:

The algorithm performs a series of operations on the input data

OUTPUT:

The algorithm produces one or more desired outputs.

Characteristics of algorithm:-

1) INPUT: The algorithm receives zero or more input data.

2) OUTPUT: The algorithm produces one or more desired outputs.

3) Unambigious: The algorithm should be unambigious means each of its steps should be clear in all aspects and must lead to only one meaning.

4) Finiteness: The algorithm must be finite i.e., it should terminate after a limited time.

5) Correctness: The algorithm steps must write with correct meaning.

6) Language independent:- Algorithm must be language independent i.e., it must be just plain instructions that can implemented in any language.

How to write an algorithm:-

1) Define the problem: clearly state the problem to be solved

2) Design the Algorithm: choose an appropriate algorithm, design and develop a step by step procedure.

3) Implement the Algorithm: Translate the algorithm into a programming language.

4) Test and Debug: Execute the algorithm with various inputs to ensure its correctness & efficiency.

5) Analyse the Algorithm: Determine its time and space complexity & compare it to alternative algorithms.

Types of Algorithms :-

Analysis of Algorithm :-

Types :- 1) Back Tracking algorithm, Recursive algorithm, Searching algorithm, Hashing algorithm, Array, stacks, linked list, Graph, Trees, Queue, Sorting.

(Analysis of algorithm :-

1) Define the problem: A problem is a function or a mapping of inputs to outputs, a program is an installation of an algorithm in any programming language.

2) Design the Algorithm: Choose an appropriate algorithm, design & develop a step by step procedure.

3) Implement the algorithm :- Translate the algorithm into a programming language.]

Analysis of algorithm :-

→ Algorithm analysis is an important part of computational complexities.

→ Analysis of algorithm is a process of analysis, the problem solving capability of the algorithm in terms of the time and size required.

→ However, the main concern of analysis of the algorithm is the required time of performance.

→ If the problem is having more than one algorithm then the best one is decided by the analysis based on two factors. 1) Time complexity analysis 2) Space complexity analysis.

Time complexity of an algorithm:-

1) It is defined as the process of determining a formula for total time required towards the execution of that algorithm.
2) It can be calculated by using two methods, one- PRIORI Analysis & two- POSTERIORI Analysis

PRIORI Analysis	POSTERIORI Analysis
1) It is an absolute analysis	1) It is an relative analysis
2) It is independent of language, compiler & types of hardware	2) It is dependent of langauge, compiler & types of hardware
3) It will give approximate answers.	3) It will give exact answers.
4) It uses asymptotic notations to represent how much time the algorithm will take inorder to complete its execution.	4) It doesn't give asymptotic notations.
5) It is done before execution of an algorithm.	5) It is done, after execution of an algorithm.
6) It is cheaper than Posteriori analysis.	6) It is costly than priori analysis becz of requirement of software and hardware for execution.
7) Maintenance phase is not required to run the algorithm.	7) Maintenance phase is required to run the algorithm.

The time complexity of the operations:

1) The choice of data structures should be based on the time complexity of the operations that will be performed.
2) The time complexity is defined in terms of how many times it take to run as the given algorithm based on the

length of the input.

3) The time complexity of an algorithm is the amount of time it takes for each statement to complete.

Space complexity of an Algorithm:-

It is defined as the process of a formula for prediction of how much memory space is required for the successful execution of the algorithm. The memory space is generally considered as the primary memory.

The space complexity of operations:-

→ The choice of data structure should be based on the space complexity of operations that will be performed.

→ The amount of memory used by a algorithm to execute and is represented by space Complexity.

→ Becz of algorithm requires memory to store input data & temporary values while running the space complexity at input data space.

Cases in Complexity:-

1) Best case complexity
2) Average case "
3) Worst " "

Asymptotic Notations:-

→ It is a way to describe the running time or space complexity of an algorithm

→ The three most commonly used notations are 'O' (big O), Ω (omega) 'Ω' (omega) & 'θ' notation

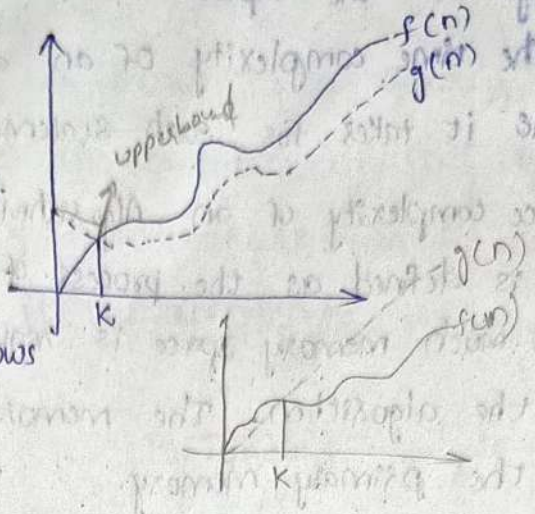
1) Big 'O' notation:

This notation provides an upper bound on the growth rate of an algorithm running time or space usage.

It represents the worst case scenario i.e, the max. amount of time or space of an algorithm may need to solve a

Problem.

A function f(n) can be represented in the order of g(n) i.e., O(g(n)). Function g(n) is an upper bound for the function f(n), as g(n) grows faster than f(n)

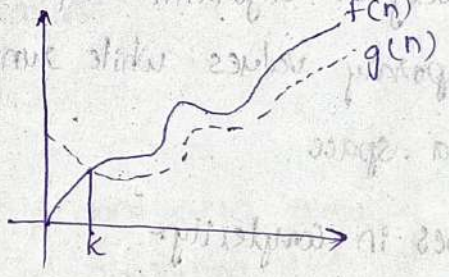


2) Omega (Ω) (or) Big 'Ω' :-

This notation provides a lower bound on the growth rate of an algorithm running time or space usage.

It represents the best case scenario i.e., the minimum amount of time or space of an algorithm may need to solve a problem.

It means function 'g' is a lower bound for function 'f', after a certain value of 'n', 'f' will never go below 'g'.

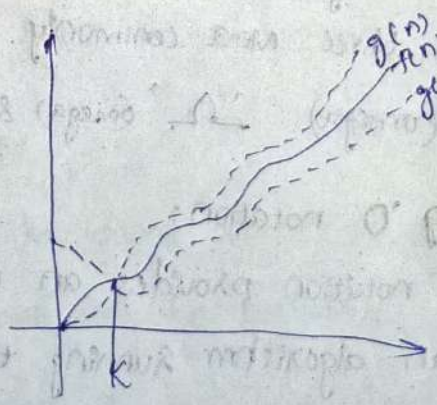


3) 'Θ' Notation: (Theta):

This notation provides both an upper & lower bound on the growth rate of an algorithm running time or space usage.

It represents avg case scenario i.e., the amount of time (or) space of an algorithm typically needs to solve a prblm.

This means function 'g' is tight bound for function 'f'.

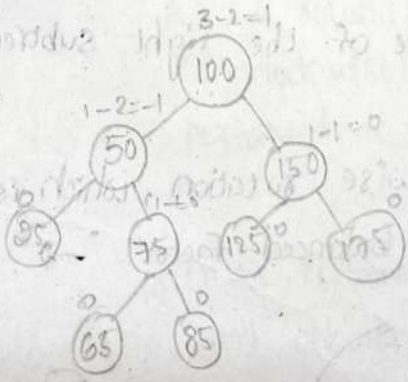


AVL TREES :-

* AVL tree is invented by Adelson Velsky Landas in the year of 1962
* AVL Tree can be defined as height balanced binary search Tree in which each node is associated with a balance factor.
* Balance factor = (height of left subTree) − (height of right sub tree)
* Tree is said to be balanced if balance factor of each node is either "−1, 0 or 1" otherwise the tree will be unbalanced.
* If Balance factor of any node is "1", it means that the left sub tree is one level higher than the RST
* If Balance factor of any node is "0" it means that the LST and RST contains equal height
* If balance factor of any node is "−1" it means that the LST is one level lower than the RST.

Ex:- Keys: 100, 50, 150, 25, 75, 65, 85, 125 & 175

Draw a tree & find balance factor.?



Operations on AVL Tree:-

1) creation
2) Insertion
3) Deletion

1) Creation: To create a node by using the structure user defined data type

Syntax:- struct node {
int data;
struct node *next;
};

2) Insertion: Insertion in AVL Tree is performed in the same way as it is perform in a binary search tree, the newnode is added into AVL tree as the leafnode.

* However, it may lead to violation in the AVL Tree. AVL tree property & therefore, the tree may need to Balancing. the tree can be balanced by apply the Rotation.

* Rotation is required only if the balance factor of any node is disturbed among inserting the newnode, otherwise the Rotation is not required.

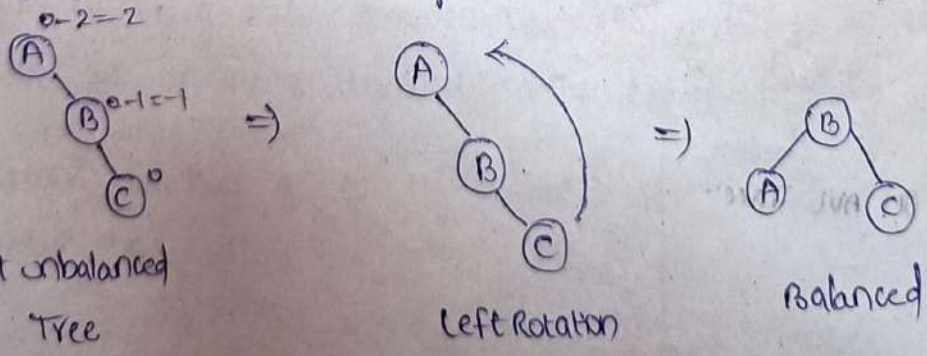
* Depending upon the type of insertion, the rotations are categorized into two types:
1. Single Rotation → LL, RR
2. Double rotation → LR, RL

RR Rotation:-

1) When Binary Search Tree becomes unbalanced due to a node is inserted into the right subtree of the right subtree of A then, we perform RR rotation.

2) RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balanced factor "-2".

Ex:-



Right unbalanced Tree

Left Rotation

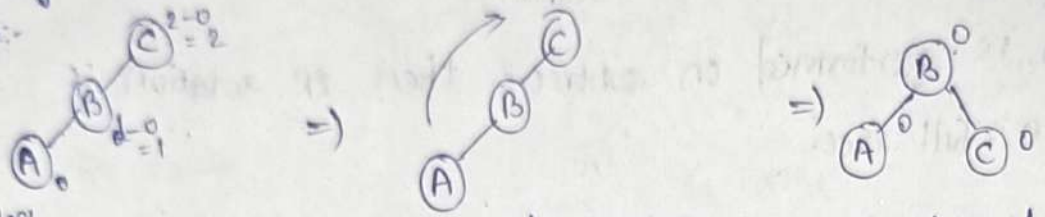
Balanced

⇒ LL Rotation:-

1) If the Binary search tree becomes unbalanced due to a node is inserted into the left subtree of the left subtree of 'C' then we perform LL rotation.

2) LL rotation is a clockwise rotation, which is applied on the edge below a node is having balanced factor.

Ex:-



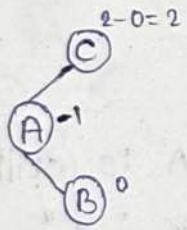
left unbalanced Tree — Right rotation — Balanced

LR Rotation:-

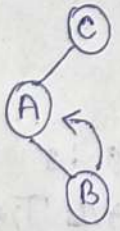
LR Rotation = RR rotation + LL Rotation

i.e, first RR rotation is perform on subtree and LL rotation is performed on full tree, the full tree means the first node from the path of inserted in a node, whose Balanced factor is other than $-1, 0, 1$.

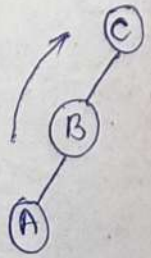
Ex:-



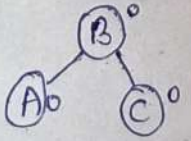
'A' node B has been inserted into the right subtree of 'A'. The left subtree of 'C'. because "C" has become an unbalanced node having balancing factor "2". This case is called LR rotation.



AS LR rotation = RR + LL rotation. Hence RR (anticlock wise) on subtree rooted at "A" is performed at first by doing RR rotation.



After Performing RR rotation node "C" is still unbalanced. i.e, having Balance factor is "2". Now, we can perform LL rotation on full tree.

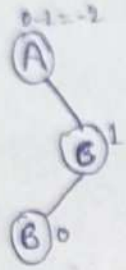


Balance factor of each node is Now either $-1, 0, 1$

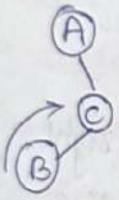
RL Rotation:

RL Rotation = LL rotation + RR Rotation.

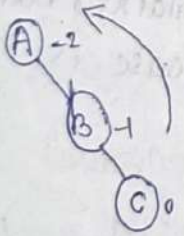
First LL rotation is performed on subtree & then RR rotation is Performed on full tree.



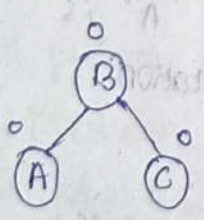
'A' node 'B' has been inserted into the left subtree of 'C', The right subtree of 'A' becz which 'A' has become an unbalanced node having balance factor "-2" This case is RL rotation where inserted node is in the left sub tree of right subtree of 'A'.



AS RL rotation = LL + RR, hence LL (clockwise) on Subtree rooted at "C" is performed first.



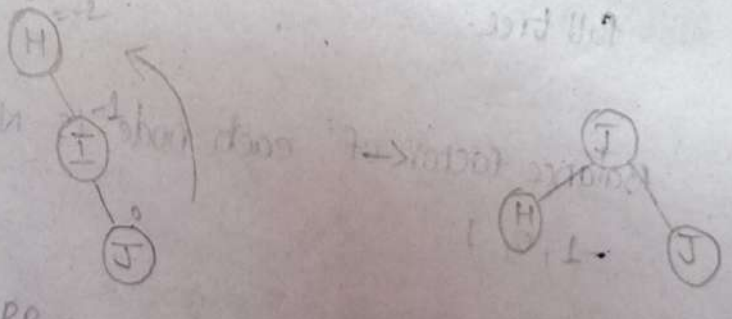
After performing LL Rotation node A is still unbalanced i.e., having balance factor "-2". Now, we can perform RR rotation (anticlockwise) on full tree

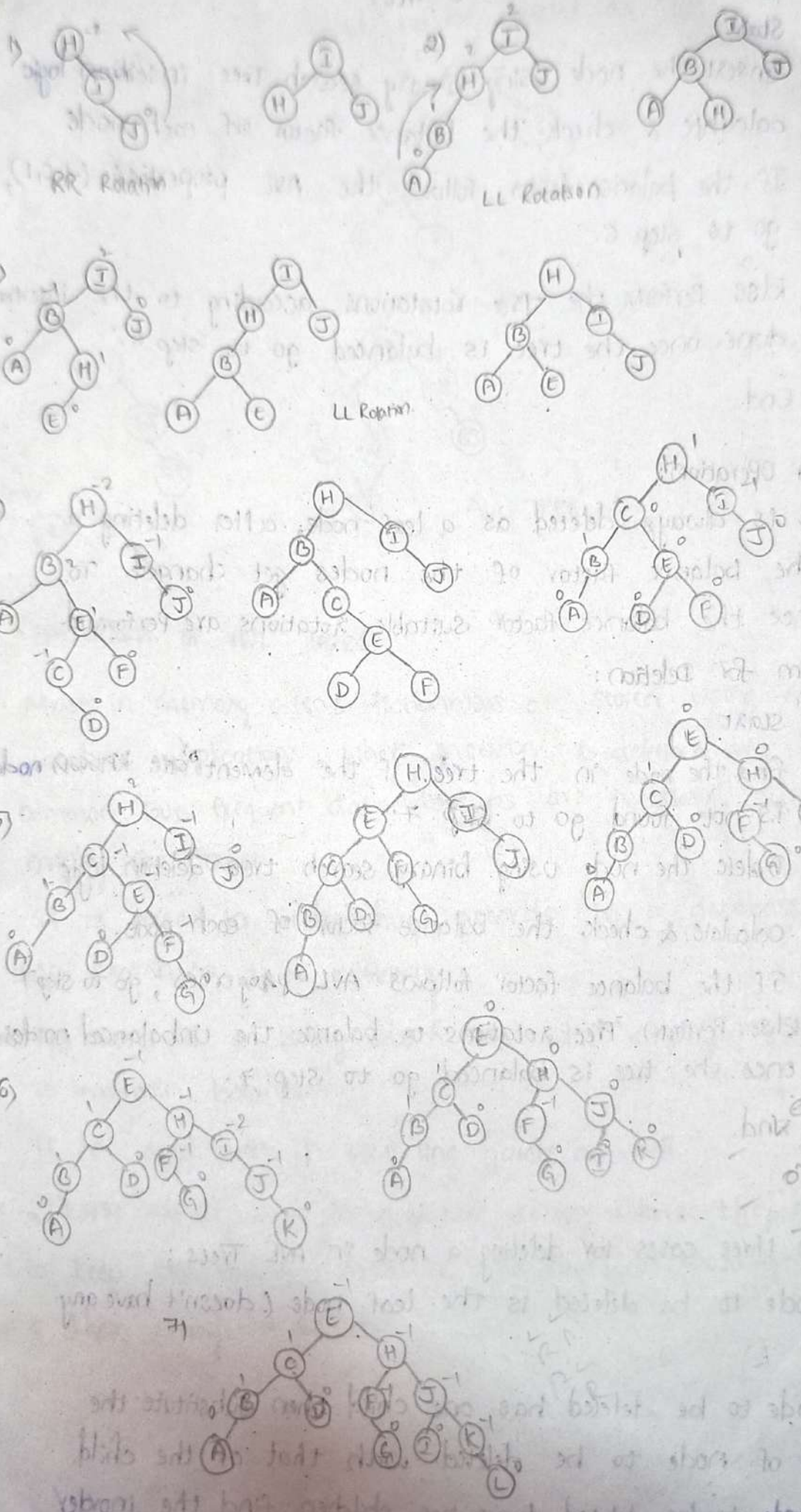


Now the balance factor of each node is either "-1", 0, "1" then we call as AVL Tree.

Ex:- Constauct an AVL Tree having the following elements.

H, I, J, B, A, E, C, F, D, G, K, L




RR Rotation
LL Rotation
LL Rotation

Algorithm for insertion in AVL Tree:

Step 1: Start

Step 2: Insert the node using binary search Tree insertion logic

Step 3: calculate & check the balance factor of each node.

Step 4: If the balance factor follows the AVL properties (-1, 0, 1), go to step 6.

Step 5: Else Perform the ~~Pre-~~ rotatations according to the insertion done. once the tree is balanced go to step 6.

Step 6: End.

Deletion Operation:

A node is always deleted as a leaf node, after deleting a node the balance factor of the nodes get changed. To rebalance the balance factor suitable rotations are Performed.

Algorithm for Deletion:

Step 1: start

Step 2: find the node in the tree. if the element ~~or~~ are known node is not found go to step 7.

Step 3: Delete the node using binary search tree deletion logic

Step 4: calculate & check the balance factor of each node.

Step 5: If the balance factor follows AVL properties, go to step 7.

Step 6: Else Perform Tree rotations to balance the unbalanced nodes. once the tree is balanced go to step 7.

Step 7: End.

Note:-

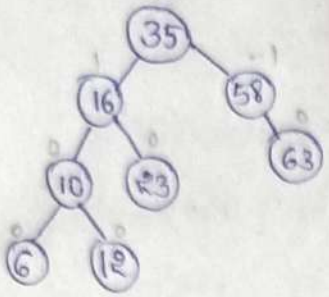
There are three cases for deleting a node in AVL Trees:

1) If node to be deleted is the leaf node (doesn't have any child)

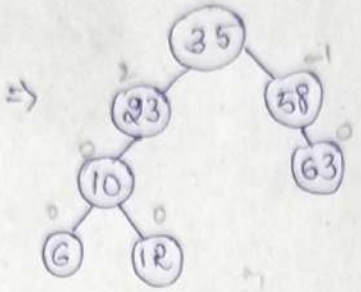
2) If node to be deleted has one child then substitute the content of node to be deleted with that of the child.

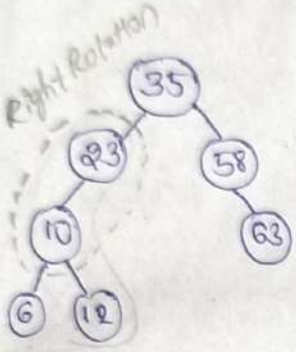
3) If node to be deleted have two children find the inorder successor of node to be deleted. (i.e, node with a min value of key in the right sub tree).

