Searching

INTRODUCTION :-

Tree Search (Binary Search Tree).

In most of the cases, files are organized as a binary tree which makes searching efficient.

BST

A binary search tree is a binary tree in which each node has,

(i) Value greater than its left child node.

(ii) Value less than its right child node.

Operations on BST

(i) Insert (V, T) → Inserting new node with value V.

(ii) Delete (V, T) → Deleting a node with value V.

(iii) Search (V, T) → Search for a node with value V.

CONSTRUCTING BST (INSERTION)

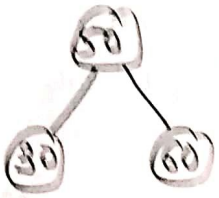
Let us take seven numbers and insert them in binary tree which is initially empty.

50, 30, 60, 22, 38, 55, 34

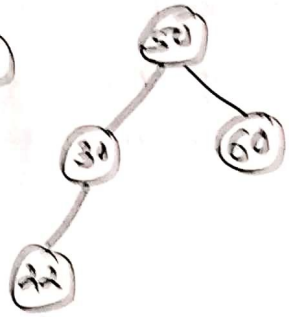
(i) 50 — 1st element as root

(ii) 

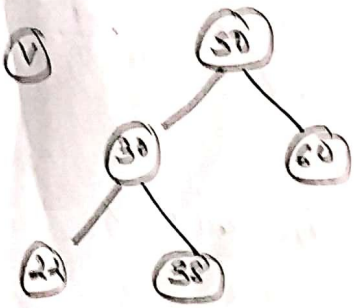
Since (30 < 50) it is added to left.

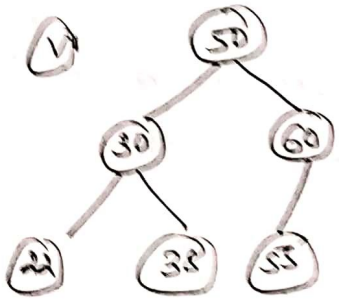
(iii) 

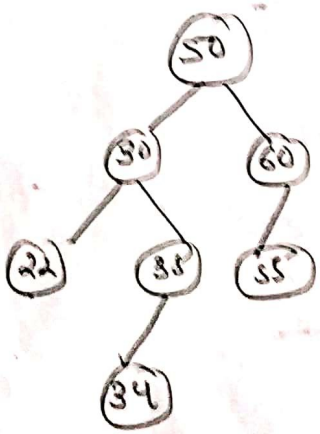
Since (60 > 50) it is added to right.

(iv) 

(22 < 50) and (22 < 30)

(v) 

(vi) 

(vii) 

// code.

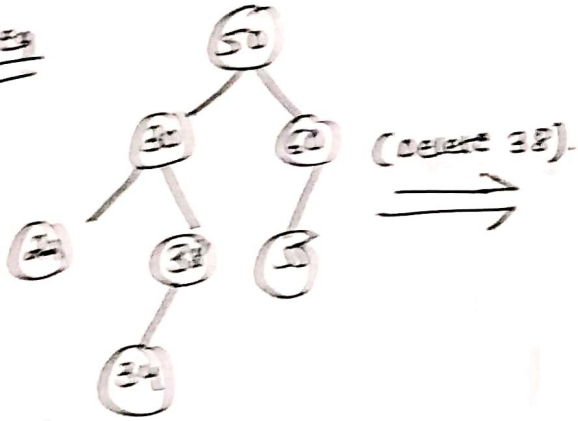
```
Void insertBST ( ).
{
    temp = (node *) malloc (sizeof(node));
    temp → info = item;
    temp → left = NULL;
    temp → right = NULL;

    if ( parent == NULL)
        root = temp;
    else
        if( item < parent → info)
            parent → left = temp;
        else
            parent → right = temp;
}
```

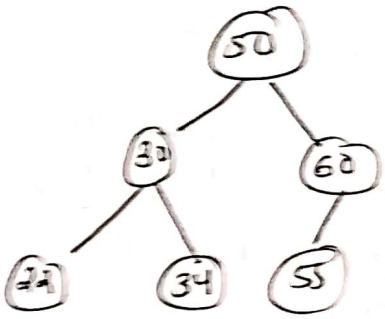
Deleting from BST

```
p = tree;
q = null;
/* search for the node with value (v), set 'p' to point
   to the node and 'q' to its father node, if any */.
while( p != null && info(p) != v)
    {
        q = p;
        p = (v < info(p)) ? left(p) : right(p);
    }
    if( p == null)
        /* value not found */.
```

Eg



(delete 38).

