Unit-8

Trees and Giraphs:

A) Trees:

A tree is a non-linear data structure and is generally defined as a nonempty finite set of elements which consists of set of nodes called vertices and set of edges which links vertices such that:

→ Tree contains a distinguished node called root of the tree.
→ The remaning elements of tree form an ordered collection of zero or more disjoint sets called subtree.

⊛. **Binary Trees:-** A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees of the original tree. A left or right subtree can be empty. Each element of a binary tree is called a node of the tree.

Properties:-

→ The order of binary tree is 2.
→ Binary tree does not allow duplicate values.
→ Binary tree has maximum of two child-node at each node.
→ Maximum number of nodes in a binary tree of height 'h' is 2^{h-1}

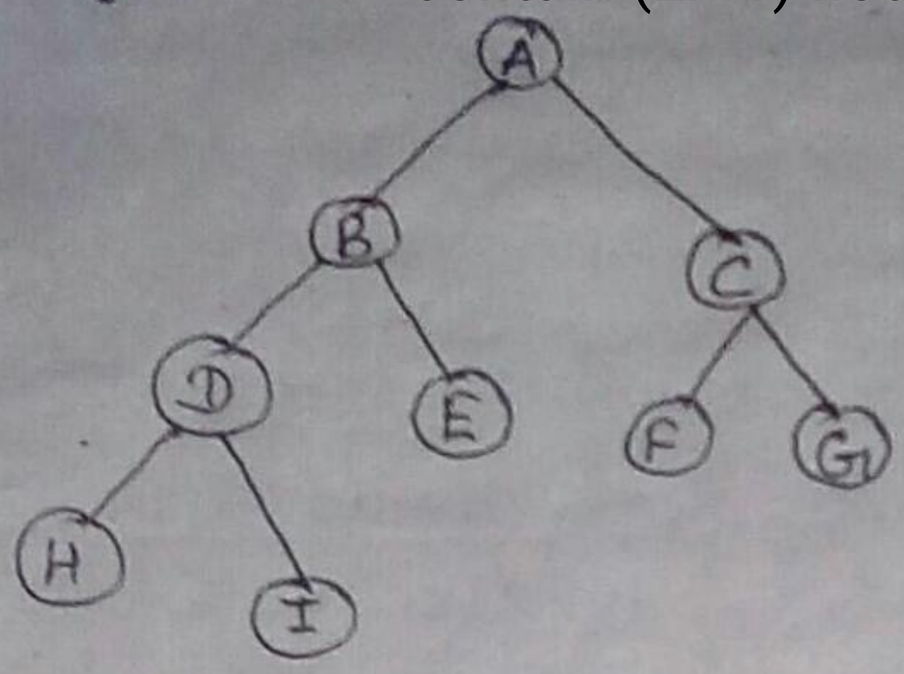
Advantages:-

→ Searching in Binary tree becomes faster.
→ Maximum and minimum elements can be directly picked up.
→ It is used for graph traversal and to convert an expression to postfix and prefix forms.

Types of Binary Tree:-

1) **Strictly binary tree** → If every non-leaf node in a binary tree has non-empty left and right sub-trees, then such tree is a strictly

binary tree. A strictly binary tree with n leaves always contain (2n-1) nodes.



non-leaf node e.g. node D , B in fig.

fig. A strictly binary tree.

ii) Complete binary tree → A ~~complete~~ binary tree of depth d is called complete binary tree if all of the leaves are at level d. A complete binary tree with depth d has 2^d leaves and 2^{d-1} non-leaf nodes.

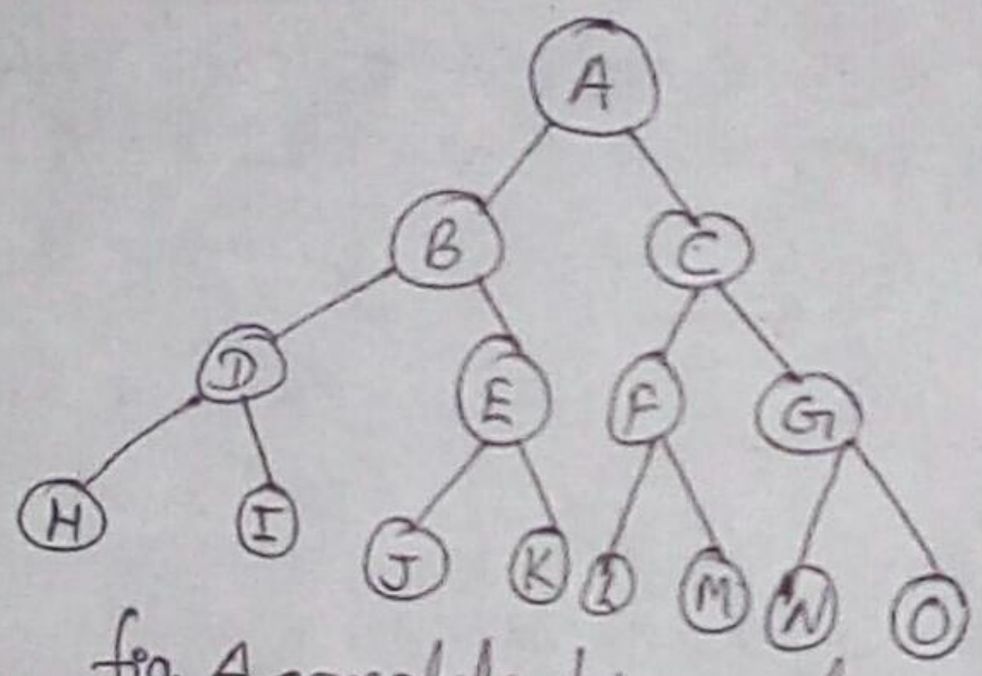


fig. A complete binary tree of depth 3.

iii) Almost complete binary tree → A binary tree of depth d is an almost complete binary tree if:

→ any node nd at level less than d-1 has two sons.

→ for any nose nd in the tree with a right descendant at level d, nd must have a left son and every left descendant of nd is either a leaf at level d or has two sons.

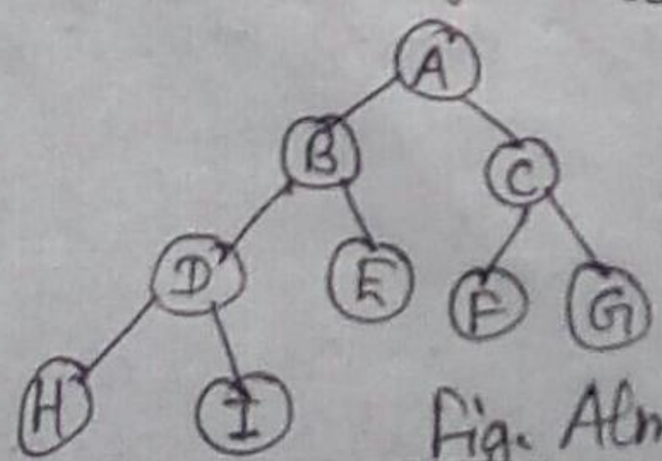


Fig. Almost complete binary tree.

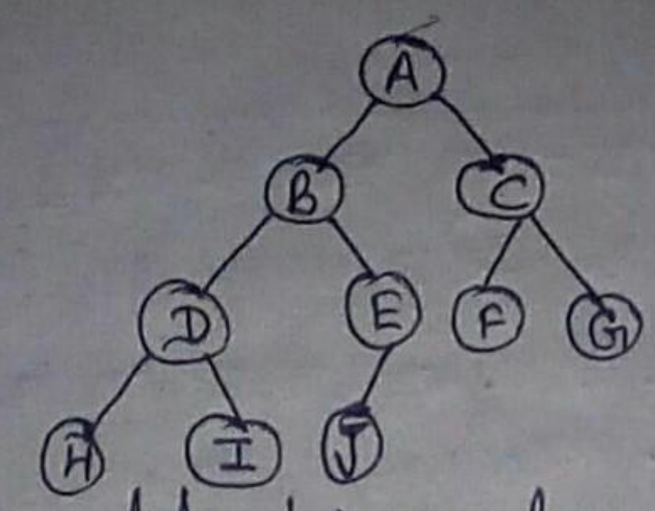


fig. Almost complete binary tree but not strictly binary tree.
Since node E has a left child but not a right child.

⊛ Operations on binary trees:

→ Father(n,T): Return the parent node of the node n in tree T.
If n is the root, Null is returned.

→ LeftChild(n,T): Return the left child of node n in tree T. Return NULL if n does not have a left child.

→ RightChild(n,T): Return the right child of node n in tree T.
Return NULL if n does not have a right child.

→ Info(n,T): Return information stored in node n of tree T.

→ Sibling(n,T): Return the sibling node of node n in tree T.
Return NULL if n has no sibling (Sibling meaning as childs of same parent)

→ Root(T): Return root node of tree if and only if the tree is nonempty.