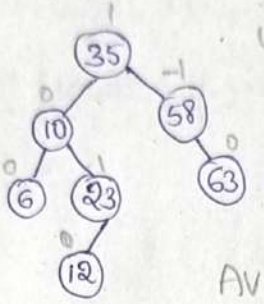
Ex:-



Node to be deleted as "16"







AVL TREE

Applications of AVL Trees:-

* Most in Memory sets & dictionaries are stored using AVL Trees.
* Database applications, where insertions & deletions are less common but frequent data lookups are necessary, also frequently employ AVL Trees.
* It is used to index huge records in a database & also search in that efficiently.
* AVL Tree is a binary search tree which employees rotation to maintain balance.
* It has application in story line games as well.
* It's use mainly used in corporate sectors where they have to keep the information about the employees working there & their change in shifts.

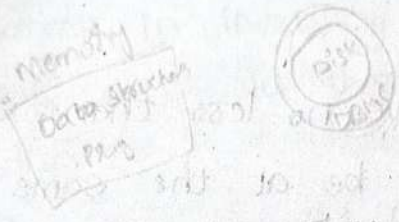
```
#include<stdio.h>
#include<stdlib.h>
struct node{
 int key;
 Structnode * left;
 structnode * right;
 int height;
};
int get Height (structnode* n){
 int if (n==NULL)
    return 0;
    return n-> height;
 }
structnode * create Node (int key){
 structNode* node = (structNode*) malloc (size of (struct Node));
 node-> key = key;
 node->left = NULL;
 node-> right= NULL;
 node ->height = 1;
 return node;
}
int max(int a, int b){
  return (a>b) ? a:b;
}
int getbalance factor (structNode*n)
{
if (n==NULL){
    return0;
 }
 return getHeight (n->left) - getHeight (n-> right);
 }
```

```
struct Node* rightRotate (struct Node* y){
    struct Node* x = y->left;
    struct Node* T2 = x->right;
    x->right = y;
    y->left = T2;
x->height = max(getHeight (x->right), getHeight (x->left)) +1;
y->height = max(getHeight ( y->right), getHeight ( y->left)) +1;
return x;
}
struct Node* leftRotate (struct Node* x){
struct Node* y= x->right;
struct node *T2 = y->left;
 y->left = x;
 x->right = T2;
 x->height = max ( getHeight (x->right), getHeight (x->left)) +1;
 y->height = max (getHeight ( y->right), getHeight (y->left)) +1;
 return y;
}
struct node* insert ( struct Node* node, int key
if (node == NULL)
return · createNode (key);
if (key < node -> key)
node->left = insert (node->left, key),
 else if ( key->node->key)
node->right = insert (node->right, key);
node->height = 1 + max(getHeight (node->left), getHeight (node->right));
int bf= getBalancedfactor (node);
```

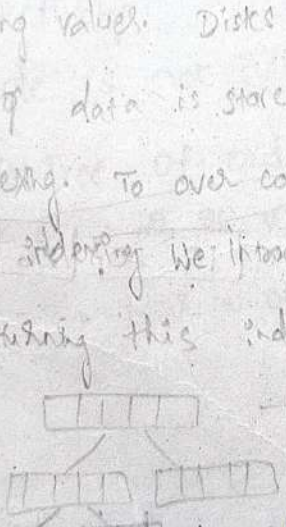
```
// left left case
if (bf > 1 && key < node->left->key){
  return rightRotate(node);
}
// Right Right case
if (bf < -1 && key > node->right->key) {
   return leftRotate(node);
}

// left Right case
if (bf > 1 && key > node->left->key){
   node->left = leftRotate (node->left);
   return rightRotate(node);
}
// Right left case
if (bf < -1 && key < node->right->key){
   node->right = rightRotate (node->right);
   return leftRotate(node);
}
return node;
}
Void PreOrder (structnode * root)
{
 if (root != NULL)
 {
 Printf ("%d", root->key);
 PreOrder (root->left);
 PreOrder (root->right);
 }
}
```

```
int main()
{
  struct Node* root = NULL;
  root = insertion(root, 1);
  root = insertion(root, 0);
  root = insertion(root, 4);
  root = insertion(root, 5);
  root = insertion(root, 6);
  root = insertion(root, 3);
  PreOrder(root);
  return 0;
}
```

B-Trees :



Disks are used to store information. There information is accessed through indexing values. Disks consists sectors, tracks, blocks. Large amount of data is stored through multilevel indexing. To over come time complexity of Multilevel indexing we introduce sub level indexing. By turning this indexing from left to right called as B-Trees



-> we can store multiple values based on order. All leaf nodes must be at same level

1) B-Tree is a self balanced Tree as well as a specialized M-way Tree is used for Disk access.

2) A disk access is the memory access to the computer disc there the information is stored & disc access time is the time taken by the system.

3) B-Tree was developed in the year 1972 by Bayer and MCC Reight with the name of Height Balanced M-way Search Tree.

→ B-Trees are extended of Binary search Trees that are specialized in a M-way searching, since the order of B-Trees is "M" m.

→ Order of a tree is defined as the max. no. of childs a node can accomodate.

Properties of B-Trees:-

1) Every node in a B-Tree will hold a max. of "m" children & "m-1" keys. Since, the order of the tree is "m".

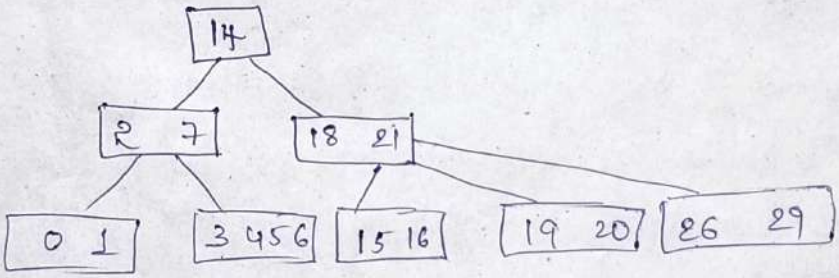
2) Every node in a B-Tree except root & leaf can hold atleast "m/2" childrens.

3) The root node must have no less than 2 children.

4) All the leaf nodes must be at the same level.

5) A B-Tree always maintain sorted Data. (Ascending order only)

Ex:-



Operations on B-Trees:-

1) Insertion operation on B-Tree:-

1) While inserting an element to B-Tree you have to check whether the element is already present in the tree. 2) If the element is found, then the insertion doesn't take place. If not, we will proceed further with the insertion.

3) Two cases are needed more than max no. of keys to be taken care of while inserting the key in the leaf node.

case 1: Node doesn't contain more than max. no. of keys - simply insert the key in the node at its proper place.

case 2: Node contains max no. of keys - key is inserted to the fullnode & then a split takes place splitting the

full node into three parts. The second or median key value move to parent node & the first and third parts acts as left & right children.

steps to insert an element in B-Tree:-

1) Calculate the max. no. of keys in the node based on the order of the B-Tree.

2) If the tree is empty, root node is allocated & the key is inserted & act as a rootnode.

3) Search the appropriate node for insertion.

4) If the node is full

4.1) insert the elements in increasing order.

4.2) if the elements are greater than the max no. of keys split at the median.

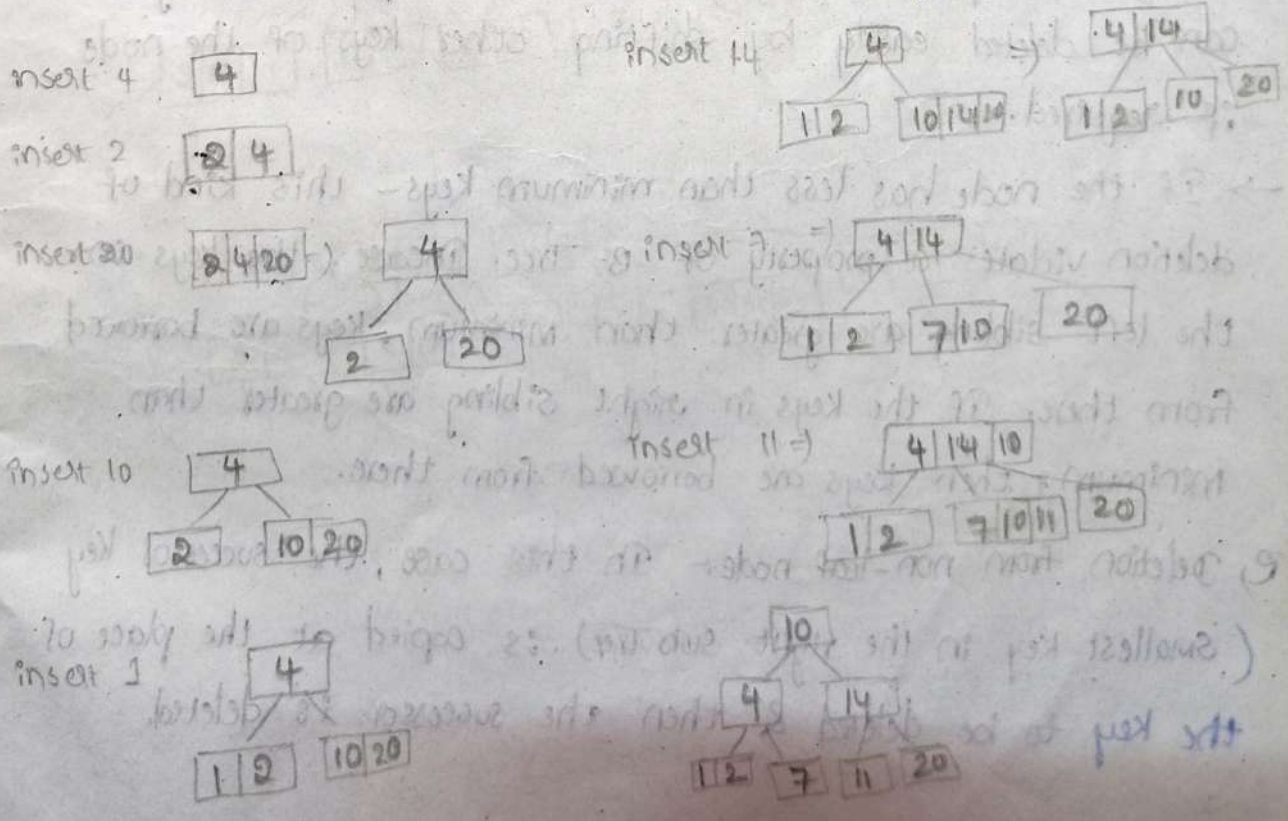
4.3) Push the median key upwards & split the left & right keys respectively.

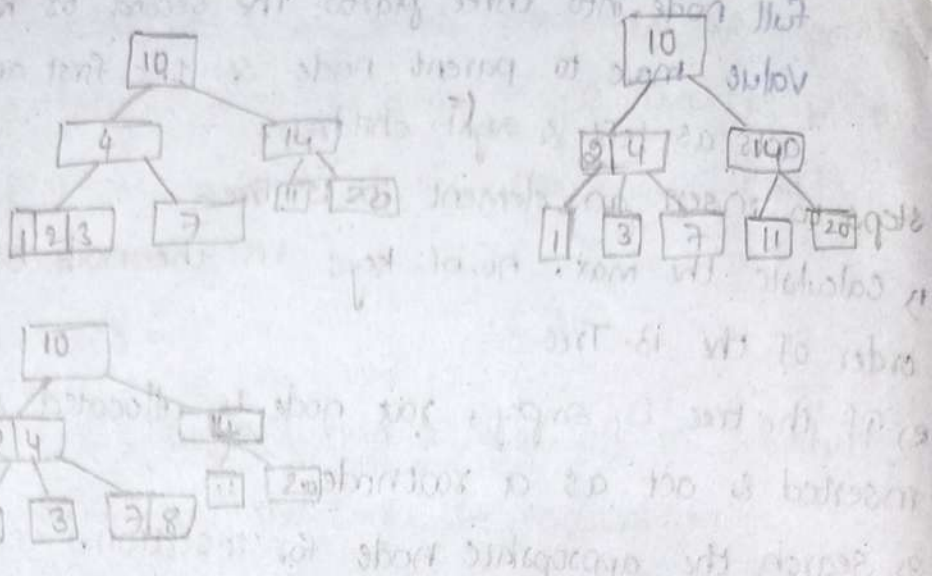
5) If the node is not full insert the node in increasing order

Ex:- Create a B-Tree of order 3.

The elements are 4, 2, 20, 10, 1, 14, 7, 11, 3, 8

Since, order M=3, Max. no. of keys for a node = M-1
= 3-1
= 2





Deletion Operation on B-Tree :-

During deletion we need to ensure that the no. of keys in the node after deletion doesn't go below the min. no. of keys that a node can hold.

Steps to delete an element in B-Tree :-

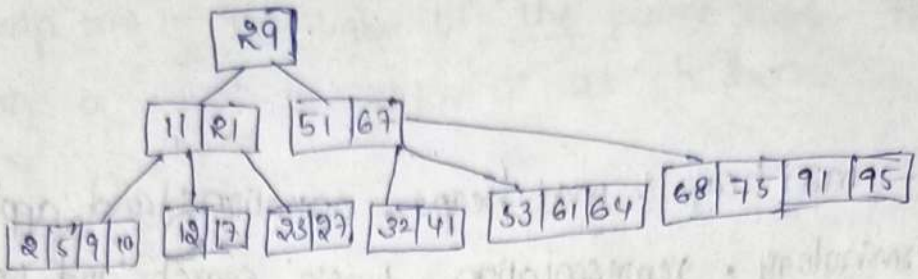
1) Deletion from a leaf node
2) Deletion from a non-leaf node.

1) Deletion from leaf node: If the node has minimum keys - deletion of key doesn't violate any property & the key can be deleted easily by shifting other keys of the node if required.

→ If the node has less than minimum keys - this kind of deletion violates a property of B-Tree. Incase, the keys in the left sibling are greater than minimum, keys are borrowed from there. If the keys in right sibling are greater than minimum, then keys are borrowed from there.

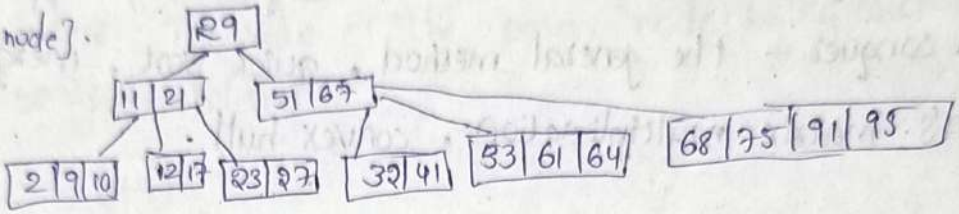
2) Deletion from non-leaf node :- In this case, the successor key (smallest key in the right sub Tree) is copied at the place of the key to be deleted & then the successor is deleted.

Ex:-



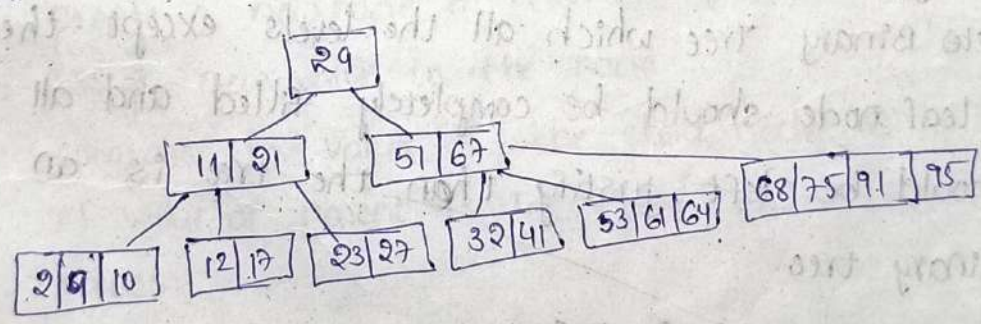
Deleting 5:-

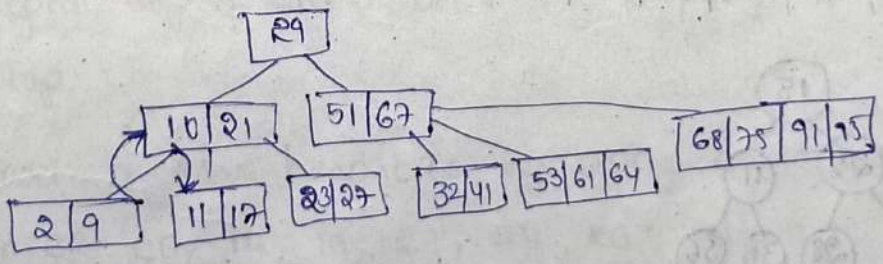
[Deleting leaf node].



Deleting 12:

Here, key 12 is to be deleted from the node 12 17 since, this node has only minimum keys, we will try to borrow from its left sibling 2 9 10 which has more than minimum keys. The parent of these nodes 11 21 contains the separator key '11'. so, the last key of the left sibling '10' is moved to the place of the separator key & the separator key is moved to the underflown node.





Heap Trees - min heap & max heaps, operations and applications
Graph - Terminology, representation, basic search and traversal, connect components and biconnected components, applications.
Divide & conquer - the general method, quick sort, merge sort, strassen's matrix multiplication, convex hull.

Heap Tree:-

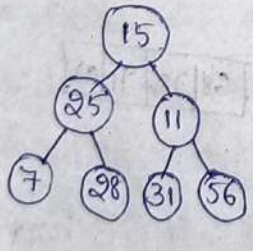
⇒ Heap Tree is an special case of balanced binary tree of data structures, where the root node value is not compared with its children and arranged accordingly.

⇒ A heap tree is a complete binary tree, the binary tree which the node can have atmost two children.

Complete Binary Tree:-

In complete Binary Tree which all the levels except the last level i.e, leaf node should be completely filled and all the nodes should be left justify then the tree is an complete Binary tree.

Ex:- 15, 25, 11, 7, 28, 31, 56



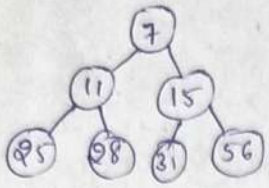
Types of Heap Trees:-

The heap trees can be classified into two types

i) Min heap tree
ii) Max heap tree

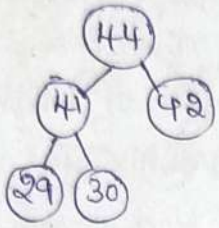
i) Min heap tree :- The value of the parent node should be less than or equal to either of its children.

Ex:-



ii) Max heap tree:- The value of the parent node is greater than or equal to its children.

Ex:-



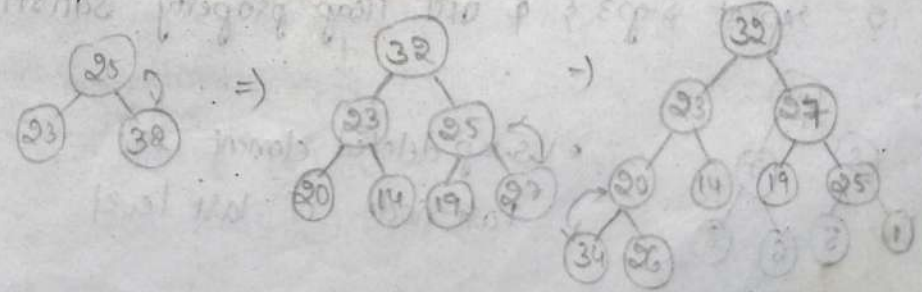
Operations on heap trees :- 1) Insertion
2) deletion

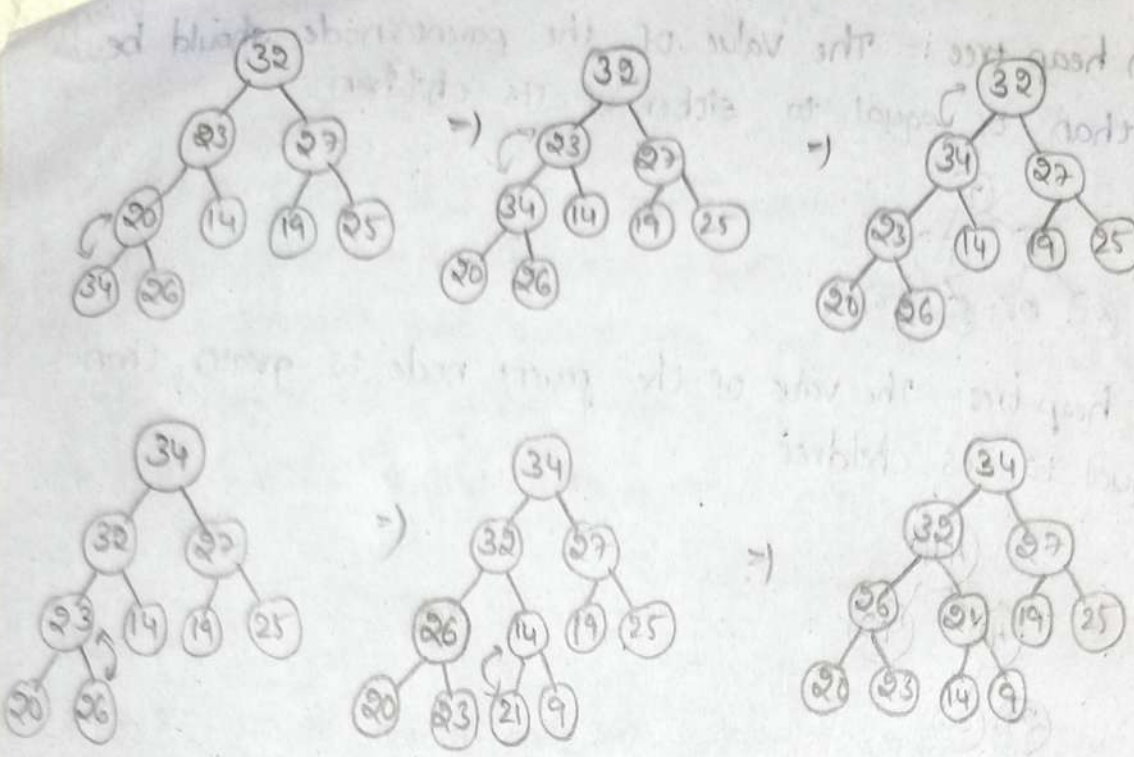
Max Heap Insertion :-

Algorithm:-

Step 1 : start
Step 2 : create a new node at the end of the key
Step 3 : Assign new value to the node.
Step 4 : compare the value of the child node with its parent.
Step 5 : If value of parent is less than child then swap the values.
Step 6 : Repeat step 4 and 5 until the property is satisfied.
Step 7 : Stop

Ex:- Construct a Max heap tree by using below elements.
25, 23, 32, 20, 14, 19, 27, 34, 26, 21, 9





Deletion operation on Max heap tree:-

When we delete an element from the Max heap. There are two cases:

case1:- Deletion of the leaf node

In this case, we can directly delete the leafnode from the max Heap tree. After deleting the leaf node we don't need to change any value in the max Heap.

case2:- Deletion of root node (or) Parent node

Algorithm:-

Step1:- Remove root node

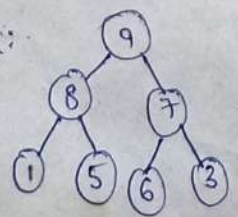
2:- Move the last element of last level to root.