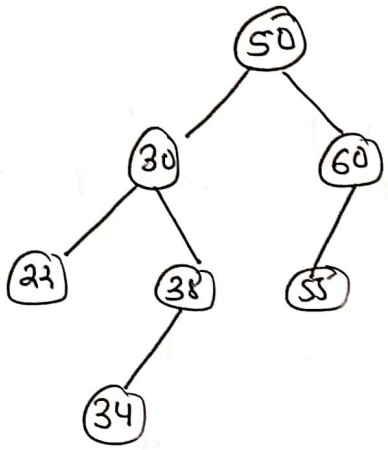


Searching in BST

Algorithm

1. Compare element to be searched (key) with root node value:
   - 1.1 if value match then search is successfull.
   - 1.2 if key is less than root node value, continue search in left subtree.
   - 1.3 if key is greater than root node value, Continue search in right subtree.
2. if key doesn't match with any value in anode then search is unsuccessfull.

eg search for 34 (key) in given BST



Compare 34 and 50, since (34 < 50), search is continued in left subtree of node (50)

↓

Compare 34 and 30, since (34 > 30), search is continued in right subtree of node (30)

Compare 34 and 38, since (34 < 38), search is continued in left subtree of node (38).

Compare 34 and 34, since (34 == 34), match is found.

// code

```
p = tree;
while (p != null && key != info(p))
    p = (key < info(p)) ? left(p) : right(p);
return (p);
```


h(8765) = 8765 % 10 = 5

HASHING

Hashing is a technique where we convert the key into array index and store record in that index position of array.

In the same way for searching, we convert key into array index and retrieve record from that index position.

For storing

Key
↓ (Hash function)
Generate array index
↓
Store record on that array index

For Retrieval / search

Key
↓ (Hash function).
Generate array index
↓
Retrieve record from that array index.

Hash function

A function that transforms a key into array index.

Hash Table

An array which supports hashing for storing and retrieving record.

eg Consider the following records for storing in Hash table of size 10.

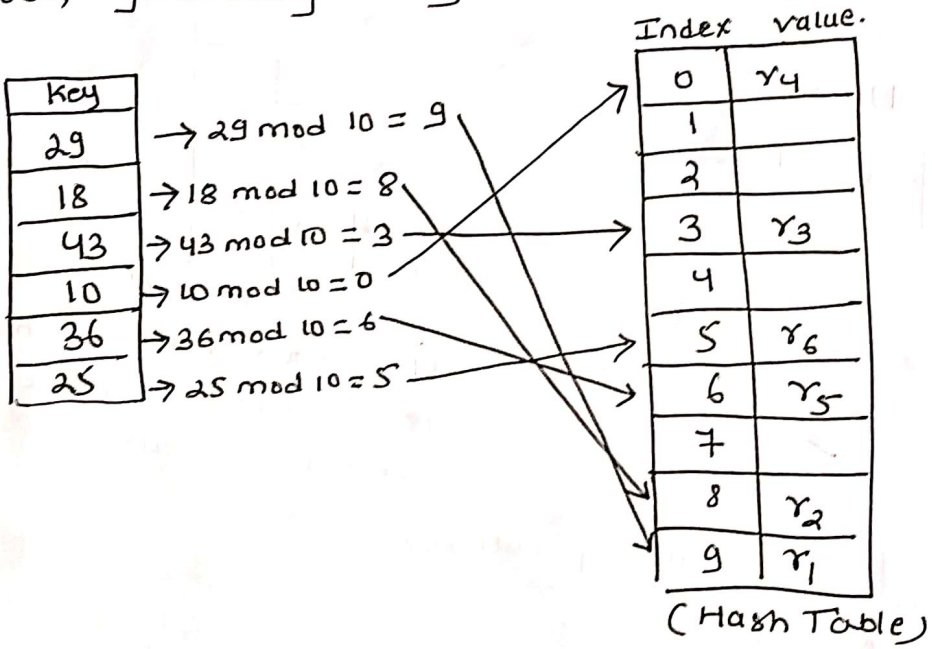
key	record
29	r_1
18	r_2
43	r_3
10	r_4
36	r_5
25	r_6

let's say hash function for generating array or table index is,

h(key) = key mod size

i.e h(key) = key mod 10.

Now, generating array index and storing record.



h(8765) = 8765 % 10 = 5 h(8828) = 8828 ...