→ Size(T): Return the number of nodes in tree T.

→ MakeEmpty(T): Create an empty tree T.

⊛. C-representation for binary tree:-

The structure of binary tree is as below:-

```
struct bnode
{ int info;
    struct bnode *left;
    struct bnode *right;
};
struct bnode *root=NULL;
```

⊛. Binary Tree Traversal:-

Traversing a tree means visiting each node in a specified order. The tree traversal is a way in which each node in a tree is visited exactly once in a symmetric manner. There are three popular methods of traversal which are as follows:-

1) Preorder traversal → In preorder traversing first root is visited followed by left sub-tree and right sub-tree. (i.e, RLR → First visit root then left, finally right). The preorder traversal of a nonempty binary tree is defined as follows:

→ Visit the root node.
→ Traverse the left sub-tree in preorder.
→ Traverse the right sub-tree in preorder.

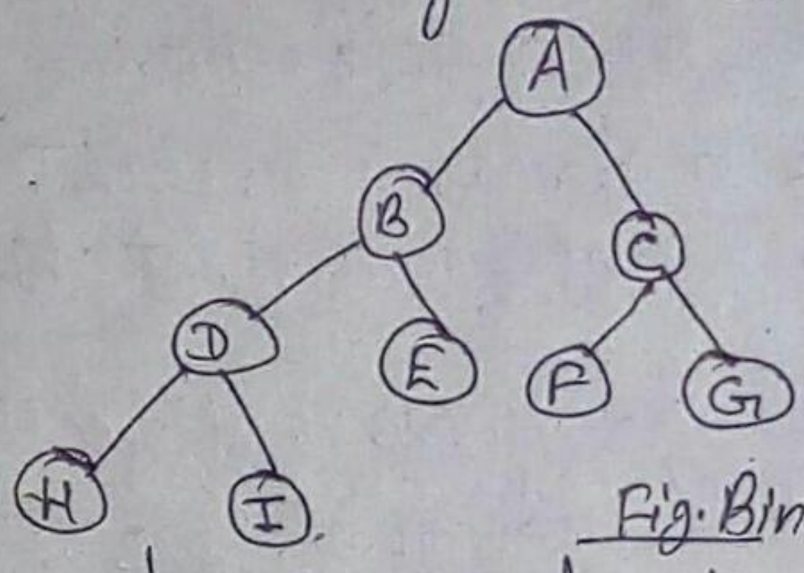


Fig. Binary Tree

The preorder traversal output of the given tree is: ABDHIECFG.

2) In-order traversal → The in order traversal of a non-empty binary tree is defined as follows:-

→ Traverse the left-subtree in in-order.
→ Visit the root node.
→ Traverse the right-subtree in in-order.

(i.e, LRR → first visit left, then root and finally right)
The in-order traversal of above binary tree is:- HDIBEAFCG.

3) Post-order traversal → The post-order traversal of a non-empty binary tree is defined as follows:-

→ Traverse the left sub-tree in post-order.
→ Traverse the right sub-tree in post-order.
→ Visit the root node.

(i.e, LRR → First visit left, then right and finally ~~node~~ root),
The post-order traversal of above binary tree is:- HIDEBFGCA.

⊛ Binary Search Tree (BST):-

A Binary Search Tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:

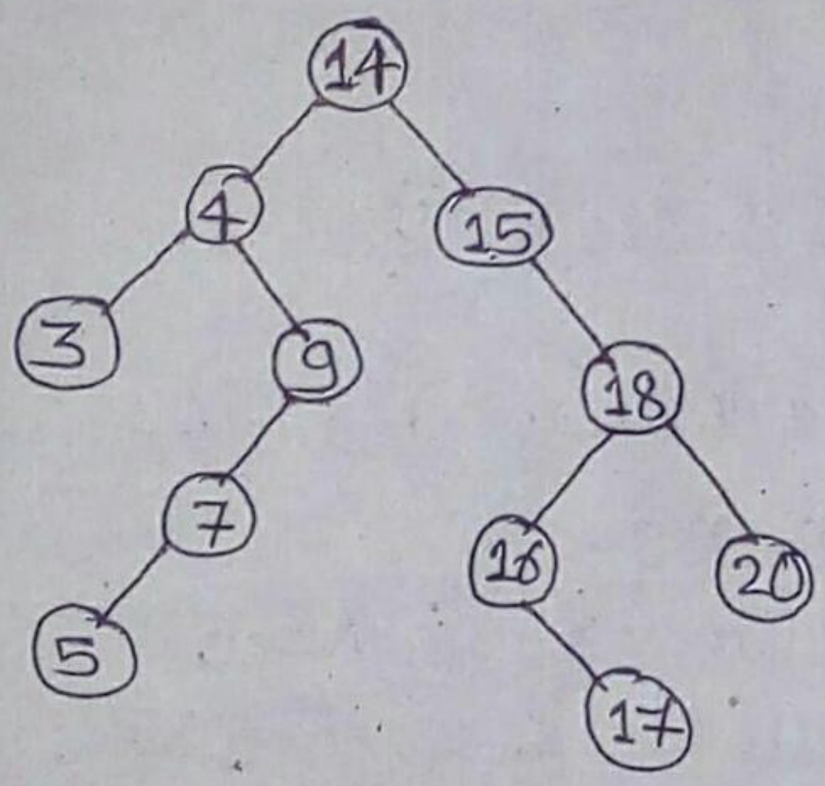
→ All keys in the left sub-tree of the root are smaller than the key in the root node.

→ All the keys in the right sub-tree are greater than the key in the root node.

→ The left and right sub-trees of the root are again binary search trees.

Example: Construct BST using the following given sequence of numbers: 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9.

Solution:



⊛ Operations on Binary Search Tree (BST):-

Following operations can be done in BST:

→ Search (k, T): Search for key k in the tree T. If k is found in some node of the tree then return true otherwise return false.

→ Insert (k, T): Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.

→ Delete (k, T): Delete a node with value k in the info field from the tree T such that the property of BST is maintained.

→ FindMin (T), FindMax (T): Find minimum and maximum element from the given non empty BST.

Algorithm for BST Searching:-

1. Start.
2. Check, whether value in current node and searched value are equal. If so, value is found. Otherwise;
3. If searched value is less, than the node's value:
   → If current node has no left child, searched value doesn't exist in the BST;
   → Otherwise, handle the left child with the same algorithm.
4. If searched value is greater, than the node's value:
   → If current node has no right child, searched value doesn't exist in the BST;
   → Otherwise, handle the right child with the same algorithm.
5. Stop.

Algorithm for Insertion of a node in BST;

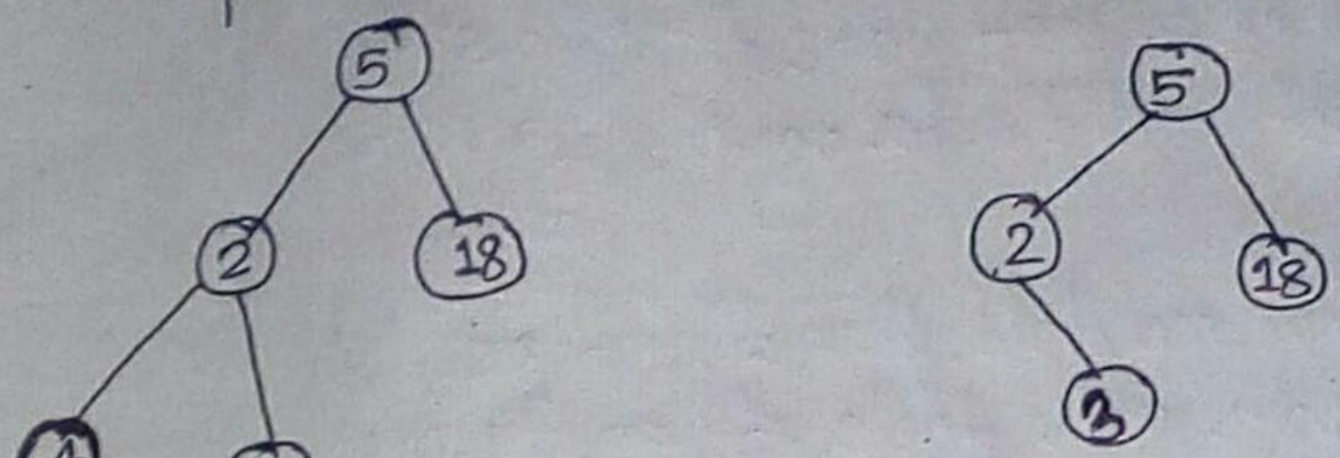
1. Start
2. Create a newNode with given value and set its left and right to NULL.
3. Check whether tree is Empty or not.
4. If the tree is Empty, then set root to newNode.
5. If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node. (here it is root node).
6. If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
7. Repeat the above step until we reach to a leaf node. (i.e, reach to NULL).
8. After reaching a leaf node, then insert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.
9. Stop.