3:- Compare the value of this child node which is parent

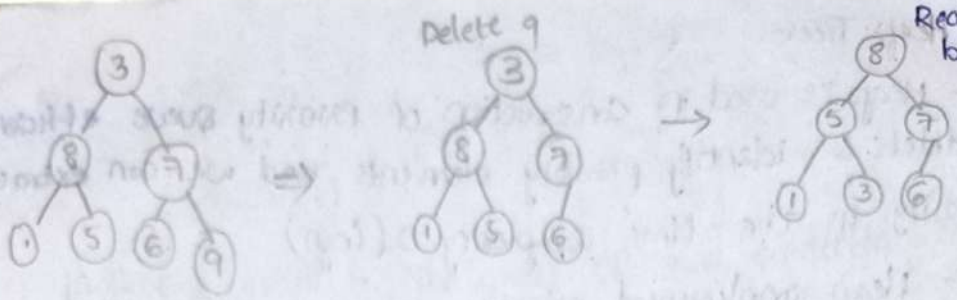
4:- If the value of parent is less than child then swap then

5:- Repeat Step3 & 4 until Heap property satisfied.

Ex:



* swap deleted element to last node of last level



Insertion operation on Min Heap Tree:-

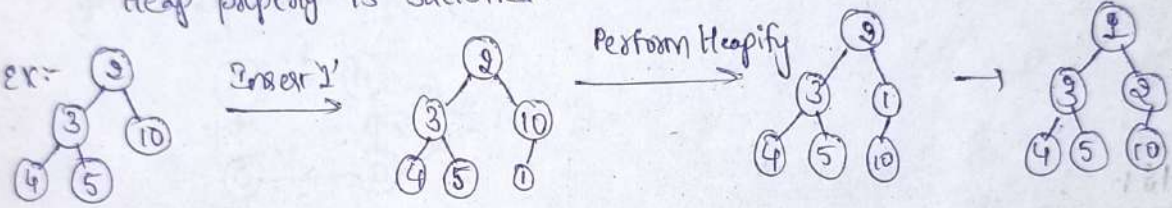
The elements can be inserted into the minimum Heap it involves the following steps:-

Step 1:- Add the new element at end of the Heap, in the next available position in the last level of the tree.

Step 2:- Compare new element with its parent, if parent is greater than new element, swap them.

Step 3: Repeat step 2 until the parent is smaller than or equal to new element

Step 4: The new element is now in its correct position in the Min. Heap & Heap property is satisfied.



Deletion in Min. Heap Tree:-

Algorithm:- [Deletion of root or parent node]

Step 1: Start

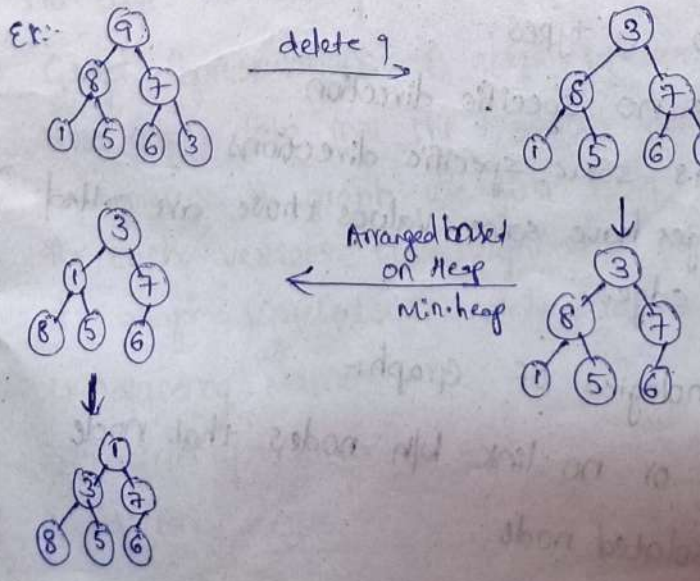
2: Remove root node.

3: move the last element of last level to root

4: Compare value of child node, with its parent

5: if value of parent is greater than child then swap them

6: repeat step 3 & 4 until heap property satisfied.



Applications on Heap Tree:-

→ Priority Queue :- Heap is used in construction of priority queue efficiently. we can insert, delete & identify priority elements and we can extract the element priority in the time complexity O(Logn)

→ Graph Algorithm :- Heap implemented priority queues are also used in Graph algorithms such as Dijsk trace & Prims algorithm.

→ Order statistics :- we can use easily heap data structure to find the largest or smallest element in the array.

→ Embedded System :-

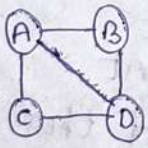
we can use heap data structure efficiently in system concerned with security & also embedded system such as linux, Kernel.

GRAPHS :-

* A graph is a non - linear data structure, it is a collection of vertices & edges.

* The edges connect any two vertices in the graph.

*



$G = (V, E)$

$V = (A, B, C, D)$

$E = \{(A,B), (A,C), (A,D), (B,D), (C,D)\}$

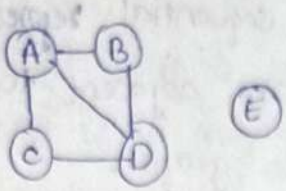
* Edges can be classified into 3 types :-

1) Undirected Edge : which has no specific direction
2) Directed Edge : which has some specific directions
3) Weighted Edge : which edges have some values those are called weighted edges.

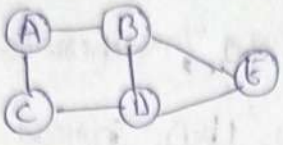
Types of Graphs :- (or) Terminologies of Graph :-

1) Isolated Node :- No adjacency or no link b/n nodes that node called as isolated node.

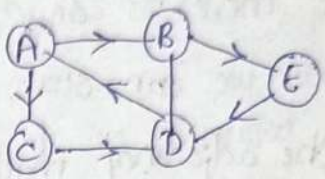
Ex:-



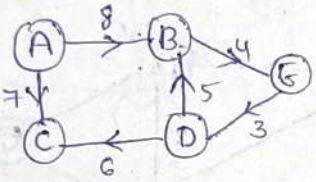
2) Undirected graph :- The order of two connected vertices is irrelevant and has no direction.



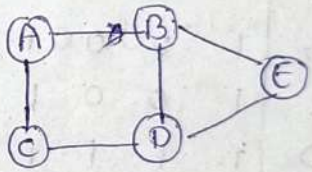
3) Directed Graph :- A directed graph also referred as a digraph, is a set of nodes connected by edges, each with a direction.



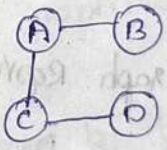
4) Weighted Graph :- In a graph each edge has a value of weight representing the cost of traversing the edge.



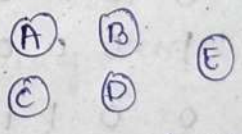
5) Cyclic Graph : If a graph contains atleast one graph cycle, it is considered by the cyclic.



6) Acyclic Graph : when there are no cycles in a graph. it is called an acyclic graph.



7) Null Graph : There are several vertices but no edges connecting them called a null graph.



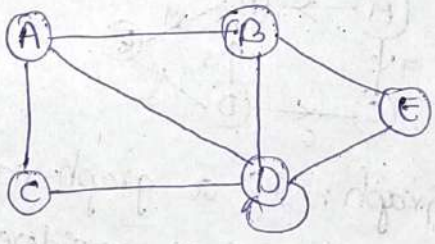
Graph Representation :- A graph representation is a technique to store graph data into the memory of a computer.
To represent a graph we just need the set of vertices & for each vertices the neighbours of the vertex.
The graph representation mainly classified into 3 types :-

1) Adjacency Matrix
2) Incidency Matrix &
3) Adjacency List

1) Adjacency Matrix:- Adjacency matrix is a sequential representation.
→ It is used to represent which nodes are adjacent to each other i.e., is their any edge connecting nodes to graph.
→ In this representation we have to construct a n x n matrix, if there is a any edge from one vertex to another vertex. then the corresponding element is "1", otherwise "0" (zero).
→ If there is any weighted graph then instead of one's & zeros we can store the weight of the edge.
→ The adjacency matrix representation can be classified into 3 types:- i) Directed
ii) undirected
iii) weighted

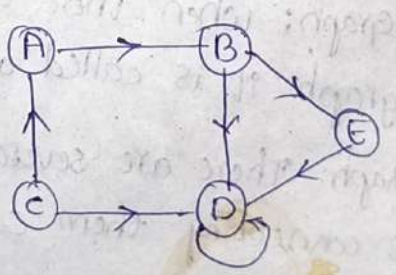
i) Directed Graph representation:-

	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

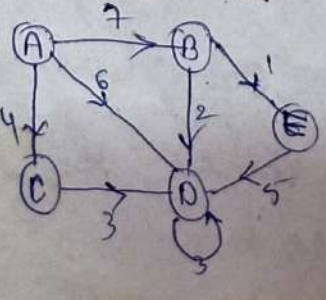


ii) Directed Graph Representation:-

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



iii) Weighted Graph representation:-



	A	B	C	D	E
A	0	7	4	6	0
B	0	0	0	2	1
C	0	0	0	3	0
D	0	0	0	3	0
E	0	0	0	5	0

2) Incidency Matrix :- In incidency matrix representation graph can be represented using a matrix size.

→ It means if a graph has four vertices & six edges then it can be represented using a matrix of 4×6. In this matrix columns represent edges and rows represent vertices.

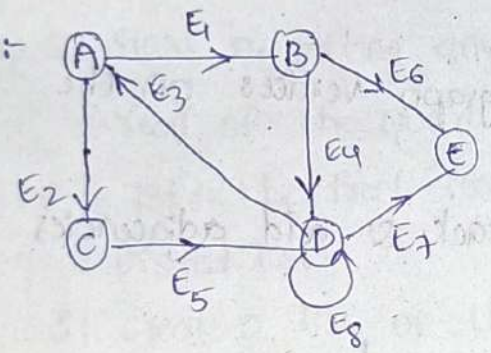
→ The matrix is filled with either '0' or '1' (or) '-1' where

i) '0' is used to represent row edge which is not connected to column vertex

ii) '1' is used to represent row edge which is connected as outgoing edge to column vertex.

iii) '-1' is used to represent row edge which is connected as incoming edge to column vertex.

Ex:-



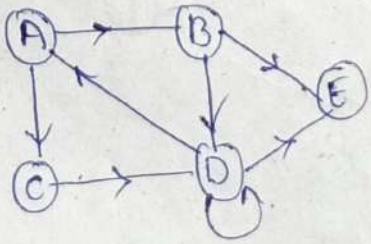
	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	-1
E	0	0	0	0	0	-1	-1	0

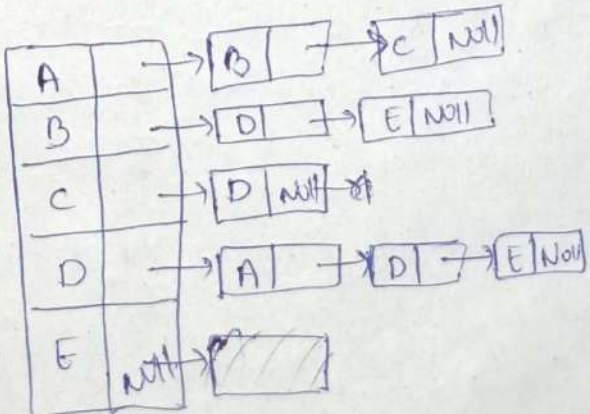
3) Adjacency List :-

→ Adjacency list is a linked representation.

→ In this representation for each vertex in the graph we maintain the list of its neighboors.

→ We have an array of vertices which is indexed by the vertex number for each vertex, the corresponding array element points to a single linked list of neighboors.





Graph Traversing :-

→ Graph Traversal is a Technique used for visiting all vertices atleast once in a graph.

→ It finds the edges to be used in the visiting process without creating loops.

→ Graph traversal techniques can be classified into two types :-

1) Depth first search (DFS)
2) Breadth first search (BFS)

1) Depth First Search :- It is a technique to visit all the vertices in a graph. by following the stack property i.e, LIFO.

Algorithm :-

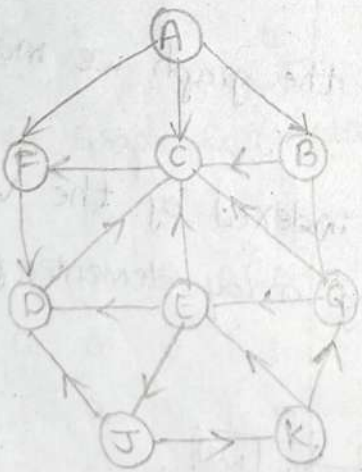
step 1: start by putting any of the graph vertices on the top of the stack.

2: Take the top item from the stack & add adjacencies of the item.

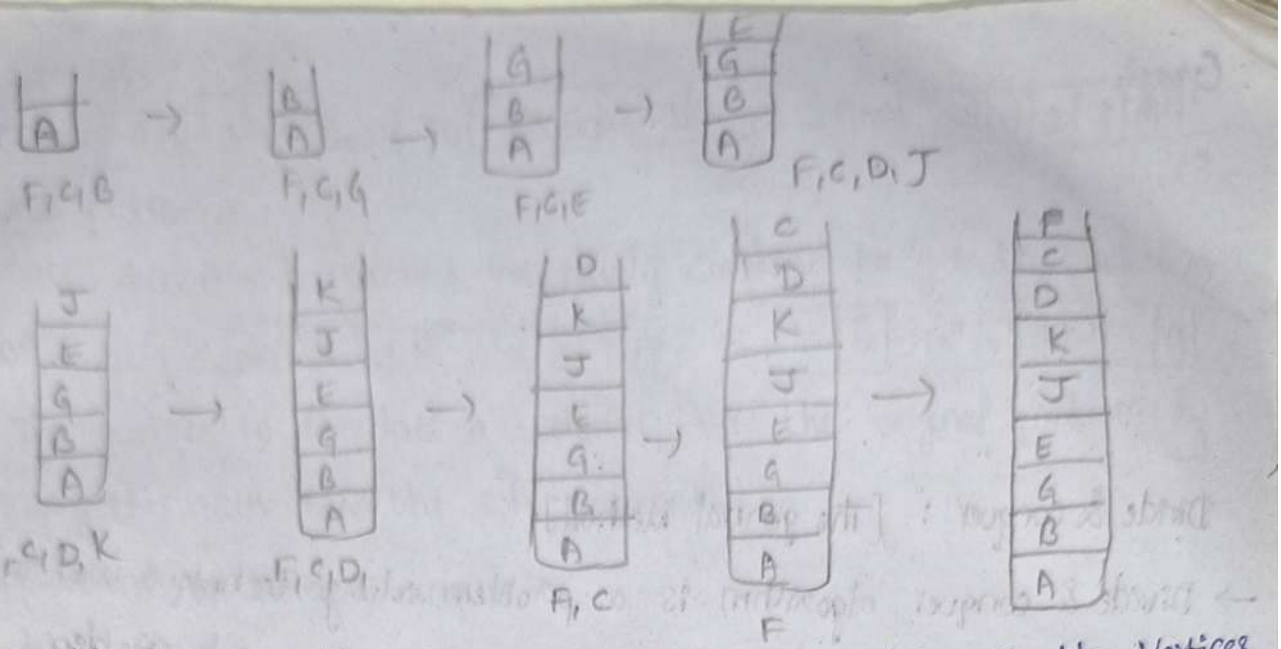
3: create a list of vertex adjacency nodes

4: Repeat step 2 & 3 until the stack is empty.

Ex :-



Vertices	Adjacencies
A	F, C, B
B	C, G
C	F, D
D	C, E, J
E	D
F	E, G
G	D, K
J	E, G
K	



2) Breadth first search:- It is a technique to visit all the vertices in a graph by following the Queue property. i.e, FIFO

Algorithm:-

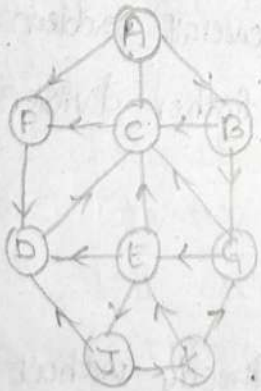
Step 1: Start by putting any one of the graph vertices of the rear of the queue.

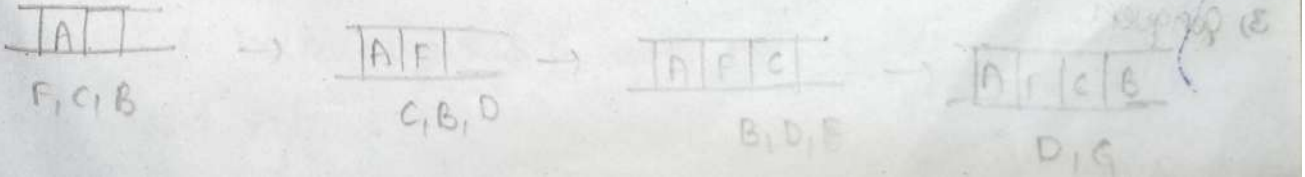
2: Take the front item of the queue & add it to the visited list.

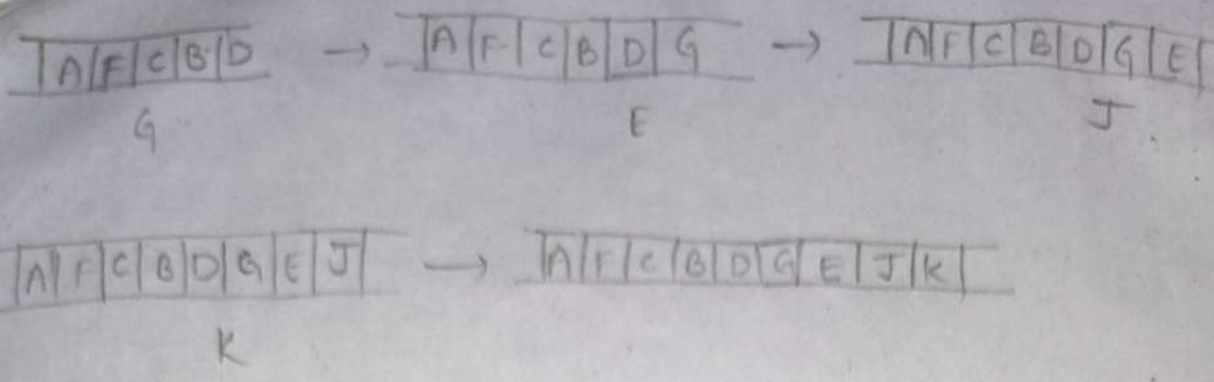
3: Create a list of that vertices adjacent nodes add the one which are its in the visited list to the rear of the queue.

4: Repeat 2 & 3 steps until Queue is empty.

Ex:-

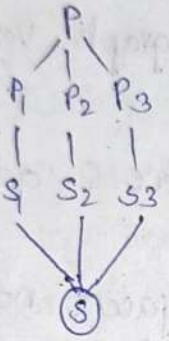






Divide & conquer : [The general Method]

→ Divide & conquer algorithm is a problem solving technique used to solve problems by dividing the main problem into sub problems, solving them individually & then to find solution to the original problem.