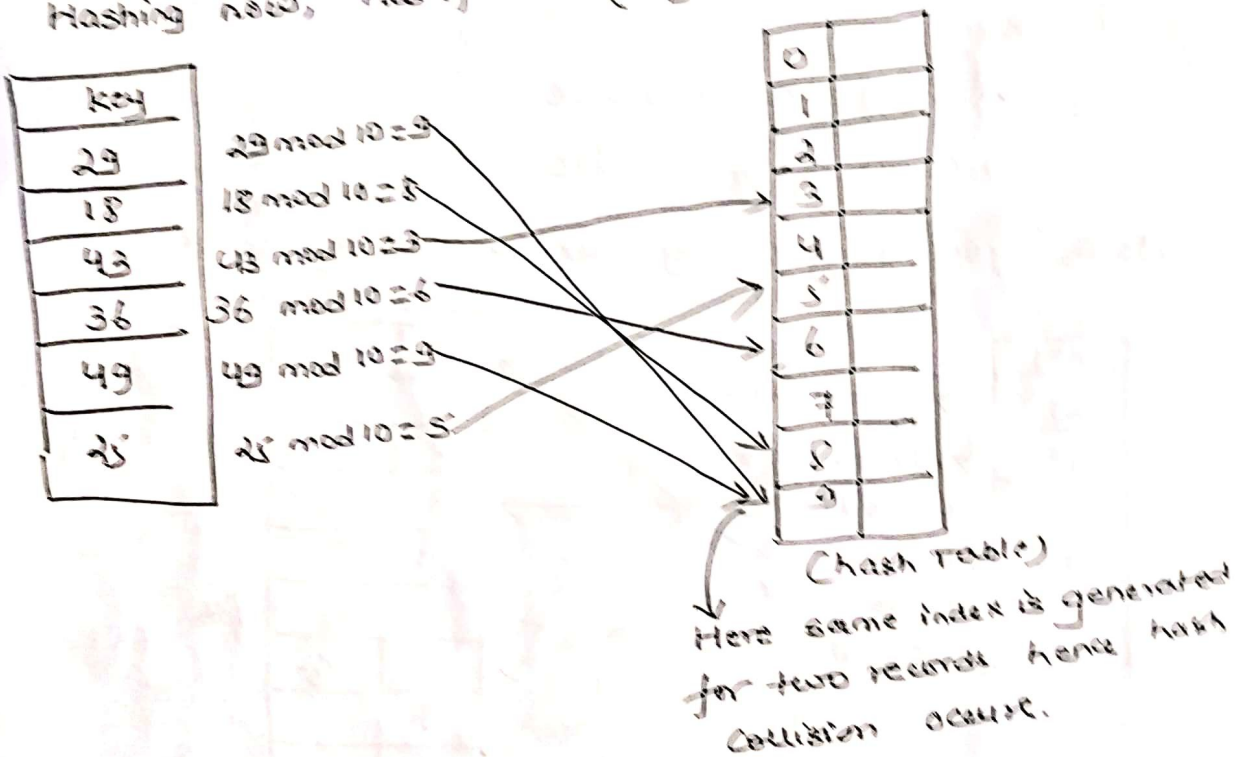
Hash Collision

If a hash function generates same array index for two or more records then such a situation is called hash collision.

Eg. Consider the following records.

Key	29	18	43	36	49	25
record	R1	R2	R3	R4	R5	R6

Hashing used, hash function = (key mod 10.)



methods for minimizing collision

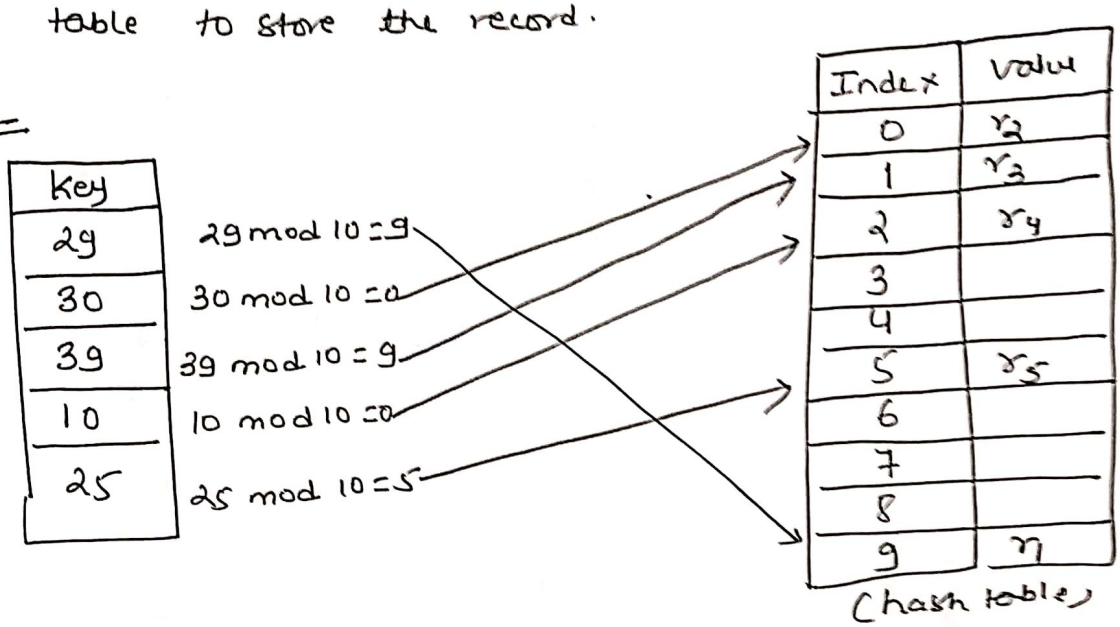
1. Linear probing
2. Chaining
3. Quadratic probing.
4. Rehasing