⊛. Deleting a node from the BST:-

While deleting a node from BST, there may be three cases:

1. The node to be deleted may be a leaf node:

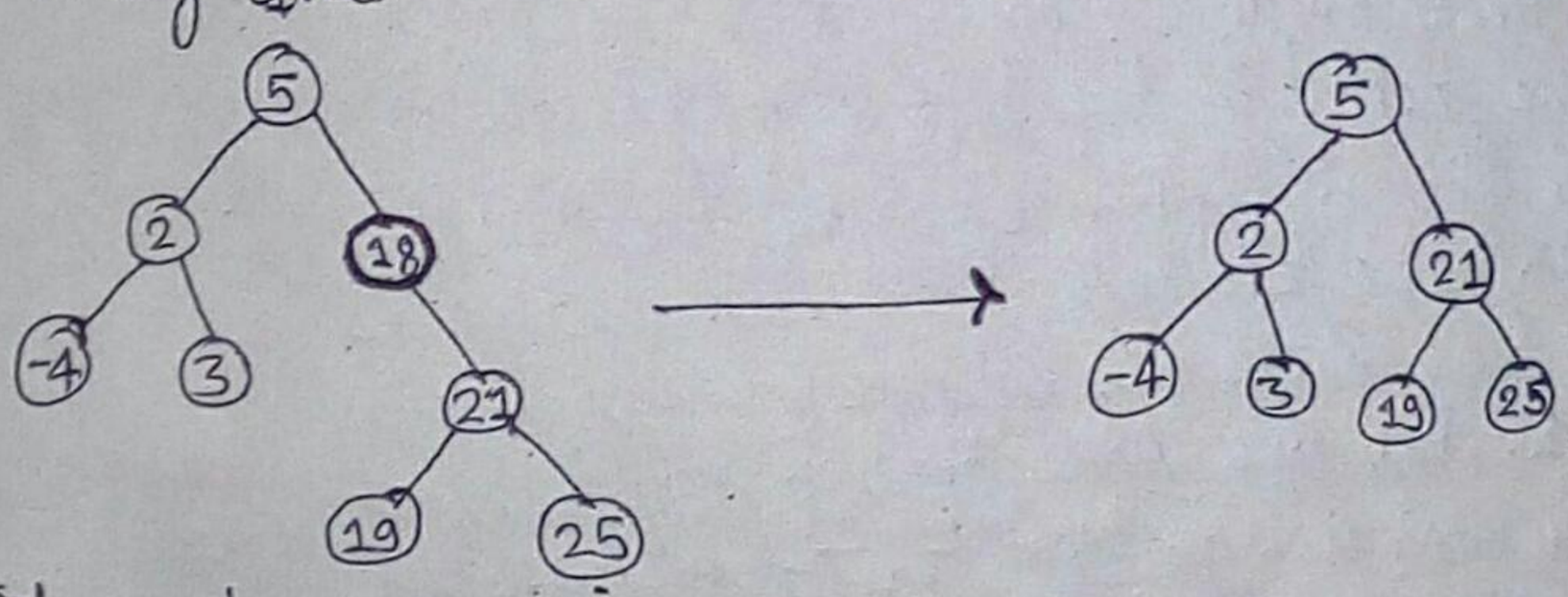
In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.



Suppose the node to be deleted is -4. This type of case.

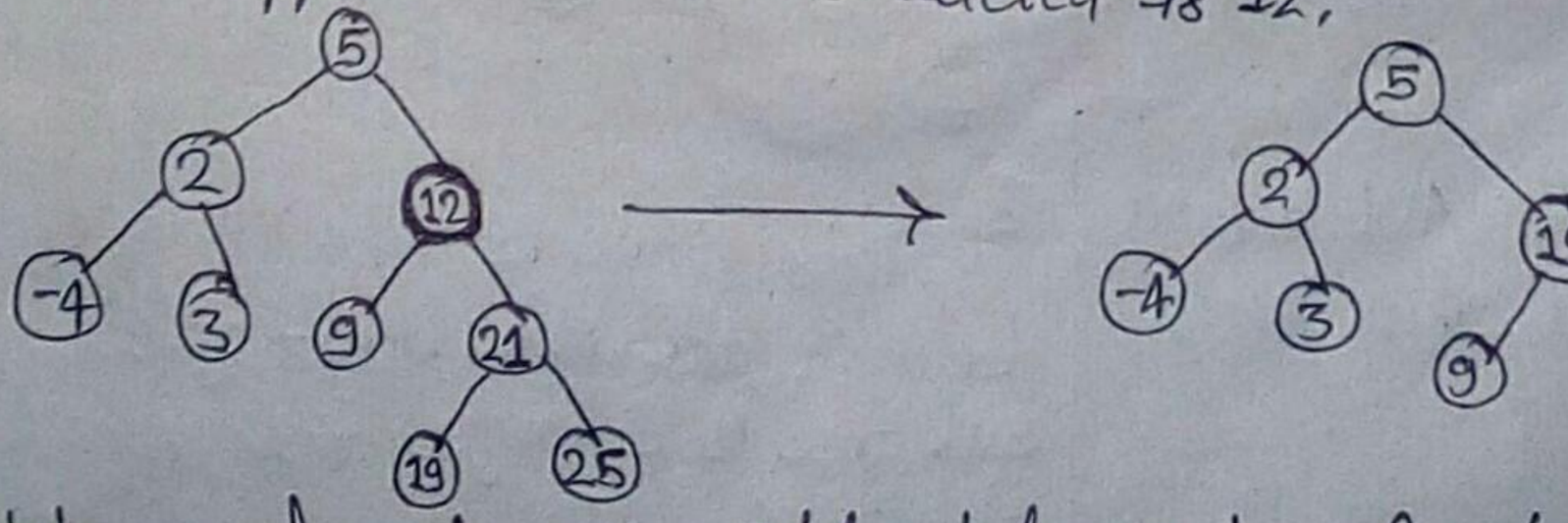
2. The node to be deleted has one child:

In this case the child of the node to be deleted is appended to its parent node. Suppose node to be deleted is 18 in following binary tree:



3. The node to be deleted has two children:

In this case the node to be deleted is replaced by its in-order successor node. OR the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node. Suppose node to be deleted is 12.



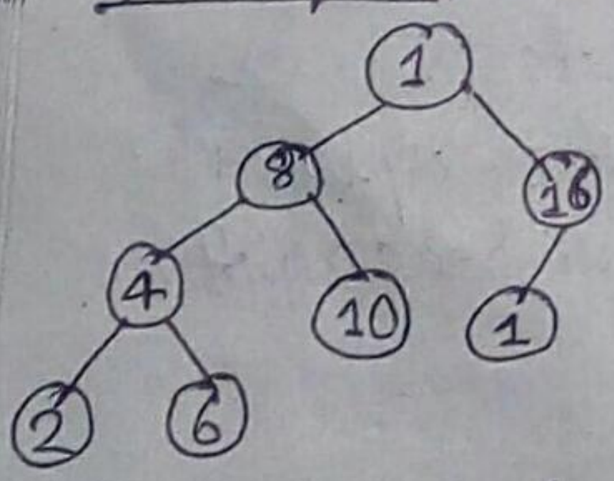
Find minimum element in the right sub-tree ~~to be~~ of node to be removed.

General Algorithm to delete a node from a BST:-

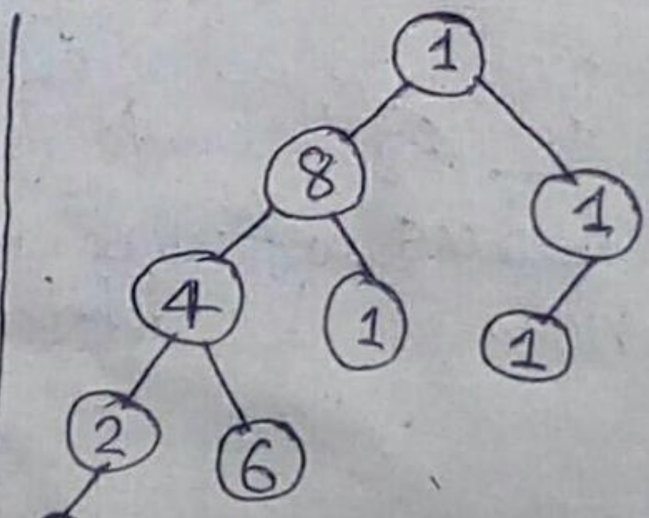
1. Start
2. If a node to be deleted is a leaf node at left side, then simply delete and set null pointer to its parent's left pointer.
3. If a node to be deleted is a leaf node at right side, then simply delete and set null pointer to its parent's right pointer.
4. If a node to be deleted has one child, then connect its child pointer with its parent pointer and delete it from the tree.
5. If a node to be deleted has two children, then replace the node being deleted either by a right most node of its left sub-tree or left most node of its right sub-tree (replace their in-order successor node).
6. End.