→ Divide & conquer algorithm can be divided into 3 steps:

1) Divide 2) conquer 3) combine/ Merge

1) Divide:

i) Breakdown original problem into smaller sub problems

ii) Each sub problem should represent a part of overall problem.

iii) The goal is to divide the problem until no further division is possible.

2) Conquer:

i) Solve each of smaller sub problems individually

ii) If sub Problem is small enough we solve it directly without further recursion.

iii) The goal is to find solutions for these sub problems independently

3) Conquer:

3) Combine:

i) Combine the subproblems of solutions to get final sol'n of the whole problem.

ii) once, smaller sub solutions recursively combine to get the solution of large problem.

iii) The goal is to formulate a solution for the original problem by merging the results from the sub problems.

Applications of Divide & Conquer:

Binary search, Merge sort, Quick sort, strasstens matrix multiplication

Quick Sort:-

Algorithm:-

→ Quick sort algorithm is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.

→ A large array is partitioned into 3 arrays. one of which holds values smaller than the specified Pivot Value and next part stores the pivot element and another array holds the values greater than the Pivot Value.

→ Quick sort partitions an array and then calls itself recursively twice to sort the resulting sub arrays.

Algorithm:-

Step 1: choose the first index value as Pivot

2: Take two variables to point left and right of the list excluding pivot.

3: Left points to the low index

4: Right points to the high index

5: while value at left is less than pivot move right

6: While value at right is greater than pivot move left

7: If both step 5 & 6 does not match, swap left and right.

8: If left ≥ right the point where they met is new pivot.

Ex: 12 4 21 30 25 17 5 8

[12] 4 21 30 25 17 5 8

P

4 > 12 ✗

21 > 12 ✓

8 < 12 ✓

l & r satisfies true then swap l, r positions

12 4 8 30 25 17 5 21

8 > 12 ✗

30 > 12 ✓

21 < 12 ✗

5 < 12 ✓

12 4 8 5 25 17 30 21

5 > 12 ✗

25 > 12 ✓

30 < 12 ✗

17 < 12 ✗

25 < 12 ✗

5 < 12 ✓

Note: whenever right position is closed to left position, then swap pivot to right position elements.

[5 4 8] [12] 25 17 30 21

[5] 4 8 5 4 8

P

4 > 5 ✗

8 > 5 ✓

8 < 5 ✗

4 < 5 ✓

4 5 8 12 [25] 17 30 21

[25] 17 30 21

P

17 > 25 ✗

30 > 25 ✓

21 < 25 ✓

25 17 21 30

21 > 25 ×
30 > 25 ✓

30 < 25 ×
21 < 25 ✓

21 17 25 30

17 > 21 ×
25 > 21 ✓

30 < 21 ×
25 < 21 ×
17 < 21 ✓

17 21 25 30

54 26 93 17 77 31 44 55 20

26 > 54 ×
93 > 54 ✓

20 < 54 ✓

54 26 20 17 77 3 44 55 93

20 > 54 ×
17 > 54 ×
77 > 54 ✓

93 < 54 ×
55 < 54 ×
44 < 54 ✓

54 26 20 17 44 31 77 55 93

44 > 54 ×
31 > 54 ×
77 > 54 ✓

77 < 54 × ✓
31 < 54 ✓

31 26 20 17 44 54 77 55 93

Merge Sort :-

Merge sort is a sorting Technique based on divide and conquer technique. Merge sort first divides the array into equal partitions and then combines them in sorted manner.

Algorithm :-

Step 1 : If it is only one element in the list, consider it already sorted, so return.

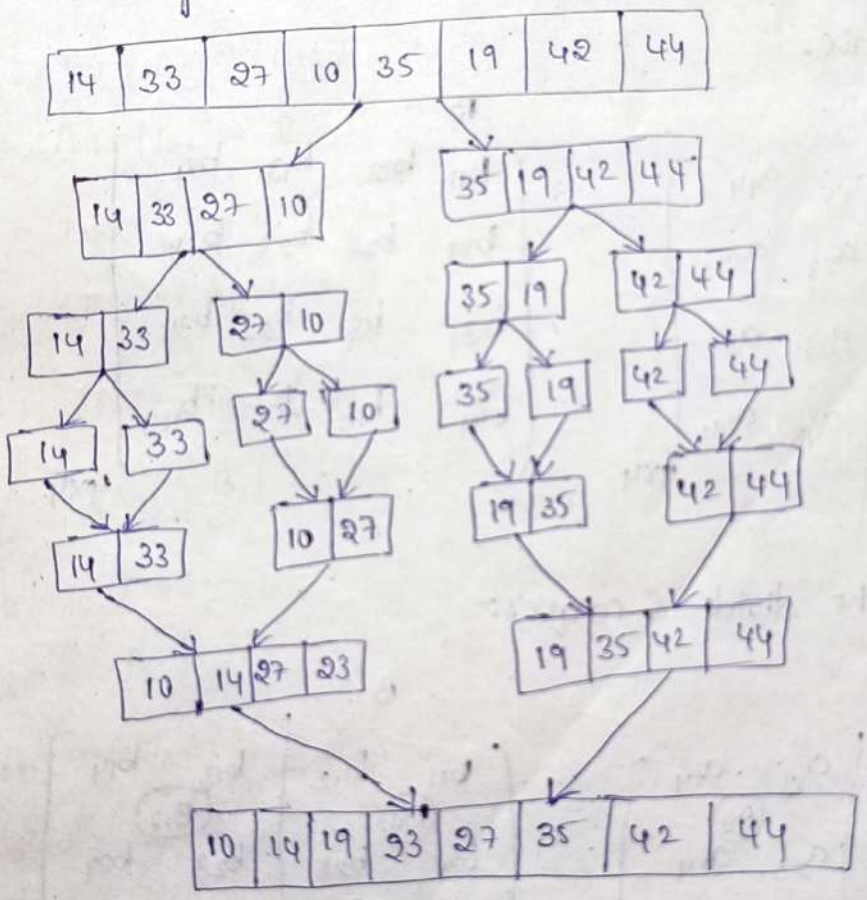
2 : Divide the list, recursively into two shelve until it can no more be divided.

3 : Merge the smaller list into new list in sorted order.

Ex:-

14	33	27	10	35	19	42	44

↓ unsorted array list.



Strassen's Matrix Multiplication :-

Matrix Multiplication has the 3 ways to calculate :

1) conventional or predictional matrix multiplication

2) Divide & conquer strategy

3) Strassen's Matrix multiplication

1) Conventional or predictional Matrix Multiplication:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_A \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}_B = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}_C$$

$$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21}$$

$$c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$$

$$c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$$

$$c_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$$

Note:- The time complexity of conventional matrix multiplication is $O(n^3)$

e. Divide & Conquer strategy of matrix:-

consider a 4X4 matrix, this 4X4 matrix divided into 4/2 * 4/2 matrix.

$$A_{4\times4} = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right]$$

$$B_{4\times4} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

By applying the divide & conquer:-

$$A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ (A_{11}) & & (A_{12}) & \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ (A_{21}) & & (A_{22}) & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \qquad B = \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ (B_{11}) & & (B_{12}) & \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ (B_{21}) & & (B_{22}) & \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right]$$

Note:- The time complexity of Divide & conquer strategy is $O(n^3)$

3) Strassen's Matrix Multiplication:-

→ It is the divide & conquer approach to solve the matrix multiplication problems.

→ By using Strassen's matrix multiplication algorithm the time consumption can be improved a little bit. The time complexity is $O(n^{2.81})$.

→ Strassen's matrix multiplication can be performed only on square matrices.

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

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = B_{22}(A_{11} + A_{12})$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (B_{21} + B_{22})(A_{12} - A_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

Here $A = \begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix}$, $B = \begin{bmatrix} 6 & 7 \\ 3 & 8 \end{bmatrix}$

$P = (1+5)(6+8) = (6)(14) = 84$

$Q = 6(7+5) = 6(13) = 78$

$R = 1(7-8) = (-1) = -1$

$S = 5(3-6) = 5(-3) = -15$

$T = 8(1+3) = 8(4) = 32$

$U = (7-1)(6+7) = (6)(13) = 78$

$V = (3+8)(3-5) = (11)(-2) = -22$

$C_{11} = P+S-T+V = 84+(-15)-32+(-22) = 15$

$C_{12} = R+T = -1+32 = 31$

$C_{21} = Q+S = 78-15 = 63$

$C_{22} = P+R-Q+U = 84-1-78+78 = 83$

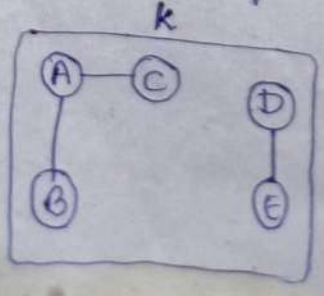
$$C = \begin{bmatrix} 15 & 31 \\ 63 & 83 \end{bmatrix}$$

Connected and Biconnected components:-

Connected components:-

A connected component is a sub graph in which any of one or two vertices are connected to each other by path, & which is connected to no additional vertices of the sub graph is called a connected component.

Ex:-



→ The above example had an undirected graph. In that graph we need to find the connected components.

→ By considering the above graph there are two different connected components they are: {B, A, C} and {D, E}

Algorithm :-

Step 1 : Initialize all vertices as not visited

2 : Do the following, for every vertex 'V'.

2.1 : If 'V' is not visited before call the DFS. and Print the new line character to print each component in a new line.

2.2 : Mark 'V' as visited and Print 'V'.

2.3 : For every adjacent of 'V', it is not visited then recursively call the DFS.

BiConnected Components :-

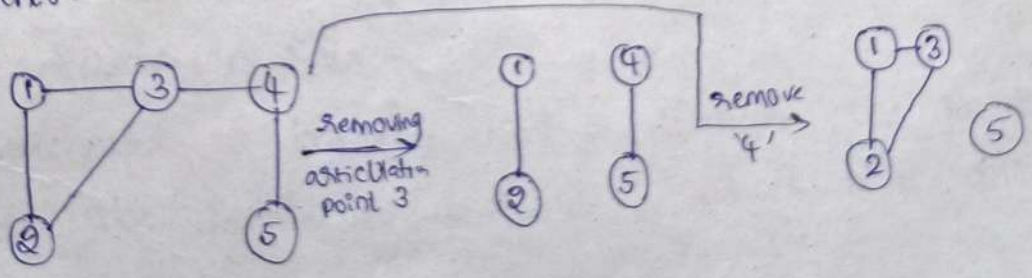
→ A biconnected undirected graph is a connected graph i.e, not broken into disconnected pieces by deleting any single vertex.

→ A biconnected directed graph is one such that for any two vertices and there are two directed paths from two vertices.

Articulation Point :-

It represents Vulnerabilities in a connected network — single point whose failure would split the network into two or more components.

Ex:-



By considering the graph Vertex '3' and '4' are articulation points. since the removal of vertex '3' or '4' along with its associated edges makes the graph disconnected.

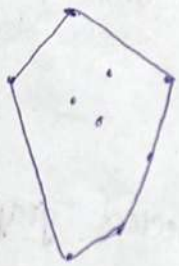
Convex Hull :-

Convex :-

A set of points (finite (or) infinite) in the plane is called Convex..

Convex Hull :-

* The convex hull is the smallest convex set that encloses all the points., forming a convex polygon.
* A convex polygon that encloses all the points, in two dimensions (2D), the convex hull is a convex polygon.
* In 3D it is a convex polyhedron.



convex hull.

* The convex hull is applied in various areas such as image processing, route planning & object modelling.
* Convex hull used by two types of algorithms :-
  1) Jarvis Algorithm
  2) Graham Scan Algorithm

UNIT-3

Greedy Method : General method, job sequence with Deadlines, Knap-Sack problems, Minimum cost spanning Trees, single source shortest Path.

Dynamic Programming : General method, All Pairs shortest Path, single source - shortest Path - General weights (Bell man ford algorithm, optimal binary search Trees, zero 0/1 Knap-sack, String editing, Travelling sales Person Problem).

Introduction of Greedy Method:-

⇒ The Greedy Method is one of the strategy like Divide and conquer used to solve the Problems.

⇒ This method is used for solving optimization Problems.

⇒ An optimization Problem demands either maximum or Minimum results.

⇒ The Greedy Method is the simplest & straight forward Technique.

⇒ The main function of this approach is that the decision is taken on the basis of the currently available information whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

⇒ This technique is basically used to determine the feasible solution, The feasible solution is a subset that satisfies the given criteria.

⇒ The optimal solution is the solution which is the best & the most favourable solution in the subset.

⇒ In the case of feasible if more than one solution will be considered as the feasible where as the optimal

solution is the best solution among all the solutions.

Applications of Greedy Method:-

1) General Method
2) job sequence with Deadlines
3) Knap-Sack Problems
4) Minimum cost spanning Trees
5) single source shortest Path.

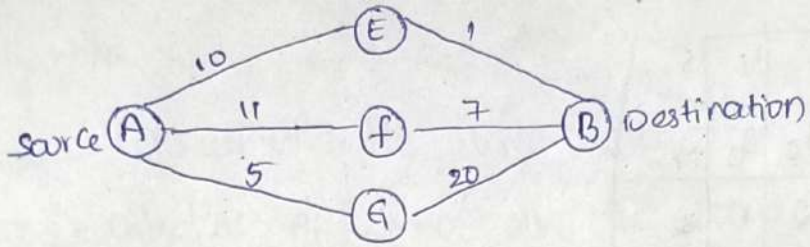
Algorithm:-

```
Algorithm Greedy (a,n)
{
 solution := 0;
 for i:=0 to n do
 {
  x: select (a);
 if feasible (solution, x)
 {
 solution := union (solution, x)
 }
 return solution;
 }
}
```

Initially the solution is assigned with zero value, we pass the array and number of elements in the greedy algorithm. Inside the for-loop we select the elements one by one & check whether the solution is feasible or not. If the solution is feasible, we perform the union.

Ex:- Travelling from 'A' to 'B'

solution: The problem is that we have to travel this journey from A to B. there are various solutions to go from 'A' to 'B' that solutions may be walk, Bike, car or Train



Job sequencing with Deadlines:-

⇒ Job scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits.

→ The greedy approach of the job scheduling algorithm states that given 'n' number of jobs with a starting time & ending time, they need to be scheduled in such a way that maximum profit is received with the maximum deadline.

Algorithm (Job scheduling Algorithm):-

Step 1: Find the maximum dead line 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 Profits, their time periods not exceeding the maximum deadline.

Step 4: The selected set of jobs are output.

Ex:-

S.No	1	2	3	4	5
Jobs	J_1	J_2	J_3	J_4	J_5
Deadline	2	2	1	3	4
Profit	20	60	40	100	80

Solution:-

1) The max deadline value from the given deadlines D = 4

2) Arranging the jobs in descending order of their profits

S.NO	1	2	3	4	5
Jobs	J_4	J_5	J_2	J_3	J_1
Deadlines	3	4	2	1	2
Profit	100	80	60	40	20

Assign slot	Job selected	Action	Profit
none	J_4	[2,3]	100
[2,3]	J_5	[3,4]	180+80
[2,3],[3,4]	J_2	[1,2]	180+60
[2,3],[3,4],[1,2]	J_3	[0,1]	240+40
[2,3],[3,4],[1,2],[0,1]	J_1	[1,2] ✗ Rejected	280

Acton:
d = (r-1, r)
= (3-1, 3)
= (2, 3)

Job sequence = { J_4, J_5, J_2, J_3 }

Profit = 280

Knap-Sack Problem :-

1) The Knap-sack Problem states that given a set of items, holding weights & profit values.

2) We must determine the subset of the items to be added in a Knap-sack such that the total weight of the items must not exceed the limit of the Knap-sack and its total profit value is maximum.

Knap-Sack Algorithm :-

→ The weights (w_i), profits (P_i) of the items to be added in the Knap-sack are taken as the input.

→ The items added in the Knap-sack without exceeding the limit and with maximum profit is achieved as the output.

Algorithm:-

Step 1: Consider all the items with their weights & Profits respectively.

Step 2: Calculate P_i/w_i of all the items and sort the items in descending order based on their P_i/w_i values.

Step 3: without exceeding the limit add the items into knap-sack

Step 4: If the knap-sack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.

Note:- The knap-sack problem is also known as fractional knap sack Problem.

EX:-

Items	1	2	3	4	5
weights	10	20	30	40	50
Profit	20	30	66	40	60

The knapsack size $(m) = 100$

$P_i/w_i = P_1/w_1 = \frac{20}{10} = 2$

$P_2/w_2 = \frac{30}{20} = 1.5$

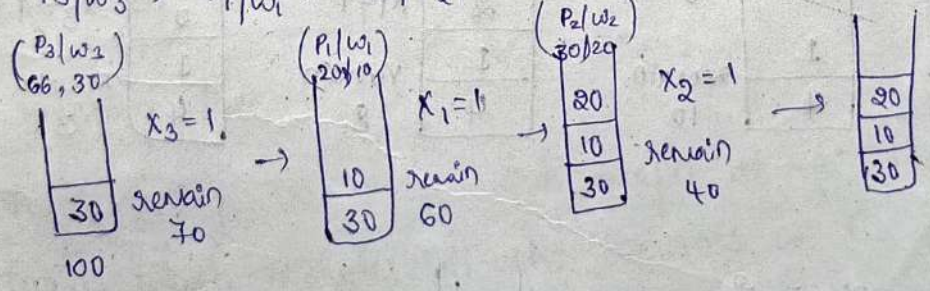
$P_3/w_3 = \frac{66}{30} = 2.2$

$P_4/w_4 = 40/40 = 1$

$P_5/w_5 = 60/50 = 1.2$

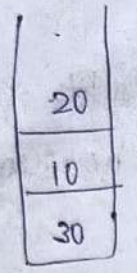
$0 \le x_i \le 1$

$P_3/w_3 > P_1/w_1 > P_2/w_2 > P_5/w_5 > P_4/w_4$



Note:- At the time of the item will be inserted into knap-sack that knap-sack size is overflow then we calculate $x_i =$

$$x_i = \frac{\text{Remaining size of knap-sack}}{\text{Actual weight of that object or item}}$$



$x_5 = \frac{40}{50} = \frac{4}{5}$

$= \frac{4}{5} \times 50$ [∵ acquired value × total weight of item]

$= 40$

$(P_4/w_4) \Rightarrow x_4 = 0$

Calculate the Profit $\sum P_i x_i = P_1 * x_1 + P_2 * x_2 + P_3 * x_3 + P_4 * x_4 + P_5 * x_5$

$= 20 * 1 + 30 * 1 + 66 * 1 + 40 * 0 + 60 * \frac{4}{5}$

$= 20 + 30 + 66 + 48 = 164$

Ex 2:-

Item	1	2	3	4	5	6	7
Profit	10	5	150	7	6	180	3
weight	2	3	5	7	1	4	1

Size $m = 15$

$P_i/w_i \Rightarrow P_1/w_1 = 10/2 = 5$

$P_2/w_2 = 5/3 = 1.6$

$P_3/w_3 = 150/5 = 30$

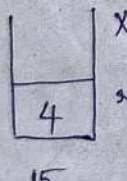
$P_4/w_4 = 7/7 = 1$

$P_5/w_5 = 6/1 = 6$

$P_6/w_6 = 180/4 = 20$ 4.5

$P_7/w_7 = 3/1 = 3$

$\therefore (P_6/w_6 > P_5/w_5 > P_1/w_1 > P_3/w_3 > P_7/w_7 \geq P_2/w_2 > P_4/w_4$



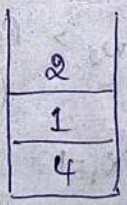
$x_6 = 1$

remain 11

15

$x_5 = 1$

remain 10



$x_1 = 1$

remain 8

$x_3 = 1$

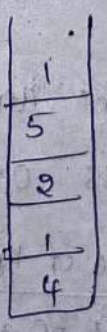
remain 3

$x_7 = 1$

remain 2

$x_2 = \frac{2}{3}$

$= \frac{2}{3} \times 3 = 2$



$(P_4/w_4) \rightarrow x_4 = 0$

)x

$X(\sum P_i x_i = P_1x_1 + P_2x_2 + P_3x_3 + P_4x_4 + P_5x_5 + P_6x_6 + P_7x_7$

$= 10 \times 1 + 5 \times \frac{2}{3} + 15 \times 1 + 7 \times 0 + 6 \times 1 + 80 \times 1 + 3 \times 1$

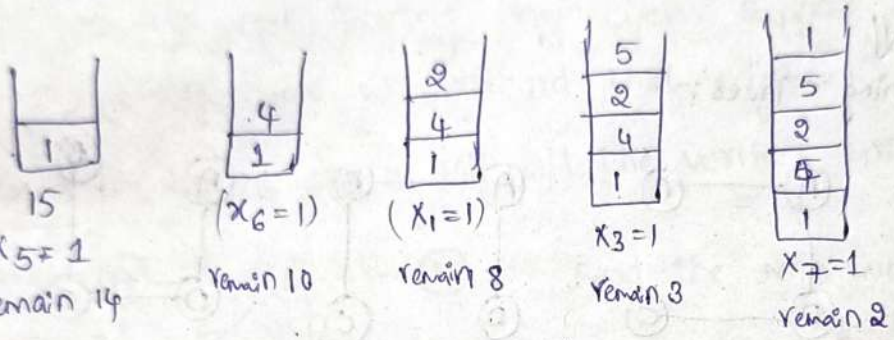
$= 10 + 5 \times \frac{2}{3} + 15 + 6 + 80 + 3$

$= 124) X$

$P_1/w_1 = 5, \quad P_2/w_2 = 1.6, \quad P_3/w_3 = 3, \quad P_4/w_4 = 1, \quad P_5/w_5 = 6$

$P_6/w_6 = 0.5 \quad P_7/w_7 = 3$

$P_5/w_5 > P_6/w_6 > P_1/w_1 > P_3/w_3 > P_7/w_7 \geq P_2/w_2 > P_4/w_4$



$x_2 = \frac{2}{3} = \frac{2}{3} \times 3 = 2$, $(P_4/w_4)\ x_4 = 0$

$\sum P_i x_i = 10 \times 1 + 5 \times \frac{2}{3} + 15 \times 1 + 7 \times 0 + 6 \times 1 + 18 \times 1 + 3 \times 1$

$= 10 + 3.3 + 15 + 0 + 6 + 18 * 3$

$= 55.3$

Minimum Cost Spanning Trees :-

Spanning Tree :-

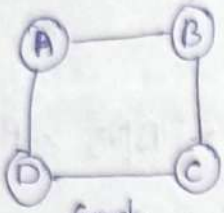
* A spanning Tree is a sub graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible no. of edges. If a vertex is missed then it is not a spanning Tree.
* A spanning Tree Must satisfy the three properties:-

1) It should contain all the vertices of a graph

2) If graph contains 'n' edges then spanning Tree contains "n-1" edges.