etc.

Linear probing (open Addressing).

In this method, if key has same index address then it will find the next empty position in hash table to store the record.

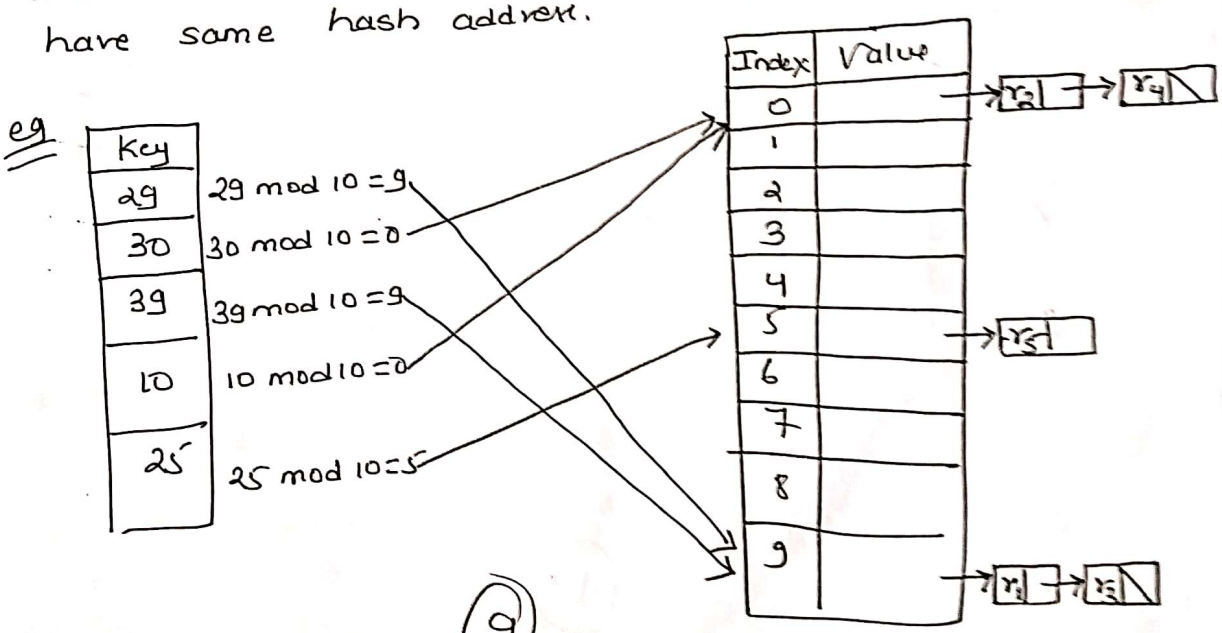
eg



Separate chaining

This method maintains the chain of elements which have same hash address.

eg



h(8765) = 8765 /

Quadratic probing (open addressing)

In this method, in case of same hash address generated we look for empty locations h+1, h+4, h+9 .....

Hashing Algorithm

1- Truncation method.

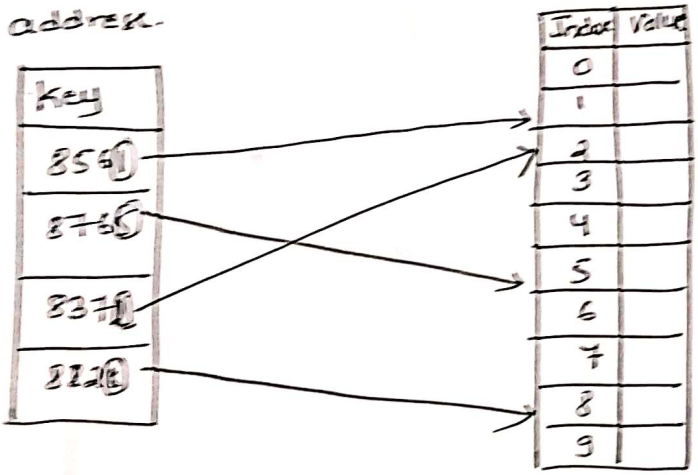
In this method, we take only part of key as address, it can be rightmost digit or leftmost digit.