Q. AVL Tree: The first balanced binary tree is the AVL tree. AVL tree checks the height of the left and right sub-trees and assures that the difference is not more than 1. The difference is called the balance factor. An AVL tree is a binary search tree where the balance number at each node is -1, 0, or 1. For an AVL tree of height H, we find that it must contain at least $F_{H+3} - 1$ nodes. (F_i is the ith Fibonacci number).

Example:-



This figure satisfies AVL balance condition and thus it is an AVL tree.



This figure is not a AVL tree because height of left sub tree is 2 larger than right sub trees.

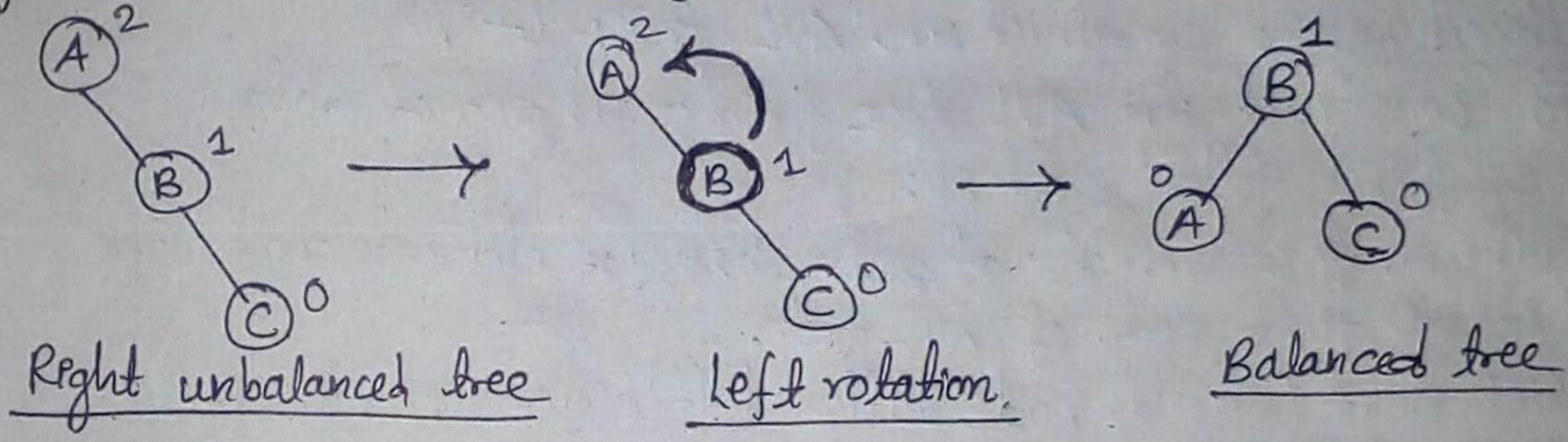
⑧. Making non AVL tree to AVL tree:-

We know that,

Balance Factor = height of left sub-tree − height of right sub-tree

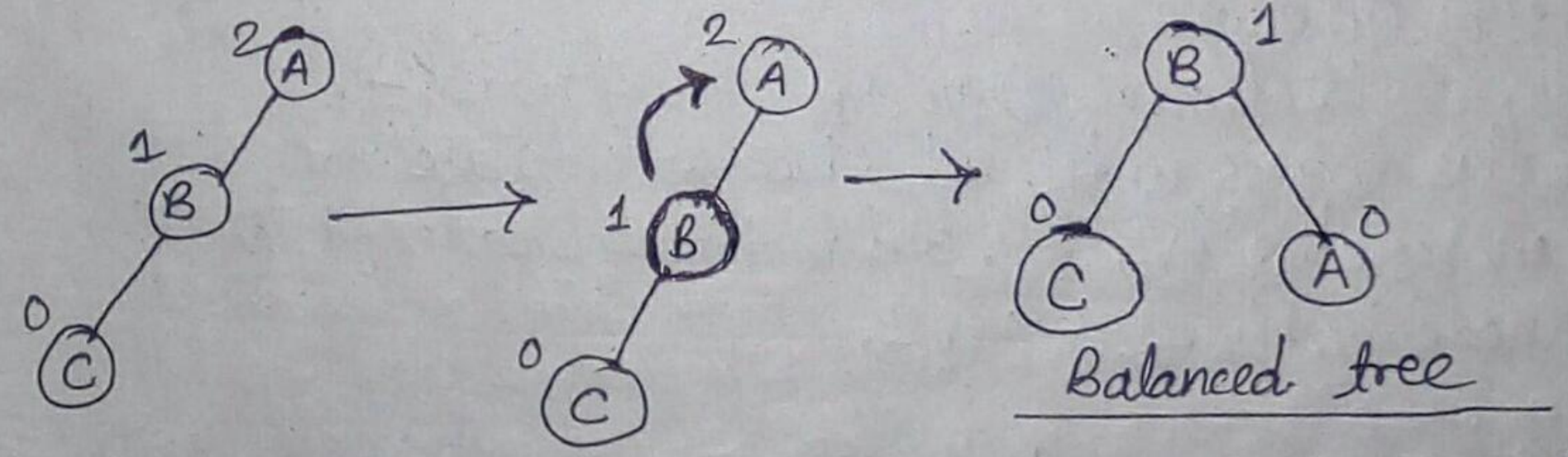
If the difference in the height of left and right sub-trees is more than 1 or, ~~then~~ less than -1 then the tree is unbalanced and we need to make height balance by using some rotation techniques. as:

i) **Left rotation** → If a tree becomes unbalanced, when a node is inserted into the right subtree ~~of the~~ then we perform single left rotation as below;



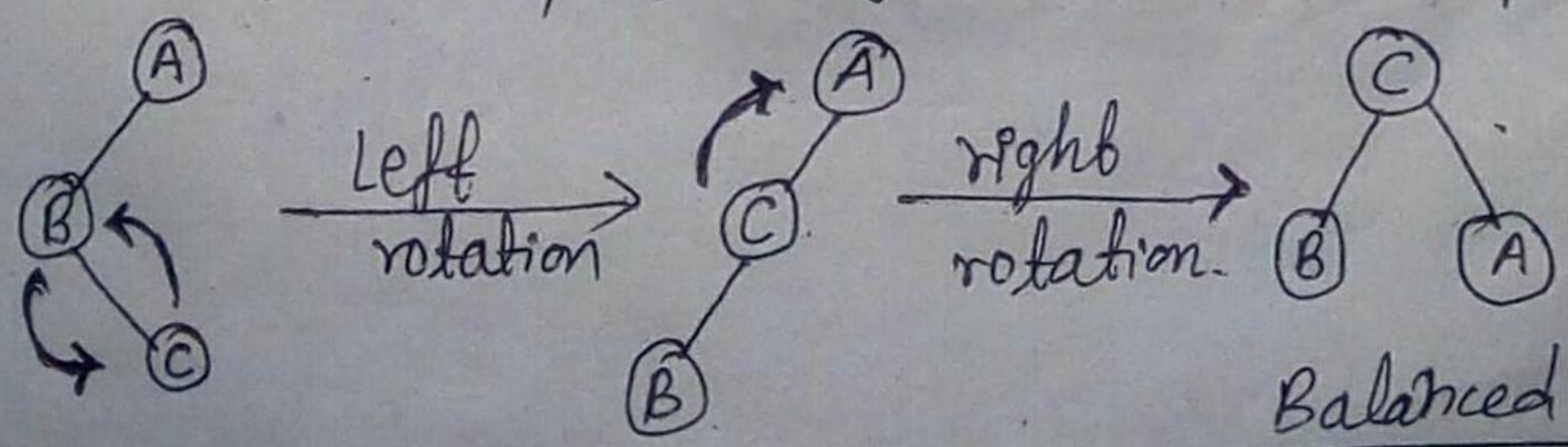
Right unbalanced tree → Left rotation → Balanced tree

ii) **Right rotation** → If a tree becomes unbalanced, when a node is inserted into the left sub-tree then we perform single right rotation as below;