3) Spanning Tree should not contain any cyclic form.

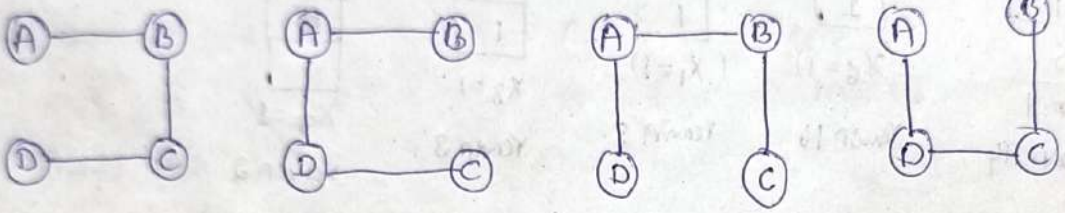
Eg:-



Graph

⇓

Possible spanning Trees:-

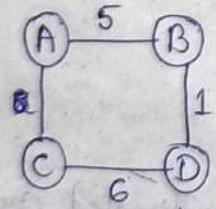


Minimum cost spanning Trees:-

→ The spanning Tree in which the cost is minimum, here the cost is also known as "weight".

→ A minimum spanning Tree is a spanning Tree in which the sum of the weight of edges as minimum as possible.

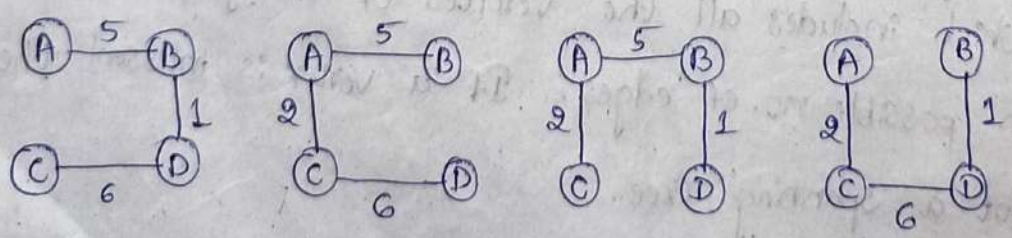
Eg:-



⇒ Graph with weights

⇓

Possible spanning trees:-



= 12 = 13 = 8 = 9

⇓

Minimum cost spanning tree = 8

→ In order to construct the minimum cost spanning Tree we're following the two algorithms :-

1) Prims Algorithm
2) Kruskal's Algorithm

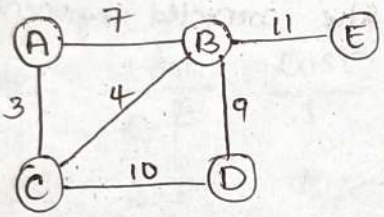
1) Prims Algorithm :-

Step 1 :- Start with any vertex of the given graph.

Step 2 :- Find the edges associated with vertex and add minimum cost edge to the spanning Tree, if it is not forming any cycle. suppose, it is forming any cycle we discard that edge.

Step 3 :- Repeat step 2 till all the vertices are covered.

Eg :-



Find the minimum cost spanning Tree.

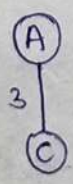
Solution :-

→ The given graph contains five vertices and six edges.

→ The spanning Tree should also contain five vertices and (6-1) means 5 edges.

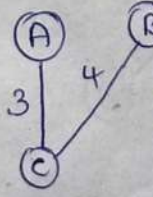
→ Step 1 :- (A) A-B(7), A-C(3)

Step 2 :-



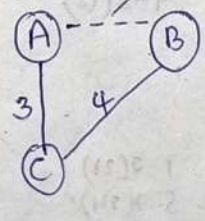
A-B(7)
C-D(10)
C-B(4)

Step 3 :-



A-B(7)
B-D(9)
C-D(10)
B-E(11)

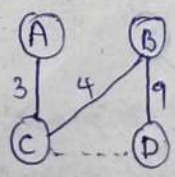
Step 4 :-



Here we're getting cyclic form so this edge is discarded

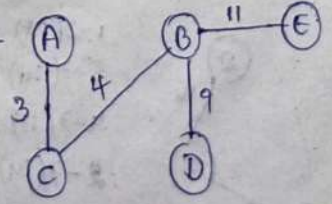
~~A-B(7)~~
B-D(9)
E-D(10)
B-E(11)

Step 5 :-



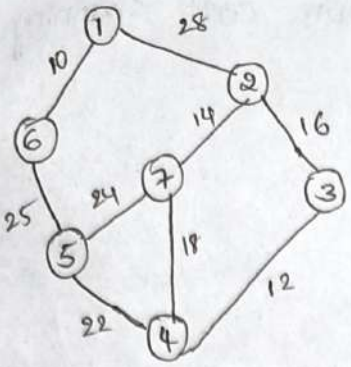
~~C-D(10)~~
B-E(11)

Step 6 :-



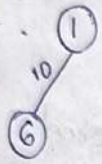
= 3+4+9+11 = 27

Eg:-



Sol:- Step:- (1) 1-6 (10)
1-2 (28)

Step (2):-

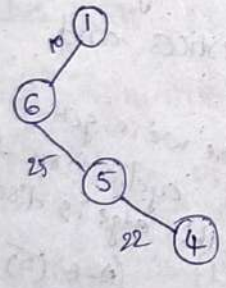


1-2 (28)
6-5 (25)

3)

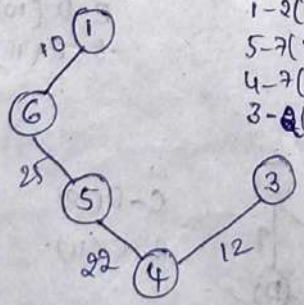
1-2 (28)
5-7 (24)
5-4 (22)

4)



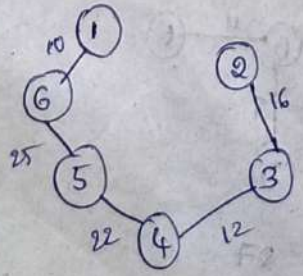
1-2 (28)
5-7 (24)
4-7 (18)
4-3 (12)

5)



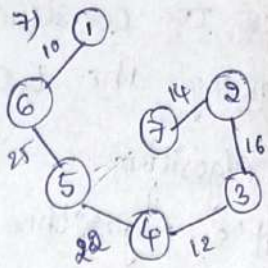
1-2 (28)
5-7 (24)
4-7 (18)
3-2 (16)

6)



1-2 (28)
5-7 (24)
4-7 (18)
2-7 (14)

7)



~~1-2 (28)~~
~~5-7 (24)~~
~~4-7 (18)~~

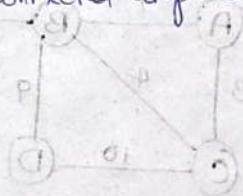
10 + 25 + 22 + 12 + 16 + 14

= 99

minimum cost spanning tree for

Note:- We can't solve connected components graph by using Prims algorithm.

By using Kruskal's algorithm we can solve connected components graph.



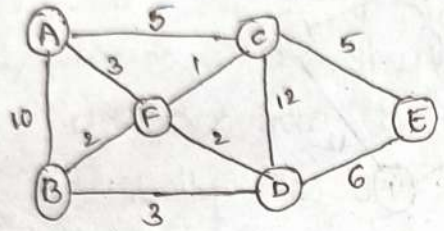
Kruskal's Algorithm:-

Step 1: We have to arrange all the nodes in ascending order based upon their cost

Step 2: Select the minimum cost edge list and add that edge to the spanning tree, if it is not forming any cycle, suppose by adding that edge if that is forming in a cycle then discard that edge.

Step 3: Repeat step 2 until all vertices are covered

Ex:-

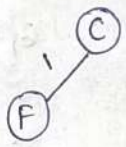


Solution:

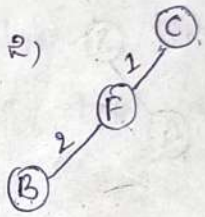
1)

SNo.	Edge	Cost
1.	C-F	1
2.	B-F	2
3.	D-F	2
4.	A-F	3
5.	B-D	3
6.	A-C	5
7.	C-E	5
8.	D-E	6
9.	A-B	10
10.	C-D	12

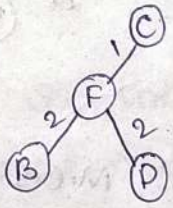
1)



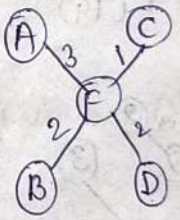
2)



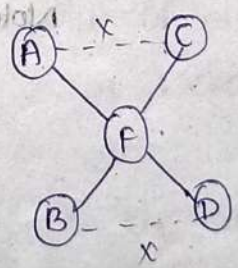
3)



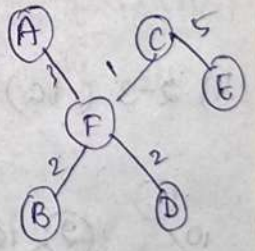
4)



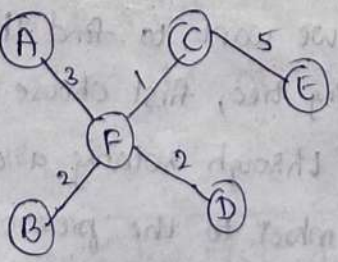
5)



6)



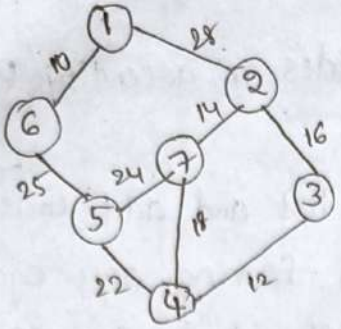
Final Tree →



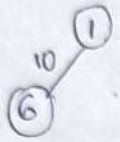
Minimum Cost Spanning Tree = 1+3+2+2+5

= 13

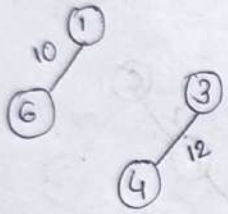
Eg 2).



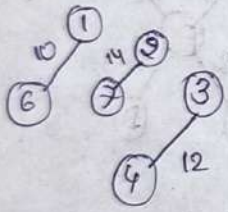
Step 1: Min. cost edge (1-6)



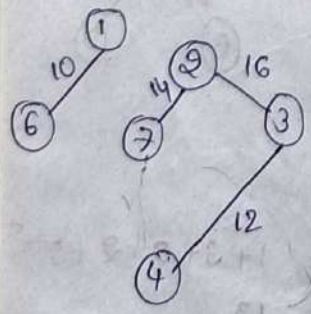
2) 3-4 (12)



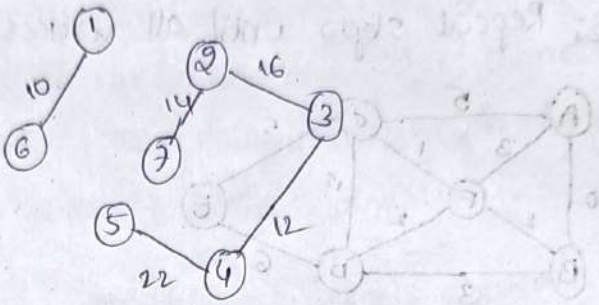
3) 2-7 (14)



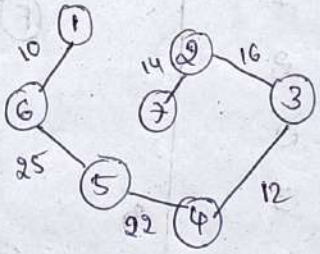
4) 2-3 (16)



5) 5-4 (22)

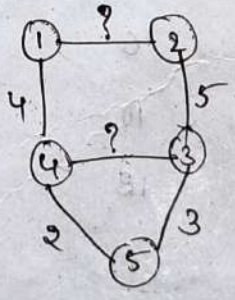


6) 5-6 (25)

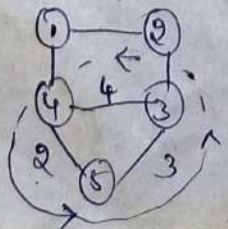


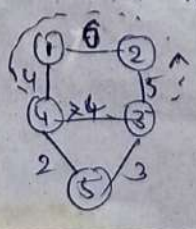
M.C.S.T = 99

Note:-



Whenever we want to find the missing value of spanning tree, first choose edge with lowest cost and through rotating a circle write next number to the previous one.





Single Source shortest Path :-

Given a weighted and Directed graph and a source vertex in the graph, find the shortest paths from the source to all the other vertices in the given graph.

Note :-

The Given graph doesn't contain any negative edges.

=) If we want to find the single source shortest path by using Dijkstra's Algorithm.

Step 1 :- Create a set shortest path tree set i.e., whose minimum distance from the source is calculated and finalised. Initially this set is empty.

Step 2 :- Assign a distance value to all vertices in the input graph. Initialise all distance values as infinite. Assign the distance value as 'zero' for the source vertex.

Step 3 :- While shortest path tree set doesn't include all vertices

3.1 :- Pick a vertex that is not there in shortest path tree set and has a minimum distance value.

3.2 :- Include that to shortest path tree set.

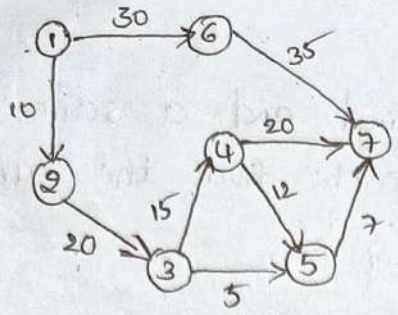
3.3 :- Then update the distance value of all adjacent vertices.

3.3.1 :- To update the distance values iterate through all the vertices.

3.3.2 :- For every adjacent vertex, if the sum of the distance value of source and weight of edge is less than the distance value then update the distance value.

$$dist[w] = \min\{dist[w], dist[v] + cost[v,w]\}$$

Eg:-



Solution:- Consider (1)

Let us "one" as source vertex, based upon the source vertex we need to find single source shortest path of all vertices by using Dijkstra's Algorithm.

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)	65→(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	
2	10
3	30
4	45
5	35
6	30
7	42

Dynamic Programming – General Method:-

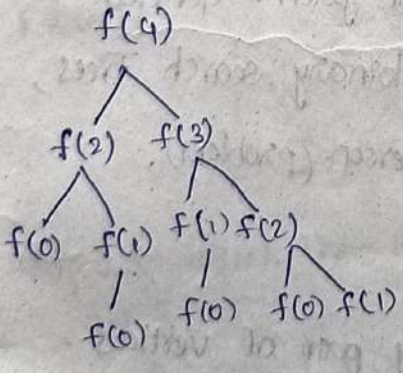
* Dynamic Programming is a method used to solve complex problems by breaking them down into simple sub problems.
* By solving each sub problem only once and stores the result, it avoids redundant computations, leading to more efficient

How does Dynamic Programming work:-

1) Identify sub problems: Divide the main problem into smaller, independent sub problems.
2) Store solutions: Solve each sub problem and store the solution in a table.
3) Buildup solutions: Use the stored solutions to build up the solution to the main problem.
4) Avoid Redundancy:- By storing solutions dynamic Programming ensures that each sub problem is solved only once, reducing computation time.

Example of Dynamic Programming: fibonacci sequence [0 1 1 2 3 5 8 13]

$$f(n) = f(n-1) + f(n-2)$$



When to use dynamic Programming:-

Dynamic Programming is an optimization technique used when solving problems that consist of the following characteristics

1) optimal sub structure: It means that we combine the optimal results of a sub problem to achieve the optimal

result of a bigger problem.

Ex:- finding the minimum cost path in a weighted graph.

e) Overlapping sub problems :- The same sub problems are solved repeatedly in different parts of the problem.

Ex:- Consider the problem of computing the fibonacci series

3) Approaches of Dynamic Programming :-

i) Top-Down Approach (Memoization) :-

In the Top-Down Approach also known as Memoization, we start the final solution and recursively break it down into smaller sub problems, this stores the results of solved sub problems in a memory.

ii) Bottom-up Approach (Tabulation) :-

In the Bottom-up Approach also known as Tabulation, we start with the smallest sub problems and gradually build up to the final solution, we store the results of solved sub problems in a table.

Applications of Dynamic programming :- general method, All pairs shortest path, single source shortest path - General Weights (Bellman Ford algorithm), optimal binary search Trees, 0/1 knapsack, string editing, Travelling sales person (problem).

All pairs of shortest Path Problem :-

→ We have to find out shortest path b/n every pair of vertices.

→ By using floyd war "FLOYD WARSHALL" Algorithm we can find the all pairs of shortest path.

FLOYD WARSHALL Algorithm :-

* The floyd warshall algorithm is a graph algorithm that is deployed to find the shortest path b/n all the vertices present in a weighted graph.
* This doesn't allow negative values
* It can't work on negative cycle graphs.

Algorithm :-

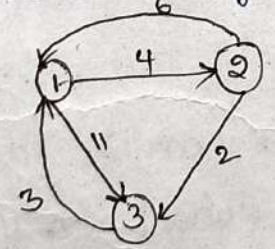
Step 1: construct an adjacency matrix 'A' with all the cost of edges present in the graph. If there is no path b/n the two vertices mark the value as infinity (∞).

Step 2: Derive another adjacency matrix 'A1' from 'A' keeping the first row & first column of the original adjacency matrix in A^1.

Step 3: Repeat step 2 for all the vertices in the graph by changing the A value for every vertex until the final matrix is achieved.

Step 4: The final adjacency matrix obtained is the final solution with all the shortest paths.

EX:-



Step 1: frame the Adjacency matrix A^0 for the given graph

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

Step 2: calculate the matrix A^1 from the matrix A^0

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

$$A' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & \\ 3 & & 0 \end{bmatrix} \end{array}$$

⇒ Keeping the first row & first column constant.

$A'[2,3] = ?$ $A'[3,2] = ?$

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

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

$(k=1, i=2, j=3)$

$= \min\{2, 6+11\}$

$= \min\{2, 17\}$

$A^1[2,3] = 2$

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

$= \min\{\infty, 3+4\} = \min\{\infty, 7\}$

$A'[3,2] = 7$

$$\therefore \text{matrix } A' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

Step3: calculate matrix A^2.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & \\ 6 & 0 & 2 \\ & 7 & 0 \end{bmatrix} \end{array}$$

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

$= \min\{11, 4+2\} = \min\{11, 6\} = 6$

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

$= \min\{3, 7+6\} = \min\{3, 13\} = 3$

$$\therefore A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

Step 4: calculate matrix 3

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

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

$= \min\{4, 6+7\} = \min\{4, 13\} = 4$

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

$= \min\{6, 2+3\} = \min\{6, 3\} = 3$

$$\therefore A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 3 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

Single Source shortest Path – General Weights

Bellman Ford Algorithm :-

→ The Bellman Ford Algorithm is the best suited to find the shortest paths in a directed graph, with one or more negative edge weights from the source vertex to all other vertices.

→ Using the Bellman Ford Algorithm on a graph with negative cycles will not produce a result of shortest paths.