Eg. Consider the following keys.

8561, 8765, 8372, 8828.

let's assume size of hash table is 10.

Now, we will take only the right most digit and truncate (eliminate) other digits to generate hash address.



2. modular method

Take the key, do the modulus operation by size of hash table and get the remainder as hash address.

i.e $h(x) = \text{key} \bmod \text{size}$.

Eg. 8561, 8765, 8372, 8828 (keys)

size = 10.

$\therefore h(8561) = 8561 \% 10 = 1 \quad h(8372) = 8372 \% 10 = 2$

$h(8765) = 8765 \% 10 = 5 \quad h(8828) = 8828 \% 10 = 8.$

INTERPOLATION SEARCH

In this, key element will be given.

Initially, low is set to 0 and high is set to n-1.

Then we find mid as given below,

$$mid = low + (high - low) * ((key - K(low)) / (K(high) - K(low)))$$

if (key < mid) then search in lower half

i.e from 0 to mid-1.

if (key > mid) then Search in upper half

i.e from mid+1 to n-1

Repeat the process until key has been found.

GRAPH

Definition

A graph 'G' is a collection of two sets V and E where

V = set of Vertices

E = set of edges.

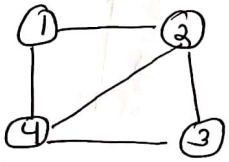
$V(G) = (V_0, V_1, \ldots V_n)$

$E(G) = (e_1, e_2 \ldots e_n)$

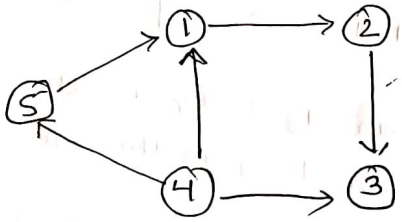
A graph can be of two types

1. Undirected Graph → A graph, which has unordered pair of Vertices.

eg.

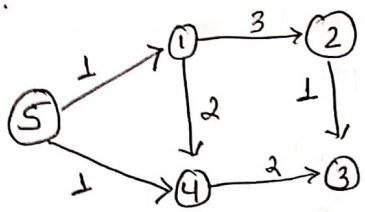


2. Directed graph → A directed graph or digraph is a graph which has ordered pair of Vertices.

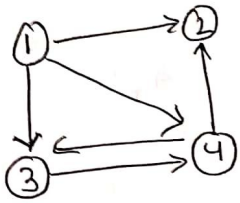


(2) GRAPH TERMMINOLOGIES

Weighted graph → A graph is said to be weighted if it's edges have been assigned some non-negative value as weight.

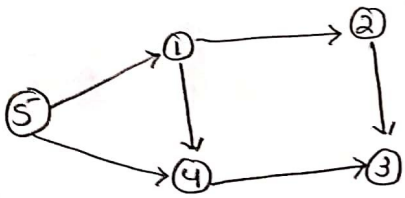


Adjacent Node: Two nodes are said to be adjacent if there exist an edge between them.



Here, node ① & ③ are adjacent node.
⟨1,2⟩ are adjacent
⟨3,4⟩ " "
⟨1,4⟩ are adjacent etc.

Path : A path from u_0 to u_n is the sequence of nodes in between that has to pass through to reach u_n.



A path from ⑤ to ③ are,
(i) 5 - 1 - 2 - 3
(ii) 5 - 4 - 3
(iii) 5 - 1 - 4 - 3

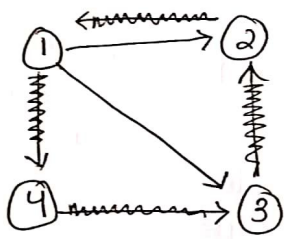
3

Length of path:- Length of a path is the total no. of edges included in the path.

cycle : Cycle is a path in which 1st node and last node is same.

cyclic graph : A graph that contains a cycle is called cyclic graph.

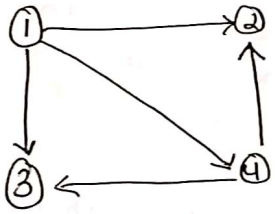
eg



In this graph, there exist a cycle, i.e 1-4-3-2-1

Acyclic graph: A graph that doesn't contain a cycle.

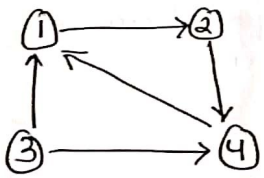
eg



Degree of a node : Number of Edges connected to a node.