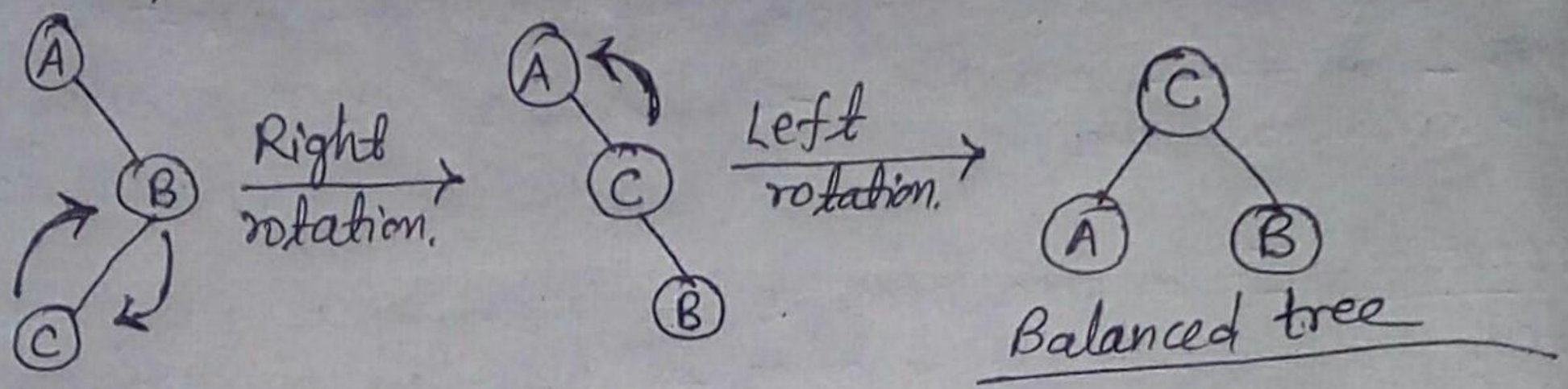
Balanced tree

iii) **Left-Right rotation** → If a tree becomes unbalanced, when a node is inserted into the right of left sub-tree, then we use double rotation: first we perform left rotation then we perform right rotation



Balanced tree

Right-left rotation → If a tree becomes unbalanced when a node is inserted into the left of right sub-tree then we use double rotation: first we perform right rotation then we perform left rotation as below:



Balanced tree

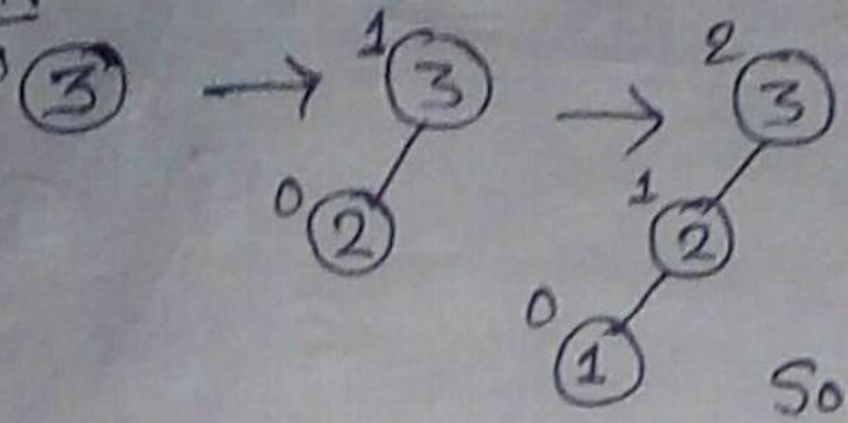
Procedure to construct an AVL tree:- (Imp)

To construct an AVL tree from given set of data points we follow following steps:

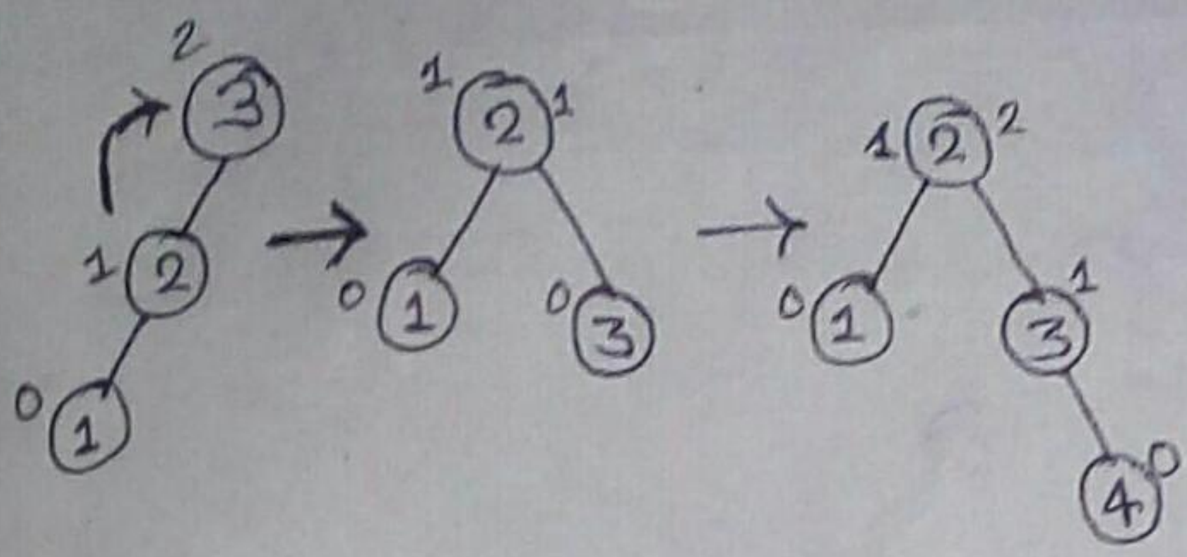
1. Take a first element of given array of elements and make it as first node of AVL tree.
2. Set next (second) node either left side or right side of given AVL tree.
   a). If node (1) > node (2) then,
   set node (2) in left side of node (1).
   b). Otherwise,
   set node (2) in right side of node (1).
3. Continue this process until all the elements are not included in resulting AVL tree and maintain balance factor for each node either −1 or 0 or 1.
4. If balance factor of a particular node is not given in range (−1 to 1) then rotate either single or double.
   a). If straight path take place, then perform single rotation.
   b). Otherwise perform double rotation.

Example:- Construct AVL tree from following data sets: {3,2,1,4,5,6,7,16, 15,14}.

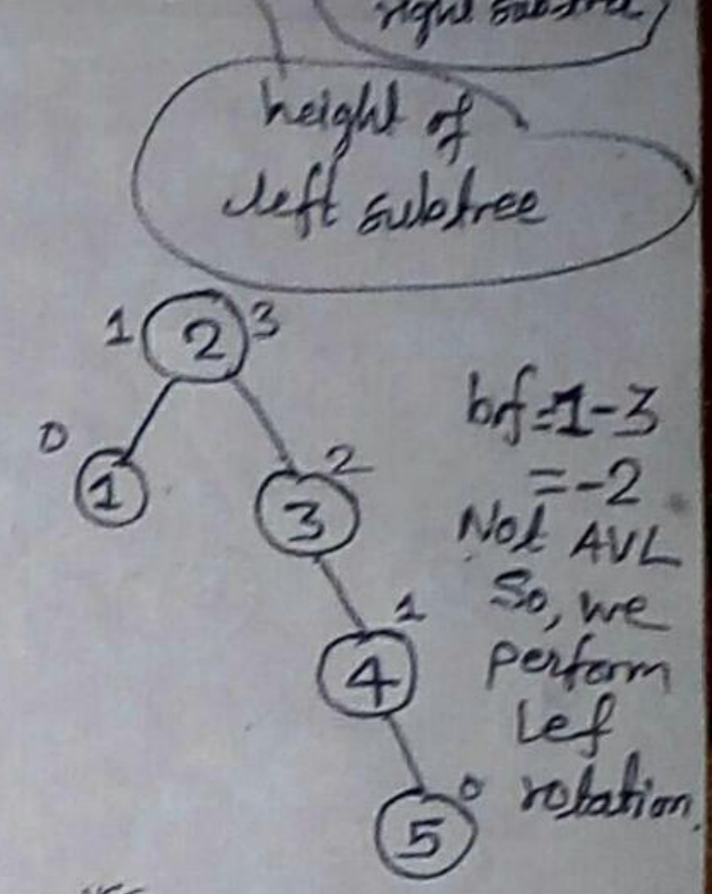
Solution:



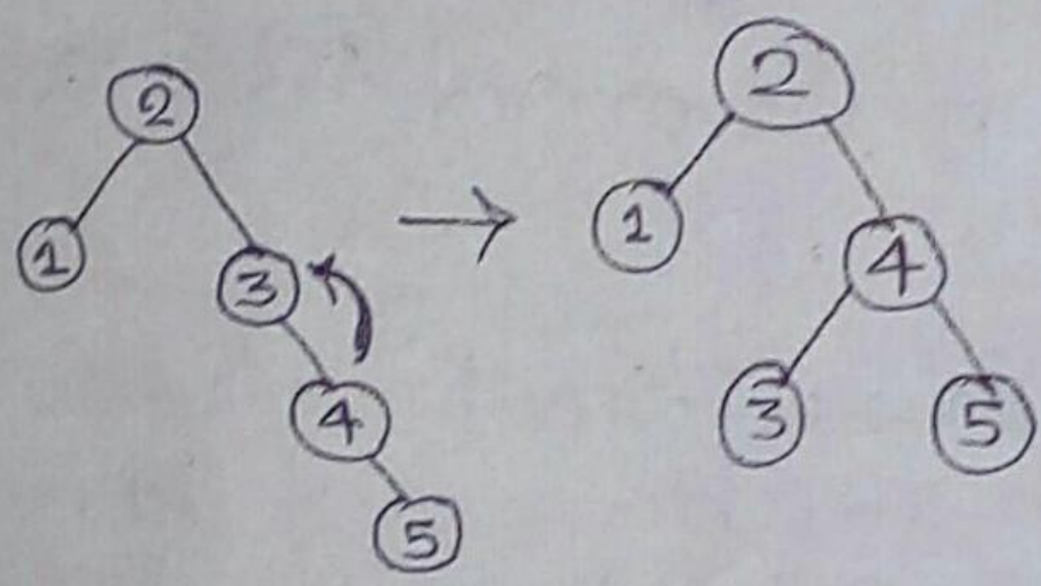
This is non AVL since balance factor is not in the rangle (-1 to 1)
since balance factor = 2 - 0 = 2
So. perform right rotation.