Algorithm :-

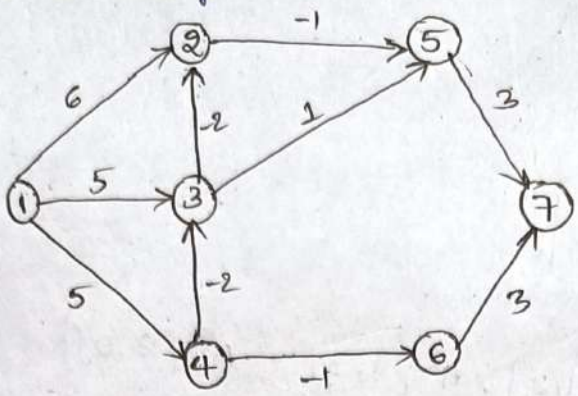
Step 1 : Set the initial distance to zero for the source vertex and set initial distances to infinity for all other vertices.

Step 2 : For each edge check if the shorter distance can be calculated & update the distance, if the calculated distance is shorter.

Step3: check all edges (step 2) "V−1" (V= Vertices) times.

This is as many times as there are vertices 'V', '-1'

Ex:-



$dist^{K}[1:7]$

K= no. of edges

distance = weight or cost

(1:7) = 1 to 7 vertices

edges = vertices − 1

source vertex = 1

1	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

0/1 Knap sack Problem by using dynamic Programming :-

Problem: given 'n' items where each item has their own weight and Profits i.e, Wi, Pi knapsack can hold almost weight 'm'.

n=3, M=6 → knapsack size

↓ (no. of objects)

Draw a table

Object	1	2	3
Profit	1	2	5
Weight	2	3	4

Sol: i) The solution is represented by knapsack instance or solution vector (Xi)

iii) by solving 0/1 knapsack problem we need to calculate S^0, S^1, S^2, S^3

iii) S^0 is the initial set, the initial values are (0,0) i.e., $S^0 = (0,0)$

iv) We find the remaining sets S^1, S^2, S^3 by calculating the (Profit, weight) sets we are following the formula $\boxed{S^{i+1} = S^i \cup S_i^i}$

Let $i=0 \quad S^{i+1} = S^i \cup S_i^i \Rightarrow S^{0+1} = S^0 \cup S_1^0$

$S^1 = S^0 \cup S_1^0$

We find S_1^0 means first object Profit as well as weight will be added to the existing set values

$S_1^0 = \{0+1, 0+2\} = \{1,2\}$ | $\therefore S^1 = S^0 \cup S_1^0$

$S^1 = \{\{0,0\} \cup \{1,2\}\} = \{(0,0)(1,2)\}$

Note: If any set got two or more pairs, then we need to check it will follow DOMINEN'S (or) PURGING rule or not.

DOMINEN'S RULE :-

If we have 2 pairs $\{(P_1, W_1), (P_2, W_2)\}$ then if $\{P_1 \leq P_2\}$ must and should $\{W_1 \leq W_2\}$. Hence, if $\{P_1 \leq P_2\}$ otherwise, $\{W_1 \geq W_2\}$ then dominen's pair P_1, W_1 will be purging (remove)

Calculate S_2^1 : $\boxed{S^2 = S^1 \cup S_1^1}$

S_1^1 = Add 2nd object Profit & weight to all existing set of values

$= \{(0+2), (0+3), (1+2), (2+3)\} = \{(2,3), (3,5)\}$

Now $S^2 = S^1 \cup S_1^1$

$= \{(0,0)(1,2) \cup (2,3)(3,5)\} \Rightarrow \{(0,0)(1,2)(2,3)(3,5)\}$

Now, we have to check whether it follows Dominen's rule or not & also over weight

$= \{(0,0)(1,2)(2,3)(3,5)\}$

Calculate $S^3 \Rightarrow S^2 \cup S_1^2$

S_1^2 = Add 3rd object Profit & weight

$= \{(0+5)(0+4), (1+5), (2+4), (2+5)(3+4), (3+5)(5+4)\}$

$= \{(5,4), (6,6), (7,7), (8,9)\}$

$= \{(5,4)(6,6)\}$

$S^3 = S^2 \cup S_1^2 = \{(0,0)(1,2)(2,3)(3,5) \cup (5,4)(6,6)\}$

Dominen's pair

check dominen's rule, consider (3,5)(5,4). Then remove (3,5) pair

$\therefore S^3 = \{(0,0)(1,2)(2,3)(5,4)(6,6)\}$

→ Select one pair in that pair, weight value is near to 6

I. $(6,6) \in S^3$

$(6,6) \notin S^2$

Note:-

$(6,6) \notin S^2$ then $x_3 = 1$

then (6,6) subtract to 3rd object Profit & weight

$= (6-5, 6-4) = (1,2)$

II. $(1,2) \in S^2$ } both $\in$ to $S_1^2 S^1$

$(1,2) \in S^1$ } $\therefore x_2 = 0$

III. $(1,2) \in S^1$ } $x_1 = 1$

$(1,2) \notin S^0$ }

$\therefore$ x_1 x_2 x_3

1 0 1

2. object A B C D

Profit 2 4 7 10 (m=8)

Weight 1 3 5 7

Travelling sales Person problem by dynamic Programming:

1) Travelling is sales Person is most computational problem

2) By using dynamic Programming approach, to evaluate every possible tour & select best one.

3) Using dynamic Programming approach, it will obtain solution in lesser time.

4) In this problem, we have to start at a starting vertex and we need to visit the remaining vertices exactly one and return back to starting vertex with minimum cost.

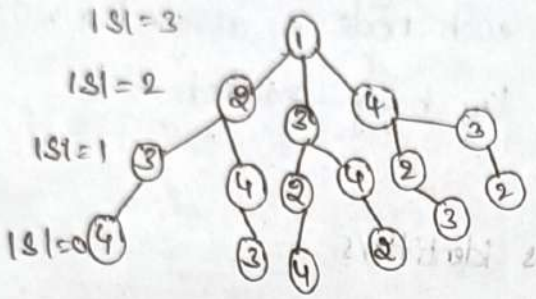
	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Based on adjacency matrix, we need to solve Travelling sales Person Problem.

Sol: $\text{Cost}[i,j] = \begin{cases} 0 & \text{if } (i=j) \\ c_{ij} & \text{if } (i,j) \in q \\ \infty & \text{if } (i,j) \notin q \end{cases}$

$$g(i,s) = \min_{j \in s} \{ c_{ij} + g(j, \{s\} - j) \}$$

i = starting vertex, j = set of remaining vertices, c_{ij} = cost of vertices



$g(i,s) = \min_{j \in s} \{ c_{ij} + g(j, \{s\} - j) \}$

$|S| = 0$ → 1st column

$g(2,\phi) = 5$

$g(3,\phi) = 6$

$g(4,\phi) = 8$

$|S| = 1$ $\quad g(i,s) = \min_{j \in s} \{ c_{ij} + g(j, \{s\} - j) \}$

$g(2,\{3\})$, $g(2,\{4\})$, $g(3,\{2\})$, $g(3,\{4\})$, $g(4,\{2\})$, $g(4,\{3\})$

$g(2,\{3\}) = \min \{ 9 + g(3,\{3\} - 3) \}$

$= \min \{ 9 + g(3,\phi) \} \Rightarrow 9+6 = 15$

$g(2,\{4\}) = \min \{ 10 + g(4,\{4\} - 4) \}$

$= \min \{ 10 + (4,\phi) \} = 10+8 = 18$

$g(3,\{2\}) = \min \{ 13 + g(2,\{2\} - 2) \}$

$= \min \{ 13 + (2,\phi) \} = 13+5 = 18$

$g(3,\{4\}) = \min \{ 12 + g(4,\{4\} - 4) \}$

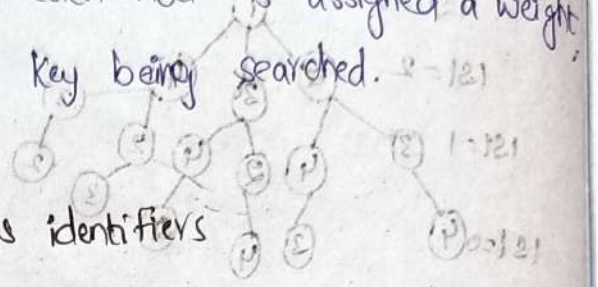
$= \min \{ 1$

$= 12+8 = 20$

Optimal Binary Search Tree:-

→ An optimal binary search Tree is also known as a weighted binary search Tree, is a binary search tree that minimizes the expected search cost. In a binary search Tree, the search cost is the no. of comparisons required to search for a given key.

→ In an optimal Binary search Tree, each node is assigned a weight that represents the probability of the key being searched.



Example:-

Let n=4 (nodes) nodes also known as identifiers

$(a_1, a_2, a_3, a_4) = (do, if, int, while)$

$P(1:4) = (3, 3, 1, 1)$

$q(0:4) = (2, 3, 1, 1, 1)$ Compute and construct OBST for above values using Dynamic approach.

where 'p' is known as Probability of successful search

'q' " " " " Unsuccessful " ?

Sol:-

$$w(i, j) = w(i, j-1) + P(j) + q(j)$$

$$c(i, j) = \min_{i < k \le j} \{ c(i, k-1) + c(k, j) + w(i, j) \}$$

$$r(i, j) = k$$

Initial conditions $w(i, i) = q_i$

$c(i, i) = 0$

$r(i, i) = 0$

$i = 0$

$w(0,0) =$

$w(i,j) = w(i,j-1) + P(j) + q(j)$

$c(i,j) = \min_{i<k\le j} \{ c(i,k-1) + c(k,j) + w(i,j) \}$

$r(i,j) = k$

$j = 1$

$r(i,j) = k$

$r(i,k-1)$ $r(k,j)$

	i=0	i=1	i=2	i=3	i=4
j-i=0	$W_{00}=2$ $C_{00}=0$ $\gamma_{00}=0$	$W_{11}=3$ $C_{11}=0$ $\gamma_{11}=0$	$W_{22}=1$ $C_{22}=0$ $\gamma_{22}=0$	$W_{33}=1$ $C_{33}=0$ $\gamma_{33}=0$	$W_{44}=1$ $C_{44}=0$ $\gamma_{44}=0$
j-i=1	$W_{01}=8$ $C_{01}=8$ $\gamma_{01}=1$	$W_{12}=7$ $C_{12}=7$ $\gamma_{12}=2$	$W_{23}=3$ $C_{23}=3$ $\gamma_{23}=3$	$W_{34}=3$ $C_{34}=3$ $\gamma_{34}=4$	
j-i=2	$W_{02}=19$ $C_{02}=19$ $\gamma_{02}=1$	$W_{13}=9$ $C_{13}=12$ $\gamma_{13}=2$	$W_{24}=5$ $C_{24}=8$ $\gamma_{24}=3$		
j-i=3	$W_{03}=14$ $C_{03}=25$ $\gamma_{03}=2$	$W_{14}=11$ $C_{14}=19$ $\gamma_{14}=2$			
j-i=4	$W_{04}=16$ $C_{04}=32$ $\gamma_{04}=2$				

$W_{12} = w(1,1)+P(2)+q(2)$

$= 3+3+1 = 7$

$C_{12} = \min_{1<k\le 2}\{c(1,1)+c(2,2)+w(1,2)\}$ (1,2)

$= \{0+0+7\} = 7$

$\gamma_{12} = 2$

$W_{23} = w(2,2)+P(3)+q(3)$

$= 1+1+1 = 3$

$W_{34} = w(3,3)+P(4)+q(4)$

$= 1+1+1 = 3$

$C_{34} = \min_{3<k\le 4}\{c(3,3)+c(4,4)+w(3,4)$

$= 0+0+3 = 3$

$= c[0,2)+c(3,3)+w(0,3)$

$= 19+0+14 = 33$

$C_{03} = \min_{0<k\le 3}\{c(0,0)+c(1,3)+w(0,3)$ $i=0$ $j=3$ $k=1,2,3$

$= 0+12+14 = 26$

$= \{c(0,1)+c(2,3)+w(0,2)$

$= 8+3+14$

$W_{04} = w(0,3)+P(4)+q(4) = 25$

$= w(14+1+1 = 16$

$W_{02} = w(0,1)+P(2)+q(2)$

$= 8+3+1 = 12$

$C_{02} = \min_{0<k\le 2}\{c(0,0)+c(1,2)+$ $i=0$ $j=2$

$w(0,2)$

$= 0+7+14 = 21$

$\min_{0\le k\le 2}\{c(0,1)+c(2,2)+$

$i=0\ j=2, k=2$ $w(0,2)$

$= 8+0+12 = 20$

$W_{13} = w(1,2)+P(3)+q(3)$

$= 7+1+1 = 9$

$W_{24} = w(2,3)+P(4)+q(4)$

$= 3+1+1 = 5$

$W_{03} = w(0,2)+P(3)+q(3)$

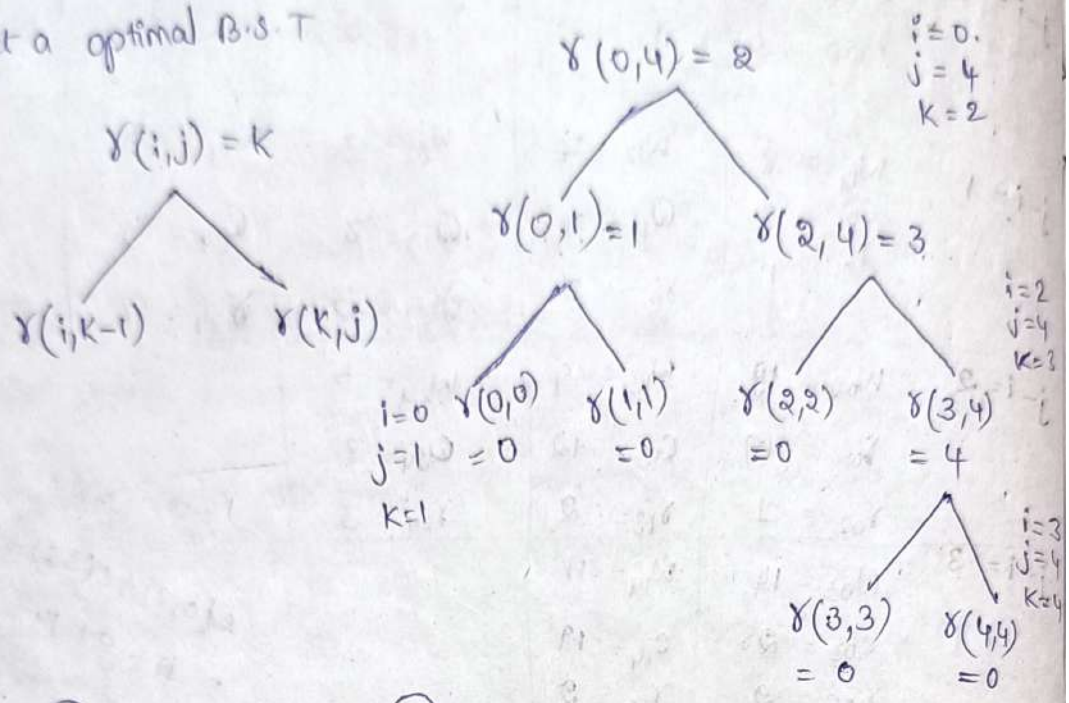
$= 12+1+1 = 14$

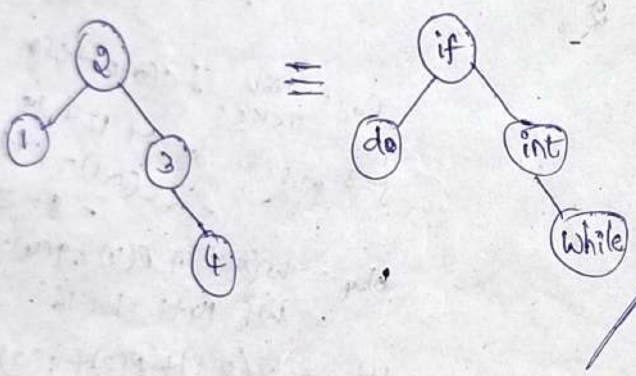
$W_{14} = w(1,3)+P(4)+q(4)$

$= 9+1+1 = 11$

By using the above table values we may construct the optimal binary search tree of T_{0n} T_{04}

To construct a optimal B.S.T





String Editing :-

→ There are two strings given, the first string is the source string & the second string is the target string. In this program we have to find how many possible edits or operations are needed to convert the first string to the second string.

→ The edit or operations of strings can be inserted, remove and replace / change.

Insert :- Insert any character before / after any index.

Remove :- Remove a character of any index.

Replace :- Replace a character at any index with some other character.

Note:-

All of the above operations are equal cost.

Example:-

string 1 = Sunday

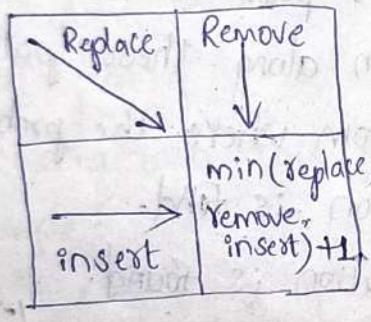
string 2 = Saturday

By observing the above two strings first & last three characters are same, we basically need to convert "UN" to "Atur" these can be done using strin 3 operations replace 'N' with 'r' & insert 't', 'a'

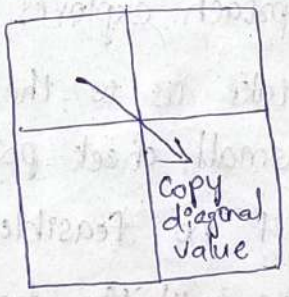
Example:-

i = row
j = column

if $i \neq j$



if $i = j$



Symbols for representing the 3 operations:

1, Insert : →

2, Replace : ↘

3, Remove : ↓

Example 1:-

string 1 = "a d c e g"

string 2 = "a b c f g"

To perform string editing operation by using Dynamic Programming

	Null	a	b	c	f	g
Null	0	1	2	3	4	5
a	1	0	1	2	3	4
d	2	1	1	2	3	4
c	3	2	2	1	2	3
e	4	3	3	2	2	3
g	5	4	4	3	3	2 → Result

UNIT – 4

Back Tracking :- General method, 8 queens Problem, Sum of subset problem, graph coloring, zero/one (0/1) knapsack Problem.

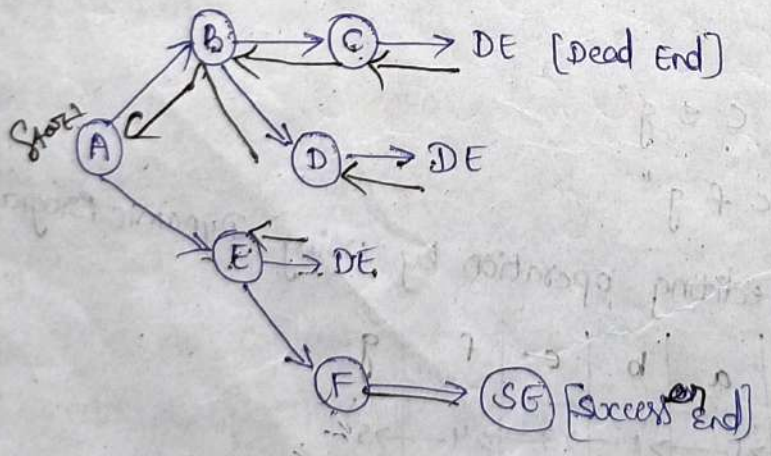
Branch & Bound: The General Method, 0/1 knapsack problem, travelling sales Person Problem.

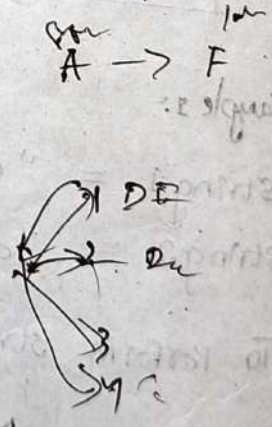
Back Tracking :-

Back Tracking is a one type of approach for solving some application, Problem, by applying this approach that tries out all the possible solutions, choose the desired one.

⇒ The Backtracking approach explores various path to find a sequence path that take us to the solution along these path it establish some small check points from where the problem can be backtrack if no feasible solution is found.

→ The Process continues until the best solution is found.





Applications of Backtracking:-

* 8 Queens Problem [N-Queens]
* Sum of subset Problem
* Graph coloring
* 0/1 Knapsack Problem

→ 8 Queen's Problem using BackTracking:-

1) the 8 Queen's Problem is the problem of placing 8 Queen's on 8x8 chessboard.

2) such that none of them attack one another (no two queens are same row & same column or diagonal)

4 Queen's Problem:-

BackTracking Approach:-

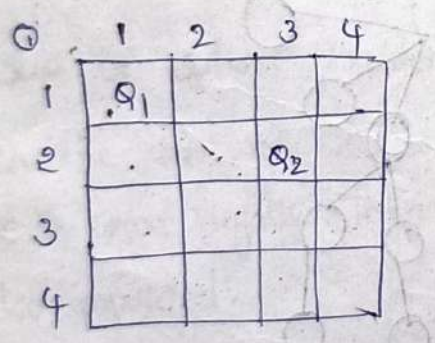
1) Start in the left most column

2) If all 4 Queen's are placed, then return true

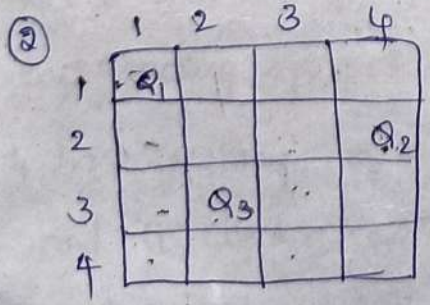
3) Try all rows in the current column for each row i.e., the Queen can be placed safely in this row, then mark this cell & try to solve the problem recursively for the rest of the column.