Indegree : Number of edges incoming to a node.

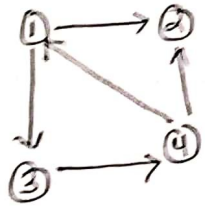
eg.



Node	Indegree
1	2
2	1
3	0
4	2

Outdegree :- The number of outgoing edges from a node.

eg.

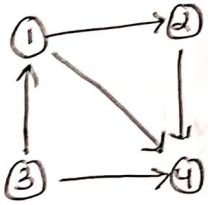


Node	outdegree
1	2
2	0
3	1
4	2

Source node :- A node, which has no incoming edges.

Sink node :- A node, which has no outgoing edges.

eg.



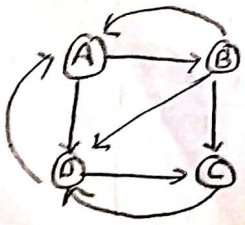
Here, node ③ has no incoming edges so, it is source node.

node ④ has no outgoing edges so, it is sink node.

Reachable node : If there is a path from a node to any other node then it is called as reachable node.

Adjacency matrix : A matrix that shows adjacency relation between nodes. The entries of this matrix will be either 1 or 0.

eg



Adjacency matrix (M) =

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

(5)

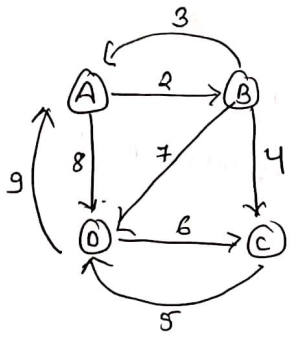
Weighted Adjacency matrix:

It is defined as follows :-

Arr[i][j] = weight on edge if there is edge from node i to j.

= 0 if there is no edge from node i to j.

eg



Weighted adjacency matrix (m) =

	A	B	C	D
A	0	2	0	8
B	3	0	4	7
C	0	0	0	5
D	9	0	6	0

C REPRESENTATION OF A GRAPH

```
#define maxnodes N

Struct node {
        struct node *next; //pointer to next node
        char name;  // name of node
      struct edge *adj; // pointer to 1st adjacent node.
      };
```

1-2-4-5-3-7-6

6)

```
struct edge
{
  char dest; // destination node of edge.
  struct edge *link; // pointer to next adjacent node.
};

struct graph
 {
    struct node Nodes[maxnodes];
   struct edge Edges[maxnodes][maxnodes];
 };
```

8 TRAVERSAL IN GRAPH

Visiting each node (Vertex) in a graph is called Graph traversal.

- Types of Graph traversal

(i) Breadth first Search (BFS)

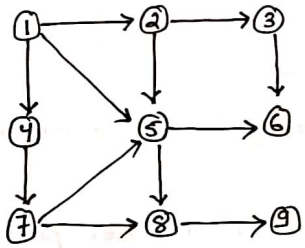
(ii) Depth first search (DFS)

Breadth First Search (BFS)

In this,

1. First we take any node as starting node and visit all its adjacent nodes.
2. Then we take these adjacent nodes one by one and visit its neighbour node.
3. Repeat the process until all nodes are visited.

eg



let's take 1 as starting node and visit its adjacent nodes.

so, now traversal is,

1 - 2 - 4 - 5

Now, first we will traverse all adjacent nodes of 2, then all adjacent nodes of 4 and then adjacent nodes of 5.

so, now traversal is,

1 - 2 - 4 - 5 - 3 - 7 - 6

(8) Now, we will traverse adjacent nodes of 3, 7 and 6.

So, traversal is,

1-2-4-5-3-7-6-8

Now, finally we will traverse adjacent node of 8.

So, we have

1-2-4-5-3-7-6-8-3

Depth First Search (DFS)

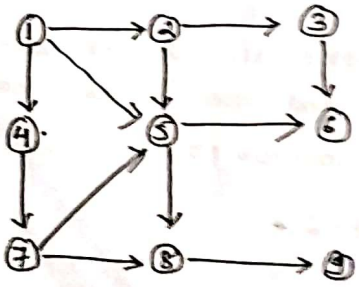
The DFS is a recursive algorithm that uses the idea of backtracking.

Backtracking means that when you are moving along a path and reach dead end then you move backward on that path to find another path that you can travel.

DFS algorithm

1. Pick any node as starting node.
2. Visit all nodes along a path, starting from a start node. until you reach dead end.
3. Traverse back in that path to find another path to be traversed.
4. Repeat the process until all nodes are visited.

eg



9) DFS continued....

Let's pick (1) as a starting node

First, we will traverse node 1. Then we will traverse any node adjacent to node 1 and follow that path until we reach dead end.