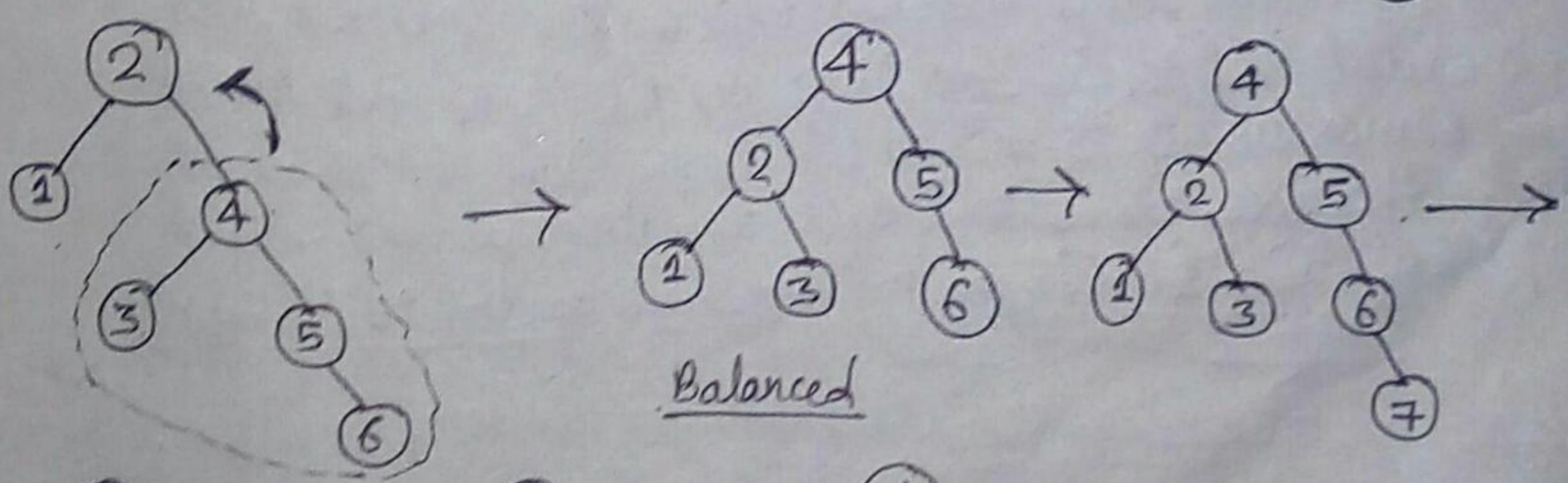
Here, balance factor = 1-2 = -1
So, AVL
Hence no need of rotation.



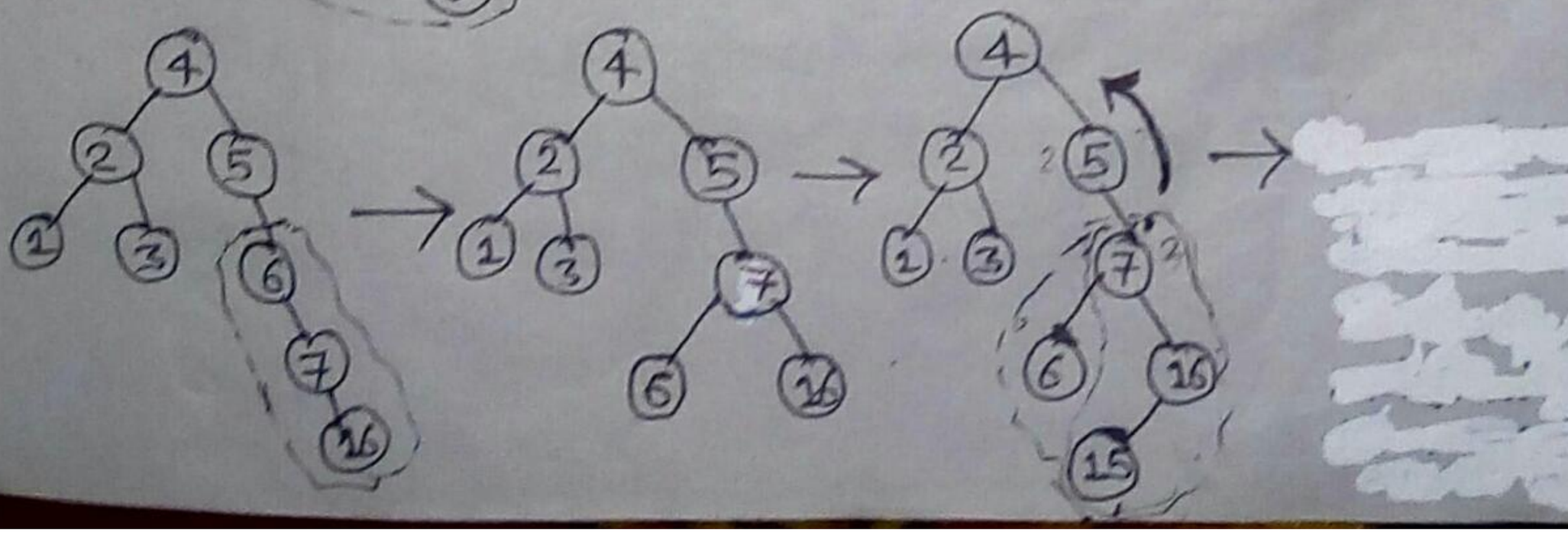
bof = 1-3 = -2
Not AVL
So, we perform lef rotation.

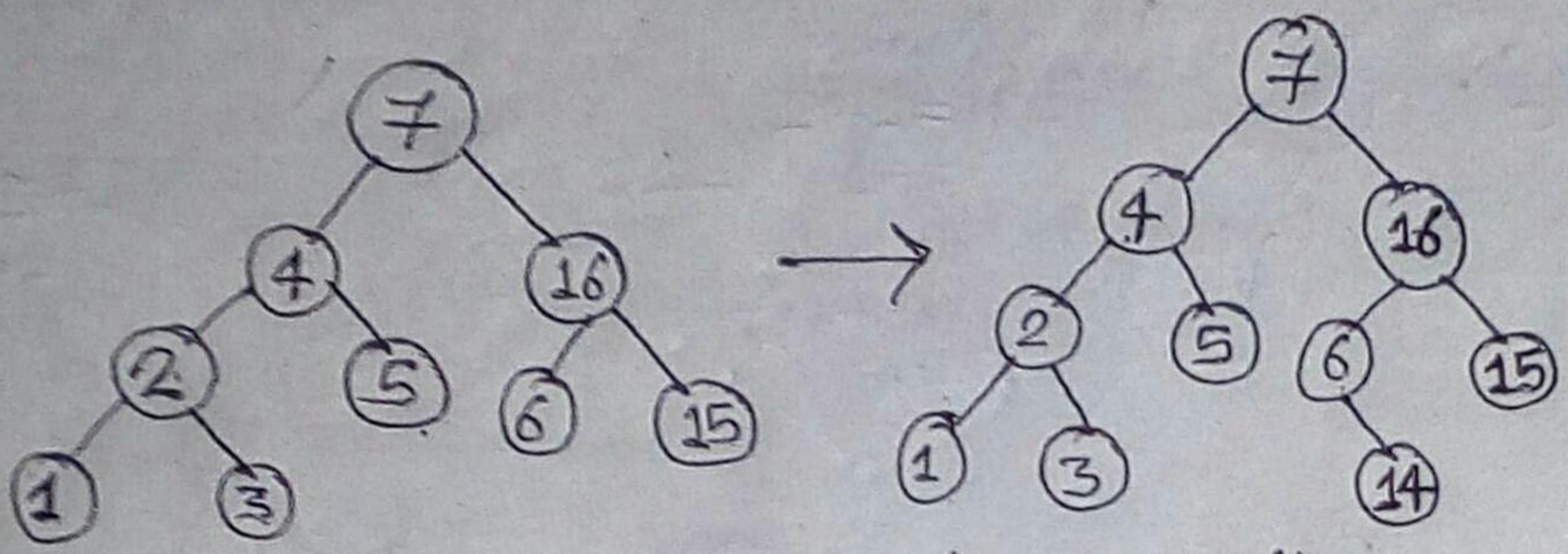


Since at node 4
b.f = 2-0 = 2
Non AVL



Balanced





This is the required AVL Tree.