4) If placing a tree in any row in the current column and Proceeding with this placement then Backtrack & return false.

Example:-

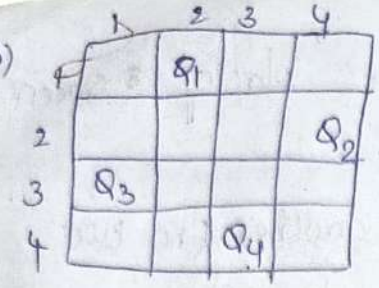


Q_3 didn't get place then we will backtrack to the Q_2.



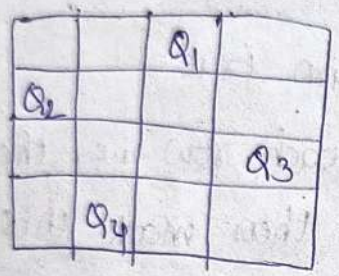
Q_4 didn't get place then backtrack to the Q_3, Q_2 & Q_1

3)

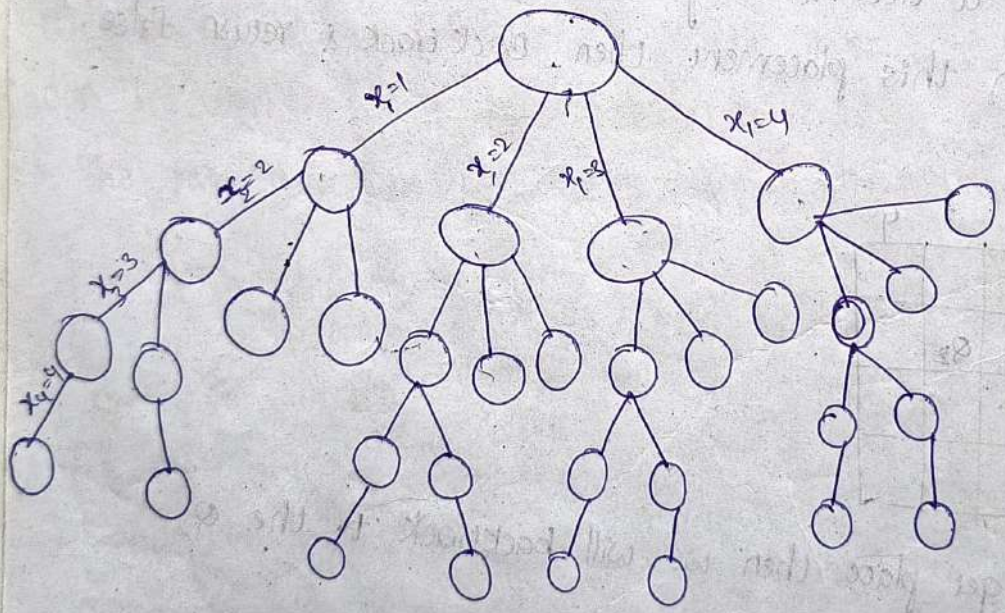


The solution vector $(x_1, x_2, x_3, x_4) = (2, 4, 1, 3)$

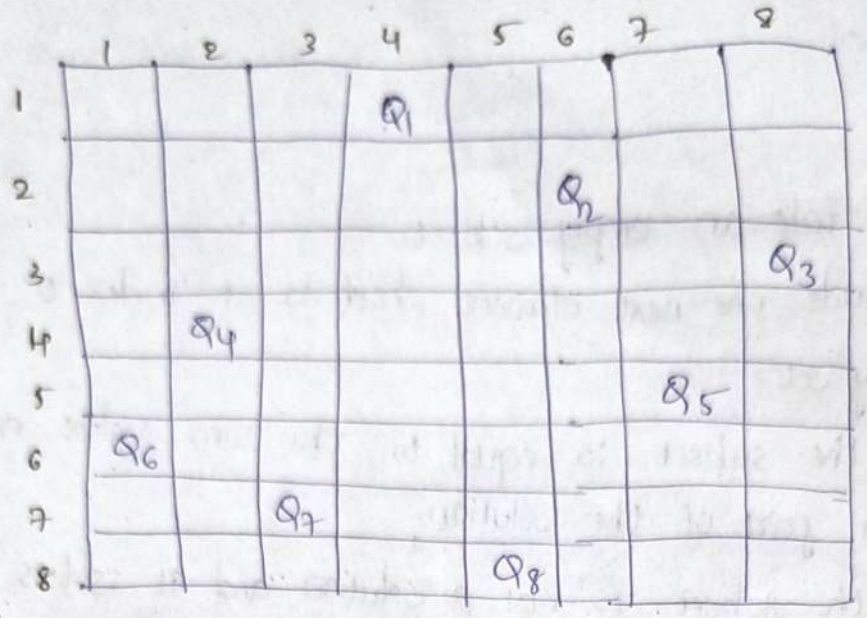
One more solution is mirror image, by getting this solution in reverse, the solution vector - $(3, 1, 4, 2)$



State space tree for 4 Queen's problem:



8 Queen's Problem:-



The solution vector $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (4, 6, 8, 2, 7, 1, 3, 5)$

State space Tree:-

$$1 + \sum_{i=0}^{7} \left[\prod_{i=0}^{j} (8-i) \right] = 69,821$$

The no. of nodes for state space tree of 8-Queen's Problem = 69,821

Sum of subset Problem by using Back Tracking:-

In the sum of subsets problem there is a given set with some non negative integer elements & another sum value is also provided.

Our aim is to find the all possible subsets of the given set whose sum is the same as the given sum value.

Back Tracking approach to solve sub set Problem:-

1) Back Tracking approach generates all the Possible Permutations and it checks for valid solution satisfies the constraints, mark it as a part of the solution.

2) The Back Tracking approach is used for selecting a valid subset. When an item is not valid we will backtrack

to get the previous subset and add another element to get the solution.

Approach:-

Step 1: First take an empty subset

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

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

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

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

Ex:-

Let $n=4$

$w_1, w_2, w_3, w_4 = \{7, 11, 13, 24\}$

Sum value $m = 31$

Solution:-

The sum value of subset should be 31

The possible permutations are x_1, x_2, x_3, x_4

$= (1\ 1\ 1\ 0)$

$= (1\ 0\ 0\ 1)$

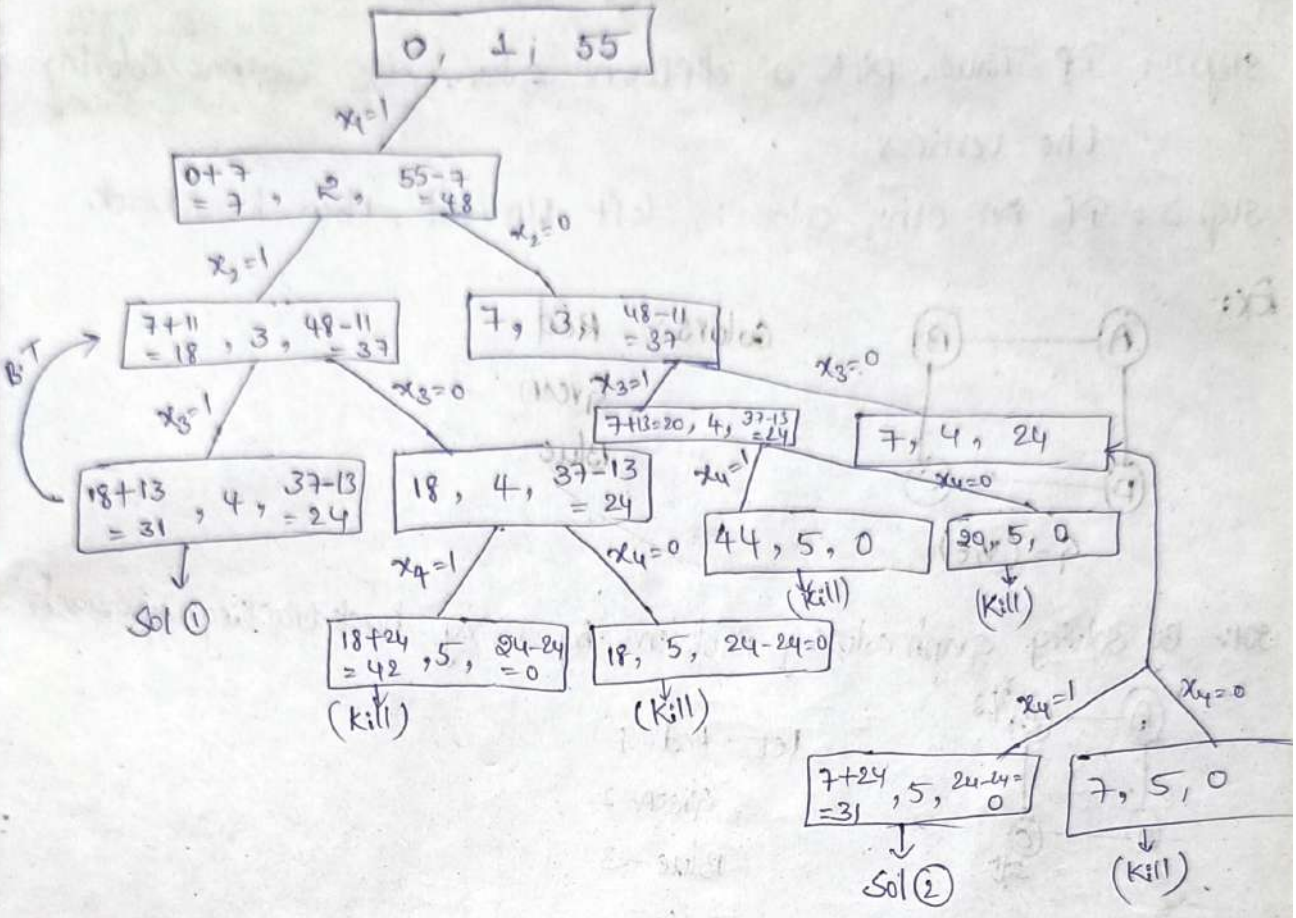
To solve the sum of subset Problem we follow the back tracking approach, by using backtracking approach we can draw a state space tree.

STATE SPACE TREE:

we represent the node as

(w_1, w_2, w_3, w_4)

$(7, 11, 13, 24)$



Graph coloring Problem :-

* The Graph coloring problem involves assigning colors to certain elements of a graph.
* The process of assigning colors to the vertices such that no two adjacent vertices have the same color. is called Graph coloring.

chromatic Number :-

The no. of colors needed to color a graph is called its chromatic number.

BactTracking approach :-

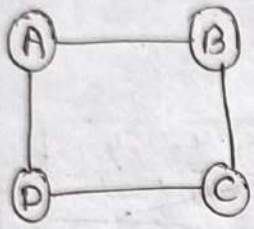
In this approach the idea is to color a vertex & while coloring any adjacent vertex, choose a different color.

Step 1: Consider a color and check if it is valid, i.e., from the given vertex check whether its adjacent vertices have been colored with the same color.

Step 2: If True, pick a different color, else continue coloring the vertices.

Step 3: If no other color is left unused, then backtrack

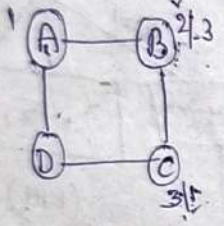
Ex:-



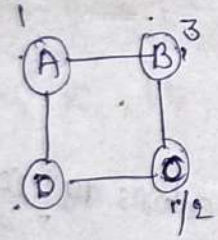
$G = (V, E)$

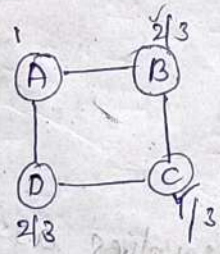
Colors:- Red
Green
Blue

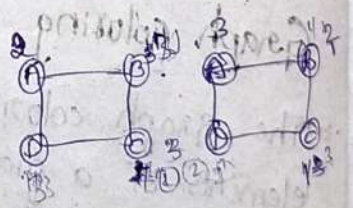
Sol:- By solving Graph coloring Problem by using back tracking approach



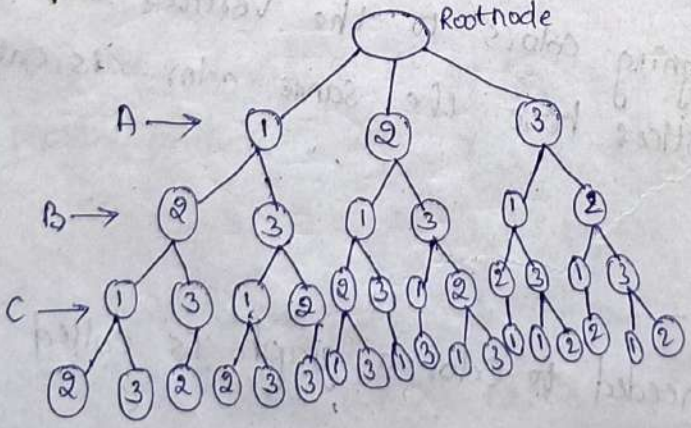
let Red - 1
Green - 2
Blue - 3







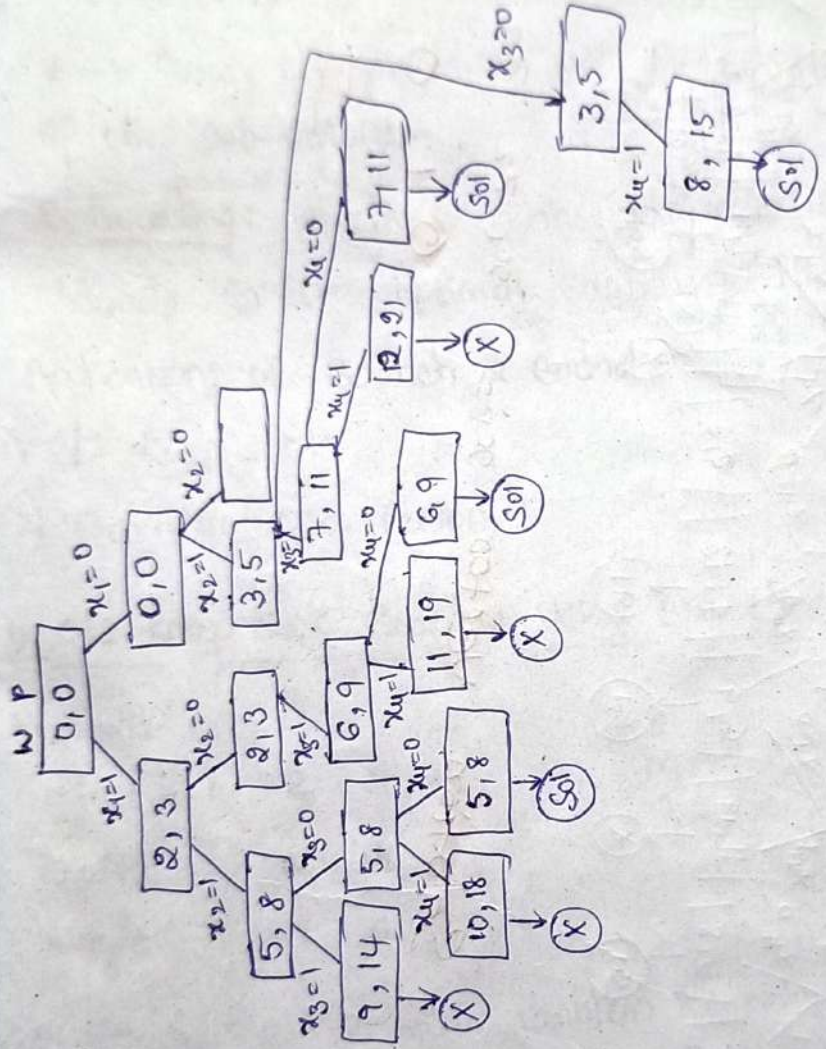
State space Tree for all possible solutions :-



0/1 knap sack Problem Using B.T

obj	1	2	3	4
weight	2	3	4	5
Profit	3	5	6	10

M = 8.



0/1 Knap Sack Problem using Back Tracking:

Obj	1	2	3	4
Profit	3	5	6	10
Weight	2	3	4	5

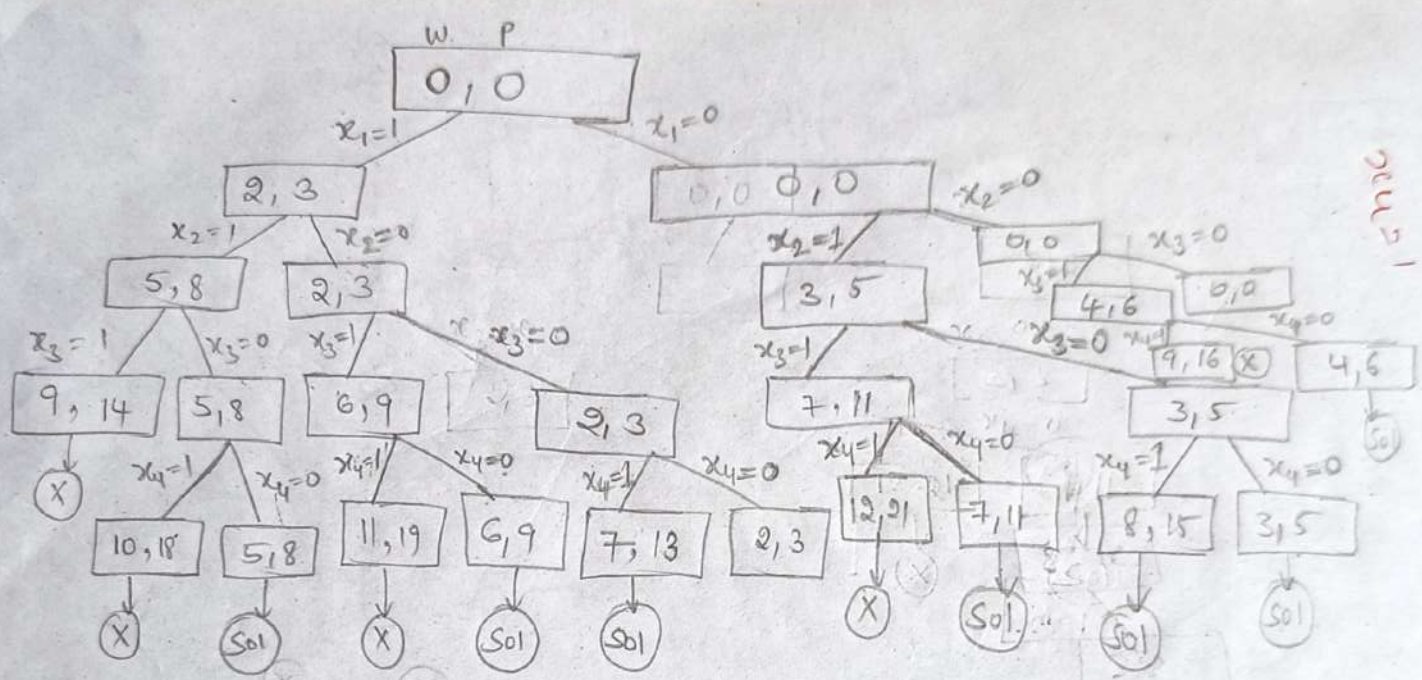
m = 8

$x_2 = 1$

$x_4 = 1$

$w = 2\ 3\ 4\ 5$

$P = 3\ 5\ 6\ 10$



Solution vector x_i = x_1 x_2 x_3 x_4

0 1 0 1

Weight = 5 + 3 = 8

PW = 15

Branch & Bound: General Method.

Branch & Bound is a systematic method for solving a Problem. It is a general optimization Technique that applies where the greedy method & dynamic Programming fail.

Branch & Bound procedure requires two tools:-

1) Branching: A way of covering the feasible region by several smaller feasible sub regions (IDLE, splitting into sub regions)

→ Since, the procedure may be repeated recursively to each of the sub regions.

2) Bounding: Which is a fast way of finding upper & lower bounds for the optimal solution within a feasible sub region.