So, traversal is,

1 - 2 - 3 - 6.

Since 6 is the dead end, we will traverse back to previous node 3, see if there is any path from node (3). No such path is there so, we will traverse back to node (2). There is another path from (2).

So, traversal is

1 - 2 - 3 - 6 - 5 - 8 - 9.

now, 9 is the dead end.

We will continue backtracking until all nodes are visited once.

Final traversal is,

1 - 2 - 3 - 6 - 5 - 8 - 9 - 4 - 7

1 or 3, 7 and 6. (7)

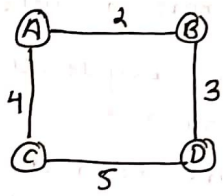
Difference between BFS and DFS (10)

BFS	DFS
(i) Search is done breadthwise.	(i) Search is done depthwise.
(ii) BFS is implemented Using queue.	(ii) DFS is implemented Using stack.
(iii) It requires more memory space.	(iii) It requires less memory space.
(iv) BFS is slower.	(iv) DFS is faster.
(v) Application : # To find shortest path. # In spanning Tree. # In Connectivity.	(v) Application : # In cycle detection. # In Connectivity Testing.

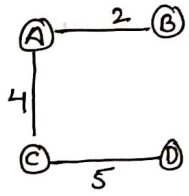
Spanning Tree (11)

A spanning tree of a connected graph 'G' is the one that contains all the nodes and has the edges, which connect all the nodes.

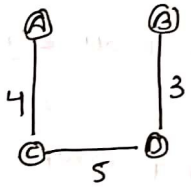
eg. let us take a Graph G,



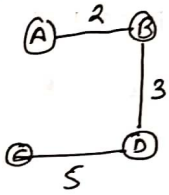
Now, spanning tree for this graph are:-



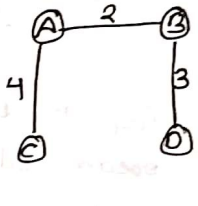
(a)



(b)



(c)



(d)

MINIMUM SPANNING TREE (MST)

Minimum Spanning tree is the spanning tree in which the sum of weights on edges is minimum.

For example: For the above graph, (d) is the MST.

Algorithm to Find MST

(i) Kruskal's algorithm

(ii) Prim's algorithm.

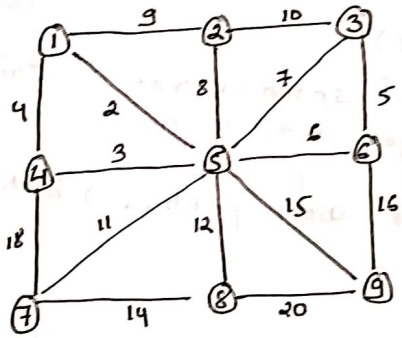
KRUSKAL'S ALGORITHM

It is an algorithm to find minimum Spanning tree.

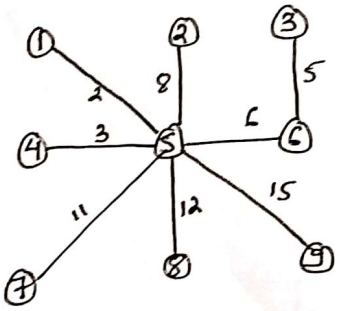
Procedure is as follows:-

1. Pick the edges in ascending order of their weight.
2. check if edge selected forms a cycle or not.
   - (2.1) If this edge forms a cycle then it is not inserted in spanning tree.
   - (2.2) if this edge doesn't form a cycle then it is inserted in a spanning tree.
3. Repeat the process until all edges have been examined.

eg Consider the following Graph,



Weight (Ascending order)	Edges	Cycle/No cycle
2	1-5	No cycle
3	4-5	No cycle
4	1-4	Cycle
5	3-6	No cycle
6	5-6	No cycle
7	5-3	Cycle
8	2-5	No cycle
9	1-2	Cycle
10	2-3	Cycle
11	5-7	No cycle
12	5-8	No cycle
14	7-8	Cycle
15	5-9	No cycle
16	6-9	Cycle
18	4-7	Cycle
20	8-9	Cycle.



MST

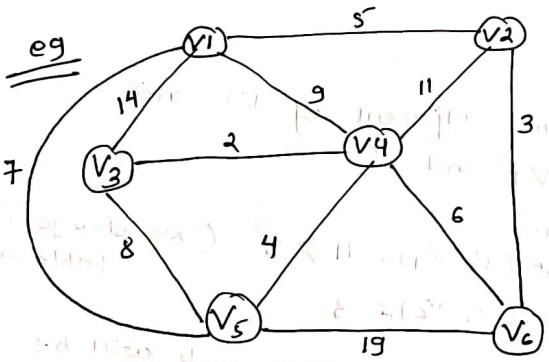
PRIM's ALGORITHM

Prim's algorithm is very similar to Dijkstra's shortest path algorithm, to find minimum spanning tree.