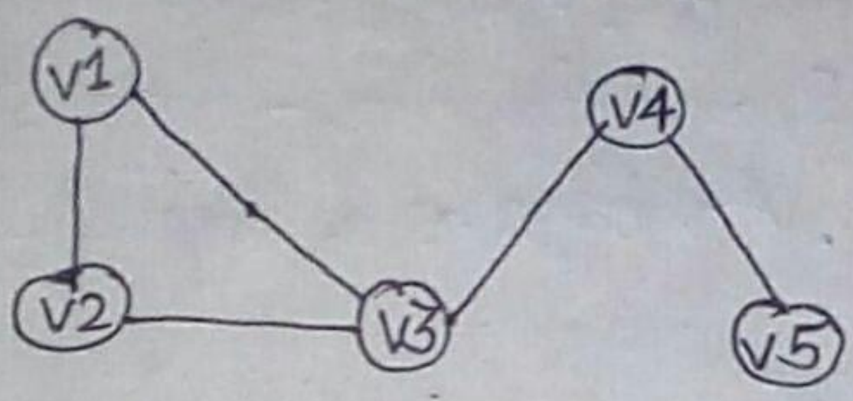
⊛ Time Complexity of AVL Tree:-

The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is $O(h)$ where h is height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So time complexity of AVL insert is $O(\log n)$.

B) Graphs:

A graph as a pair $G=(V, E)$ where V denotes a set of vertices and E denotes the set of edges connecting two vertices. Many natural problems can be explained using graph for example modeling road network, electronic circuits etc.

Let us take a graph:



V(G) = set of vertices of Graph = {v1, v2, v3, v4, v5}.

E(G) = set of edges of Graph = {(v1, v2), (v2, v3), (v1, v3), (v3, v4), (v4, v5)}

⊛ Types of Graph:

1) Simple Graph → We define a simple graph as 2-tuple consists of a non empty set of vertices V and a set of unordered pairs of distinct elements of vertices called edges. We can represent graph as $G=(V, E)$. This kind of graph has no loops and can be used for modeling networks that do not have connection to themselves but have both ways connection when two vertices are connected but no two vertices have more than one connection.

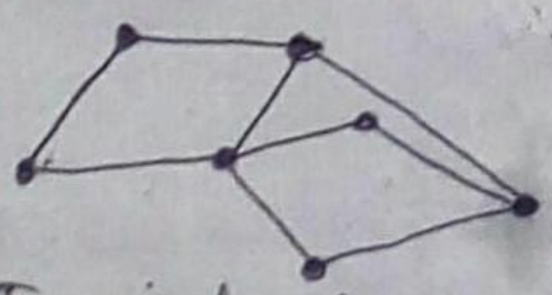


fig. Example of simple graph

2) Multigraph → A multigraph $G=(V, E)$ consists of a set of vertices V, a set of edges E, and a function f from E to {(u, v) such that $u, v \in V$, $u \neq v$ i.e, does not contain loop}. The edges e_1 and e_2 are called multiple or parallel edges if

$f(e_1) = f(e_2)$. In this representation of graph also loops are not allowded.

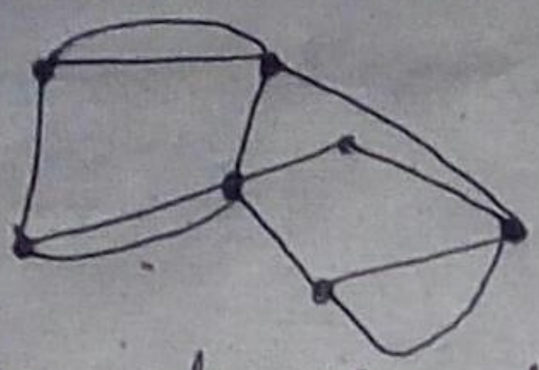


fig. Example of multigraph.

iii) Pseudograph → A pseudograph $G=(V, E)$ consists of a set of vertices V, a set of edges E, and function f from E to $\{(u,v)$ such that $u,v \in V\}$. An edge is a loop if $f(e)=(u,u)=u$ for some $u \in V$.

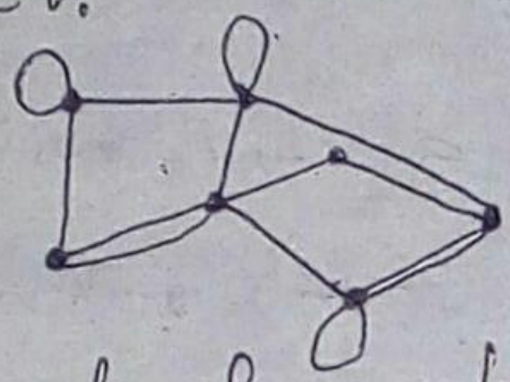


fig. Example of pseudograph

iv) Directed graph → A directed graph $G=(V,E)$ consit of a set of vertices, a set 'E' of edges that are ordered pairs of elements of V. In this graph loop is allowded but no two vertices have multiple edges in same direction.

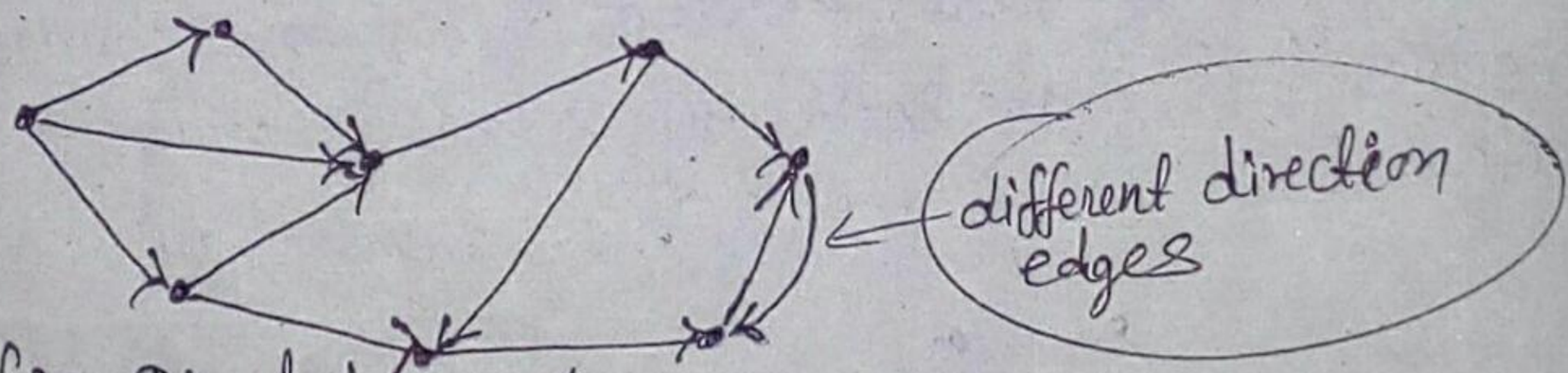


fig. Directed graph

v) Directed multigraph → Directed multigraph is similar to directed graph, the difference is that two vertices may contain multiple edges in same direction.