Applications of Branch & Bound:-

1) 0/1 Knapsack
2) Travelling Sales Person

1) 0/1 Knap Sack Problem Using Branch & Bound: (least cost BB)

Problem:-

object	1	2	3	4
Profit	10	10	12	18
weight	2	4	6	9

$M = 15$

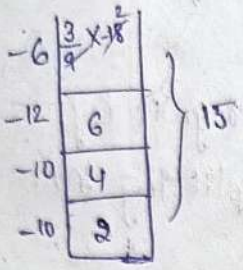
Sol:- For solving the above Problem we need to draw a state space Tree. The state space Tree for each node will be calculated based upon the lower bound ($\hat{c}$) and upper bound ($\hat{u}$)

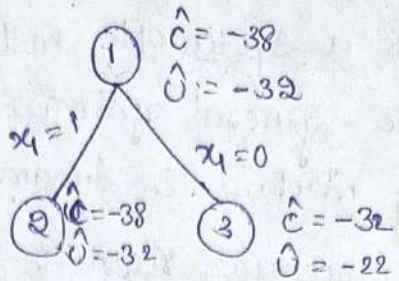
By applying the least cost Branch & Bound method we need to find Minimization of Profits. So, we will negate the given profits

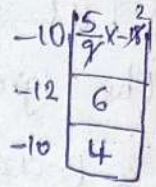
$\therefore$ Profits = $-10, -10, -12, -18$

$\hat{C} = -38$

$\hat{U} = -32$





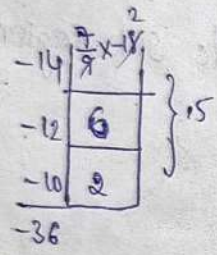


We need to find sequence for which $\hat{U}$ value is least value.

FIFO :-

1) If lb > glb then kill that node

glb → initially root node $\hat{U}$ value



x_1	x_2	x_3	x_4
1	1	0	1
-10	-10		-18

$= -38$

$= 38$

Travelling Sales Person Problem Using Least Cost Branch

Cost matrix:

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Sol:-

First we need to find the reduced cost matrix

Reduced cost Matrix:

i) Row reduction: Every row had atleast one zero.

ii) Column reduction: every column had atleast one zero.

Row reduction for given cost matrix:

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

1) Find Min value in the row
2) Then subtract all the values from the min value

⇒ Minimum values of every row :-

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix}$$

⇒ After subtraction ⇒

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

column reduction for given cost matrix:

⇒

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Min values

⇒ Min values of column –

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	13	12	∞

⇒ After subtraction –

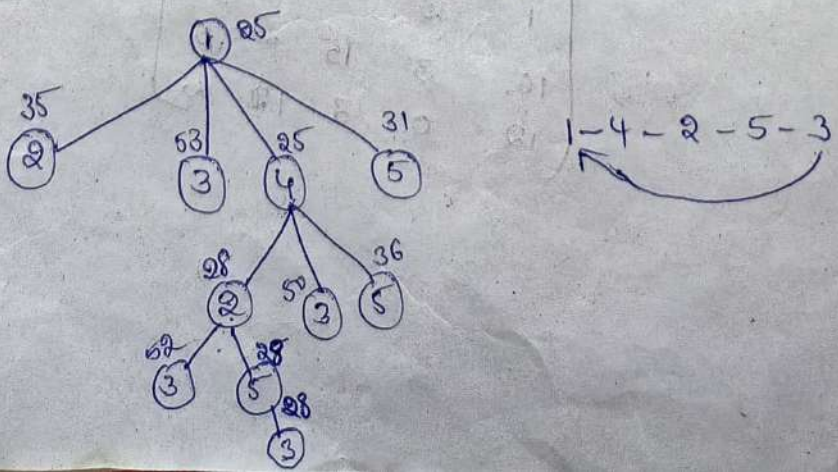
∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Root node cost = Min values of row + Min values of column

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

= 25

⇒ Let us find cost for draw state space tree.



Let us find cost for (1,2)

$(\overset{i}{1}, \overset{j}{2})$

Rule-1: change i^{th} row & j^{th} column to infinity

Rule-2: $(j,1) \Rightarrow \infty$

Rule-3: Find reduced cost matrix

$(i,j) = (1,2)$ $\quad$ $(j,1) = (2,1)$

$\Rightarrow$

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{pmatrix}$$

$\Rightarrow$ Every row & column has min value 0.

$\therefore$ Reduced cost matrix $= 0$ [RCM of (1,2)]

(1,2) Cost $\Rightarrow$ $25 + 10 + 0 = 35$

(25: root node value; 10: (1,2); 0: RCM of (1,2))

Let us find cost for (1,3)

$\Rightarrow$

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{pmatrix}$$

Reduced cost matrix $= 11$

(1,3) $\Rightarrow$ $25 + 17 + 11 = 53$

Let us find cost for (1,4):

$$\Rightarrow \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

Reduced cost Matrix = 0

$\Rightarrow$ cost of (1,4) $\Rightarrow$ 25+0+0 = 25

Let us find cost for (1,5):-

(5,1)

$$\Rightarrow \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ \infty & 3 & 12 & \infty & \infty \\ 0 & 0 & 0 & 12 & \infty \end{pmatrix}$$

Reduced cost matrix = 2+3 = 5

$\Rightarrow$ cost of (1,5) = 25+1+5 = 31

from all the above costs of (1,2), (1,3), (1,4), (1,5) the min cost is found & then it becomes root node i.e) (1,4) path has low cost i.e, 25, so Now root node is 4. we need to find the path (4,2) (4,3) (4,5)

Let us find cost for (4,2):-

we need to consider (1,4) matrix as '4' is root node now: $\Rightarrow$

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

RCM = 0

(4,2) $\Rightarrow$

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

cost = 25+3+0 = 28

Now, (4,3) :-

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{pmatrix}$$

RCM = 11 + 2 = 13

$\Rightarrow$ Cost = 25 + 11 + 12 + 2
= 50

Now (4,5) :- (3,1) $\Rightarrow$ (5,1)

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{pmatrix}$$

RCM = 11

$\Rightarrow$ Cost = 25 + 11 + 0
= 36

Matrix (4,2) $\Rightarrow$

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 4 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

(2,3) $\Rightarrow$

(3,1)

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{pmatrix}$$

= 2 + 11
= 13

$\Rightarrow$ Cost = 28 + 13 + 11 = 52

(2,5) $\Rightarrow$

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix}$$

RCM = 0

Cost = 28 + 0 + 0 = 28

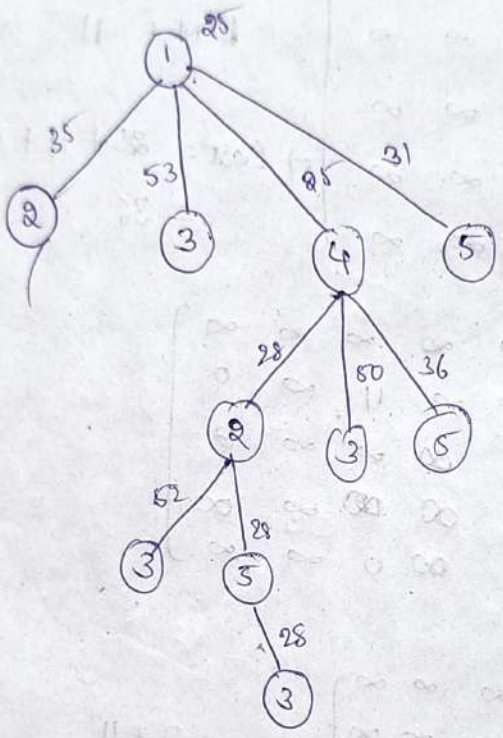
Now (5,3) :-

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

RCM = 0

Cost = 28

$1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 1$ (28)



UNIT-5

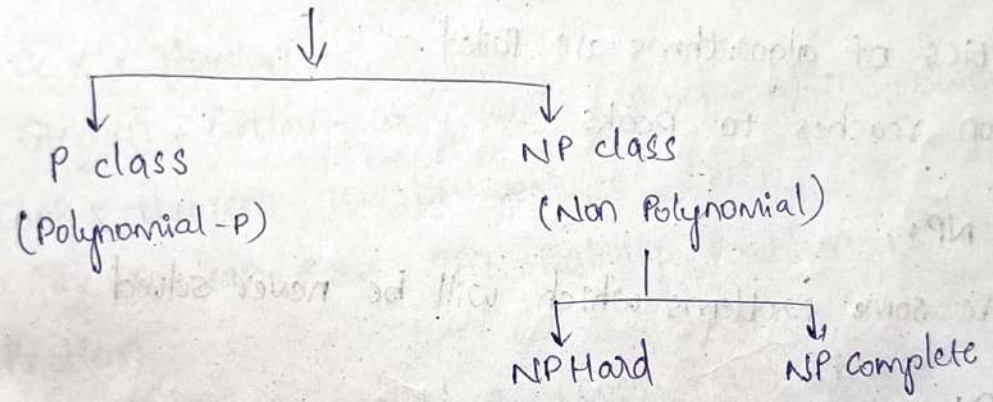
NP Hard and NP complete Problems: Basic concepts, cook's theorem

NP Hard Graph Problems: clique decision Problem (CDP), chromatic number decision Problem (CNDP), Travelling Sales Person Decision Problem (TSP)

NP Hard scheduling Problems: scheduling identical Processor, Job shop scheduling

NP Hard and NP complete Problems:-

The classes of Problems



P-class: In the P-class the 'P' stands for Polynomial's time ~~It~~ this is the collection of all problems which can be solved within the polynomial time.

Ex:- searching & sorting

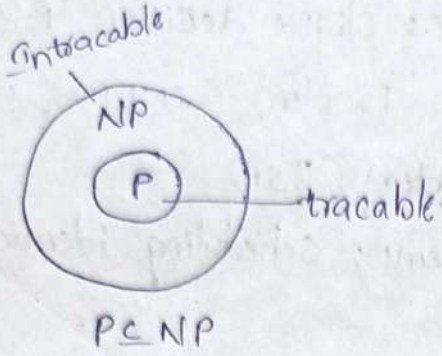
$O(n)$

$O(n^2)$

$O(n \log(n))$

NP-class: The NP-class stands for Non Polynomial (or) Non deterministic Polynomial time, It is the collection of Problems, those problems can't be solved in the polynomial time but these problems can be verified at Polynomial time.

Ex:- sum of subsets Problems
8 Queen's Problems
Pattern Recognition etc.,



Relation b/n P and NP:

1) If P = NP:

All Problems can be solved in less time.
All securities of algorithms are failed.
Computation reaches to peaks

2) If P ≠ NP:

There are some Problems which will be never solved.

Reduction:

(A) ⟹ (B)

Problem 'A' reduced to Problem 'B', If there is a way to solve 'A' by deterministic algorithm that solve 'B' in Polynomial time.

$A \propto B$ [A reduction B]

Properties of Reduction:

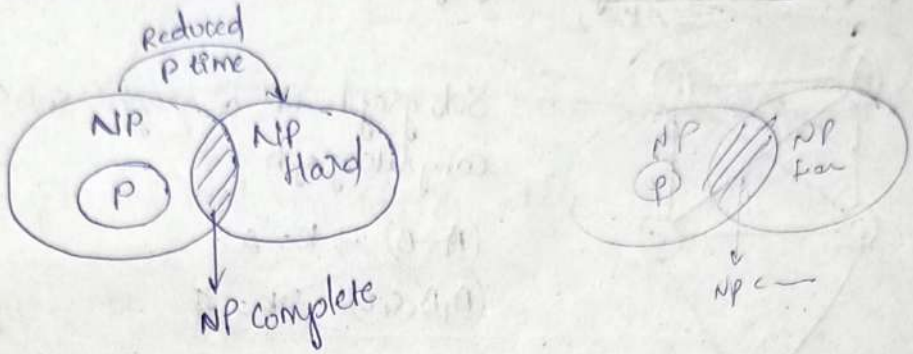
1) If 'A' is reducable to 'B' and 'B' is in 'P' then 'A' is also in 'P'.
2) 'A' is not in 'P' that implies 'B' is also not in 'P'.

NP Hard:

It will reduce all NP problems into another set in Polynomial time, the set is called as NP Hard.

NP complete:

The Problems which are also there in NP & NP Hard then these Problems are called as NP complete.



$$NPC = NP \cap NP\,Hard$$

Cook's Theorem :-

→ Stephen – Arthur – Cook and Leonid – Levin states the Cooks theorem in the year of 1971.

→ The Boolean Satis SATisfiability Problem is NP-complete Problem.

→ The theorem is called as Cook-Levin theorem.

→ Cook theorem states that satisfiability is in 'P' if and only if "P = NP".

→ $P \subseteq NP$

→ Reducability of Problems

$Q_1(A, I_1)$	$Q_2(Z, I_2)$
$P(n)/P()$	$q(m)/q()$
$O(P^3(n)\log n)$	$O(q(m))$

where Q_1: Problem1
A: Non deterministic Polynomial decision making Algorithm
I_1: Inputs
n: no. of Inputs
P(n): Polynomial time

Q_2: Problem 2
Z: Deterministic Algorithm
I_2: Inputs
m: no. of Inputs
q

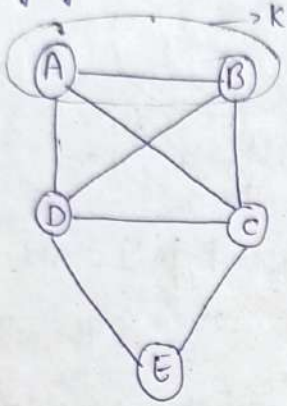
Note:- Q_1 is a Problem is a SATisfiable iff 'A' has a successfull termination with input I_1, i.e, P=NP

NP Hard Graph Problem:-

1) Clique Decision Problem:- (CDP)

Clique: A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other i.e the sub graph is a complete graph.

Eg:-



Sub graph of a graph satisfying complete graph.

(A-B) → K=2

(A,B,C,D) → K=4

(A-B-D) → K=3

(C-D-E) → K=3

No. of Vertices in a sub graph is called size of clique 'K'.

Procedure for proving graph problem had NP Hard:-

1) Let us take P= L2
2) We have to proove that L2 had NP Hard
3) So, we have to select any problem L1 which is already known as NP Hard. i.e., $L_1 \propto L_2$

$$SAT \propto CDP$$

4) By Denoting x_1, x_2, x_3

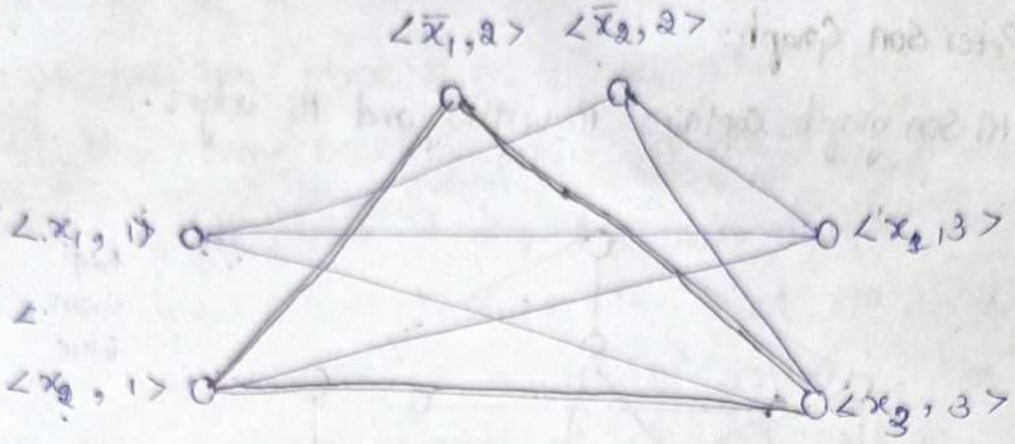
5) $$F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee x_3)$$
$$\quad\quad c_1 \quad\quad\quad c_2 \quad\quad\quad c_3$$

Rules for drawing graph:

1) should not connect the same class of vertices
2) should connect one class vertex to another class vertex
3) should not connect to the negation of classes

$$E = \{(\langle a, i\rangle, \langle b, j\rangle) \mid i \neq j \in b \neq a\}$$

Based upon the above rules we need to construct a graph



After graph, we need to identify clique.

clique- x_1 x_2 x_3

0 1 1

$$F = \overset{0\ 1}{(x_1 \vee \bar{x}_2)} \wedge \overset{1\ 0}{(\bar{x}_1 \vee \bar{x}_2)} \wedge \overset{0\ 1}{(x_1 \vee x_3)} \quad = 3$$

$c_1 = 1 \qquad c_2 = 1 \qquad c_3 = 1 \quad = 3$

Finally we proved that CDP had NP Hard

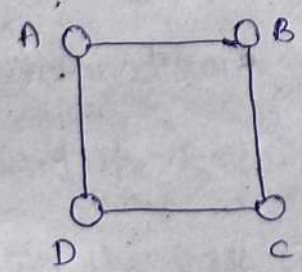
Chromatic Number Decision Problem (CNDP):-

Chromatic Number:- The chromatic number can be described as the min. no. of colors required to properly color any graph.

→ In order to color the graph no two adjacent vertices have the same color.

→ The chromatic number is denoted by $\chi(G)$

Ex:-



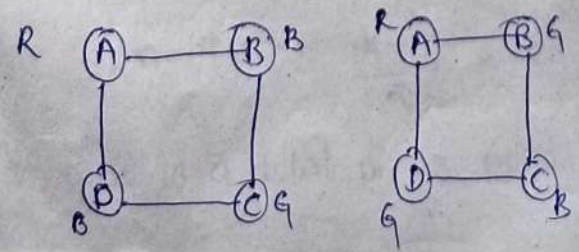
R
B
G

No. of colors taken = 3

Red (R) = 1

Blue (B) = 2

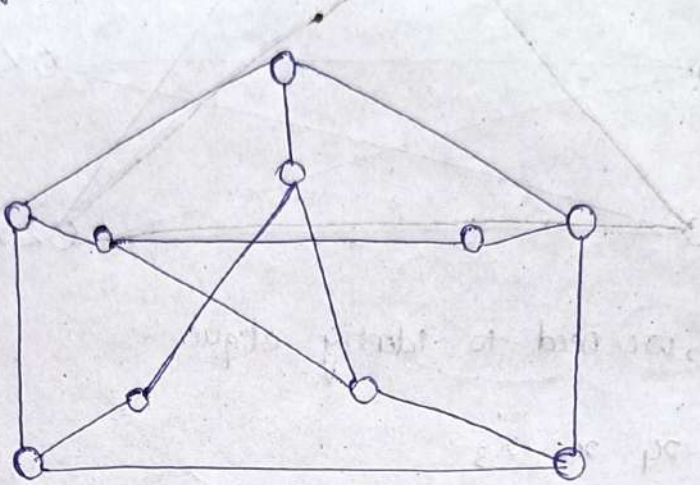
Green (G) = 3



The chromatic Number of above graph $\chi(G) = 3$

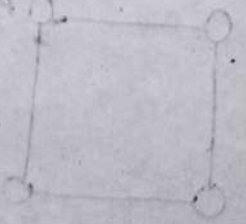
Ex1) Peter Son Graph:-

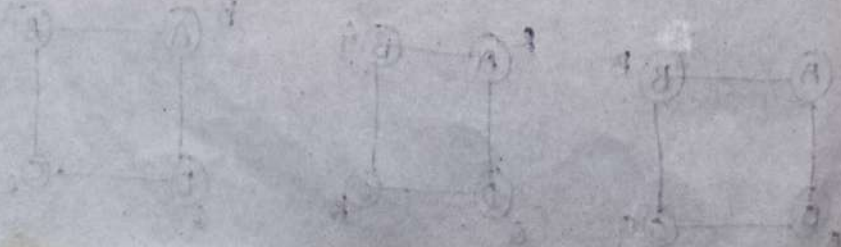
The Peter Son graph contains 10 vertices and 15 edges.



$\chi(G) = 3$

Ex 2) Hershell Graph:- Herschel





Travelling sales Person Problem :- (TSPP)

→ The TSP of a solution is where, a sales man has to start from one place & go to all other cities just once and then come back to their own place.

→ TSP is all about finding the Min. distance path.

→ The polynomial time hardness is called NP-Hard which defines the property of a class of problems, the subset sum is a simple example of NP-Hard Problem

Ex:-

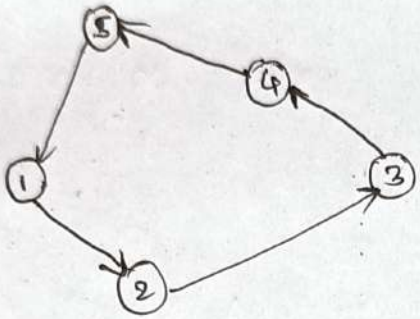


Fig-I

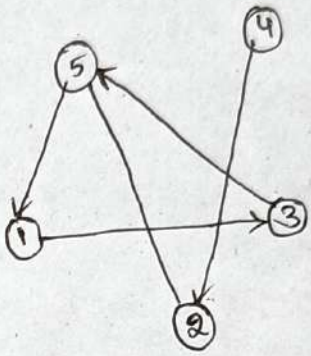


Fig-II

→ In the above 2 figures, fig-II represents the least or minimum distance path

→ In fig 1, when sales Man travel from location one to ~~two~~ five, it takes a lot of time to reach its own location and total distance covered i.e., ∅ $1 \to 2 \to 3 \to 4 \to 5 \to 1$

→ But, In the case of fig-II the sales man directly reach the point at three which saves the time & also cover the total min. distance i.e., $1 \to 3 \to 5 \to 1$

To find the Possibilities of No. of cities Problem :-

ex:- In the case of 4 cities Problem

[Formula $(n-1)!/2$]

$$n = 4 \Rightarrow \frac{(4-1)!}{2} = \frac{3!}{2} = 3.$$

So, there is a total of 3 possibilities for covering the distance