Procedure

1. An arbitrary node is chosen as current node of a spanning tree and is marked permanent (P).
2. The distance of adjacent nodes of current node is examined, and the next node to be added in a spanning tree is the one with minimum distance or weight.
3. Node that is added in a spanning tree now becomes the current node.
4. step (2) & (3) is repeated until all nodes are included in a spanning tree.

eg



(i) Original graph.

(14)

Initially, chose (V1) as current node.

Nodes	Distans	Status
V_1	0	CURRENT
V_2	5	Temp
V_3	14	Temp
V_4	9	Temp
V_5	7	Temp
V_6	∞	Temp

adjacent of (V1) are, (V2), (V3), (V4) & (V5)

now,
dist (V1, V2) = 5
dist (V1, V3) = 14
dist (V1, V4) = 9
dist (V1, V5) = 7

Now, out of all these, 5 is the minimum so, next current node will be V2.

Nodes	Dist	Status
V_1	0	cur
V_2	5	cur
V_3	14	temp
V_4	9	temp
V_5	7	temp.
V_6	3	temp.

now, adjacent of V2 are V4 and V6.

now,
dist (V_2, V_4) = 11 > 9 (No change in table row)
dist (V_2, V_6) = 3

so, next current node will be (V6)

(15)

nodes	dist	status
V_1	0	cur
V_2	5	cur
V_3	14	temp
V_4	~~9~~ 6	temp
V_5	7	temp
V_6	3	cur.

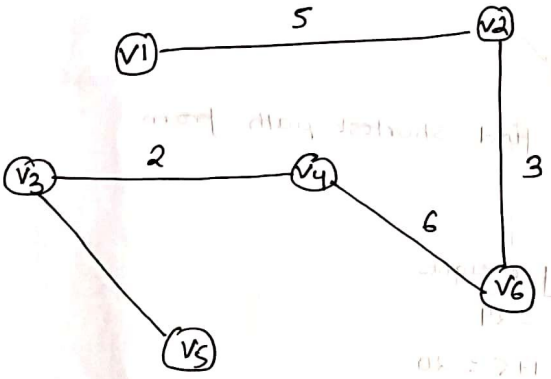
adjacent of V_6 are V_4 & V_5.

dist $(V_6, V_4) = 6 < 9$ (so, table value changes to 6)

dist $(V_6, V_5) = 19 > 7$ (so, no change).

now, out of all 'temp' nodes minimum is 6 so, V_4 will be next current node.

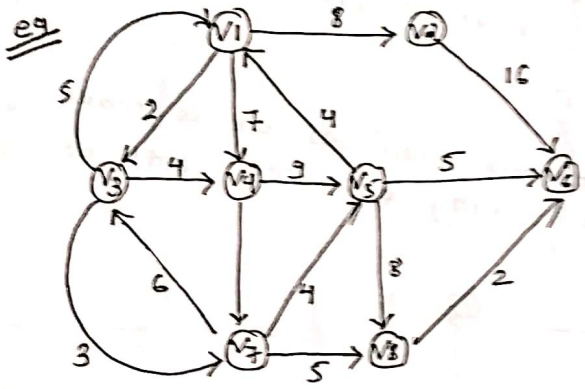
This process will continue until all node is made current node. Hence finally our MST will be.



DIJKSTRA SHORTEST PATH ALGORITHM

As we know in a graph, there can be several path from one node to another node, but shortest path is that path in which the sum of weight included on edges is minimum.

eg



let's say, we have to find shortest path from V1 to V6

Possible path	sum of weights
V1 – V2 – V6	8 + 16 = 24
V1 – V3 – V4 – V5 – V6	2 + 4 + 9 + 5 = 20
V1 – V4 – V5 – V6	7 + 9 + 5 = 21
V1 – V3 – V7 – V5 – V6	2 + 3 + 4 + 5 = 14
V1 – V3 – V7 – V8 – V6	2 + 3 + 5 + 2 = 12

Among all these paths, the path with minimum weight is,
V1 – V3 – V7 – V8 – V6.

∴ This is the shortest path from V1 to V6.

Procedure

1. Initially make source node permanent and make it the current working node. All other nodes are made temporary.
2. Examine all adjacent nodes of current working node and find out its distance from current node.
3. From all temporary nodes, find out the node with minimum weight and make that node as next permanent node.
4. Repeat step 2 & 3 until destination node is made permanent.

WARSHALL'S ALGORITHM