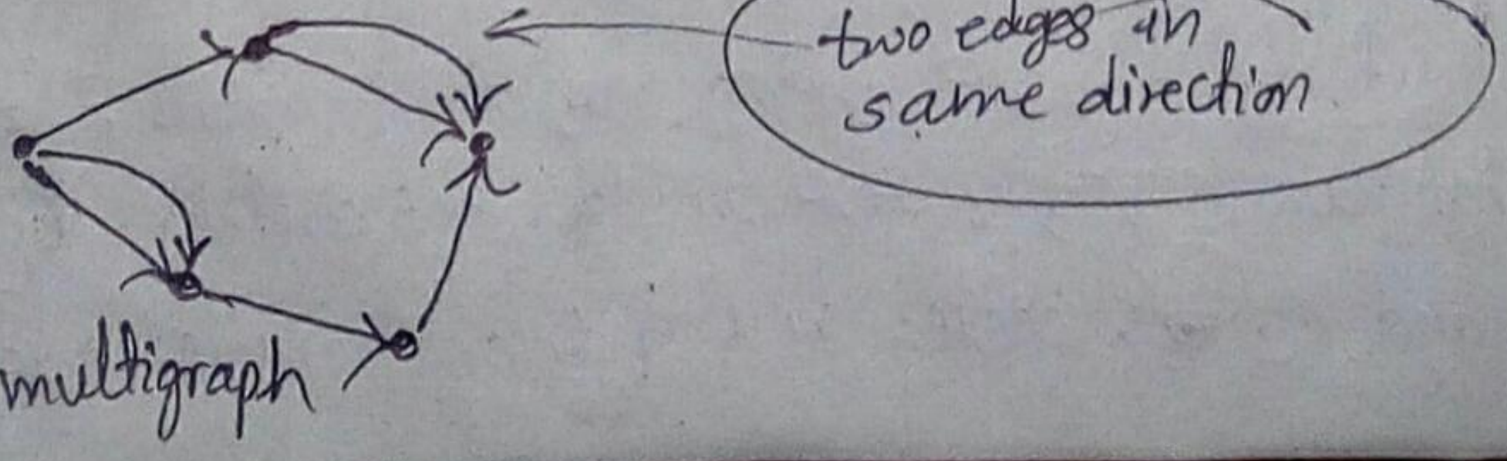


fig. Directed multigraph

⊛. Graph Representations:-

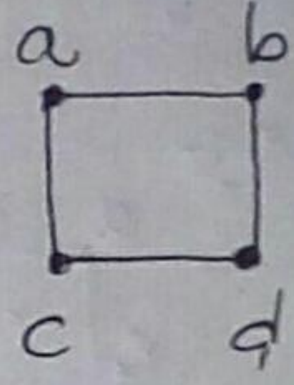
Generally graph can be represented in two ways namely adjacency lists (linked list representation) and adjcency matrix.

1. Adjacency list: This type of representation is suitable for the undirected graphs without multiple edges and directed graphs. This representation looks as in the table below:-

ⓐ For undirected Graph

Edge list for Simple Graph	
Vertex	Adjcent vertices
a	b, c
b	a, d
c	a, d
d	b, c

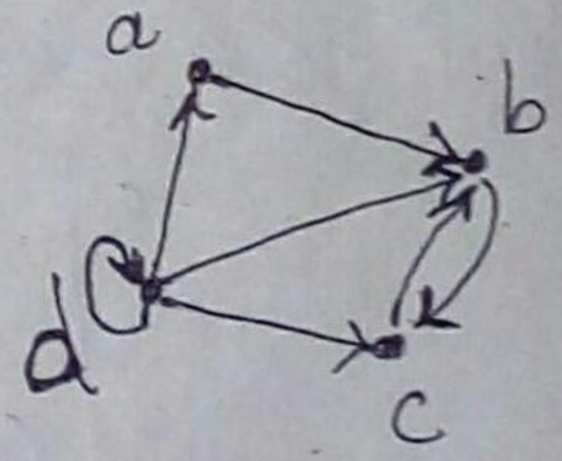
→



ⓑ. For Directed Graph

Edge List for Directed Graph	
Initial vertex	End Vertex
a	b
b	c
c	b
d	a, b, c, d

→



2. Adjcency Matrix: Given a simple graph $G=(V,E)$ with $|V|=n$. Assume that the vertices of the graph are listed in some arbitrary order like $v_1, v_2, \ldots, v_n$. The adjacency matrix A of graph G, with respect to the order of the vertices is n-by-n, zero-one matrix ($A=[a_{ij}]$) with the condition,

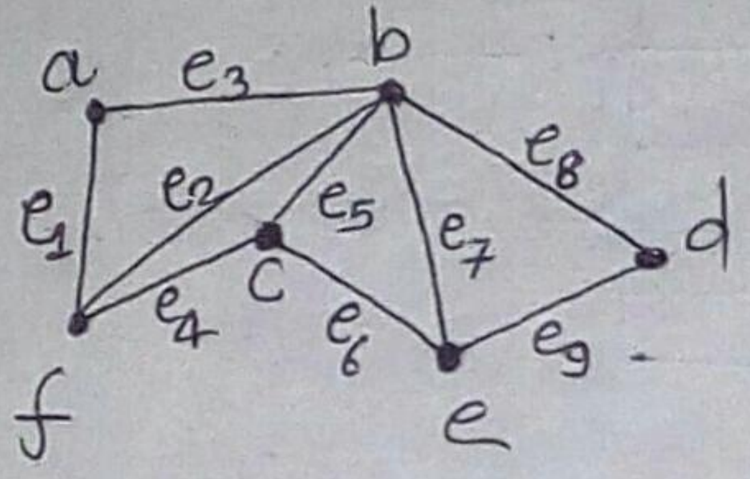
$$a_{ij} = \begin{cases} 1 \text{ if } \{v_i, v_j\} \text{ is an edge of } G. \\ 0 \text{ otherwise.} \end{cases}$$

Since there are n vertices and we may order vertices in any order, there are n! possible order of the vertices. The adjacency matrix depends on the order of vertices, hence there are n! possible adjacency matrices for a graph with n vertices.

⇒ In case of directed graph we can extend the same concept as in undirected graph as indicated by relation;

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G_1. \\ 0 & \text{otherwise.} \end{cases}$$

Example:-



Solution:-

Let the order of the vertices be a, b, c, d, e, f.

There are 6 vertices, so we construct 6×6 matrix as follows:-

	a	b	c	d	e	f
a	0	1	0	0	0	1
b	1	0	1	1	1	1
c	0	1	0	0	1	1
d	0	1	0	0	1	0
e	0	1	1	1	0	0
f	1	1	1	0	0	0

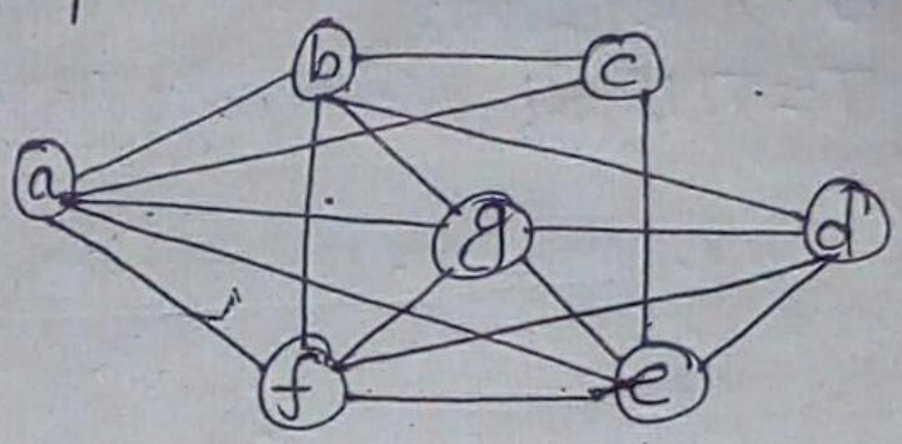
6×6.

Similarly we can construct for directed graph also.

⊛. Graph Traversals:-

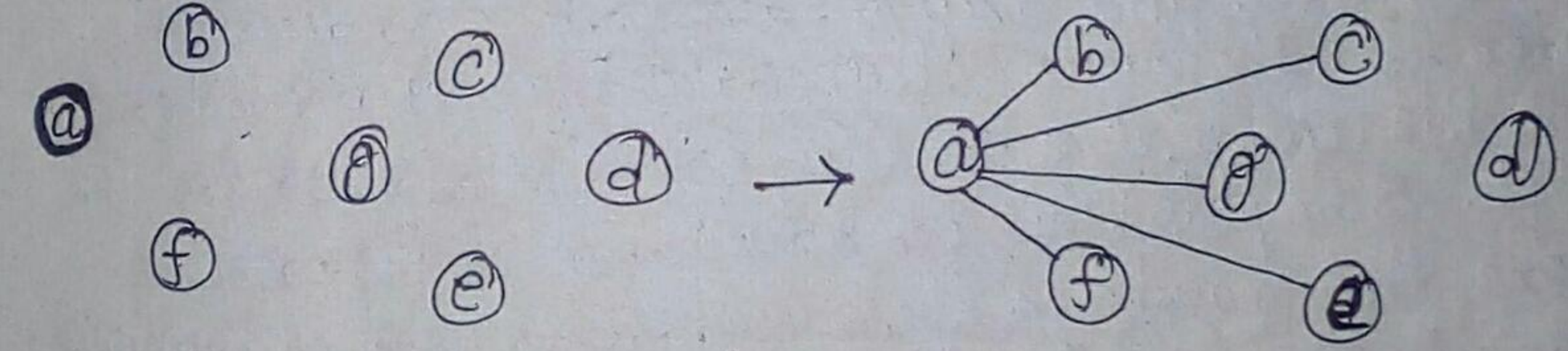
1). Breadth-First Search (BFS): This is one of the simplest methods of graph searching. Choose some vertex arbitrarily as a root. Add all the vertices and edges that are incident in the root. The new vertices added will become the vertices at the level 1 of the BFS tree. From the set of the added vertices of level 1, find other vertices, such that they are connected by edges at level 1 vertices. Follow the above step until all vertices are added.

Example: Use Breadth-First Search to find a BFS tree of the following graph.

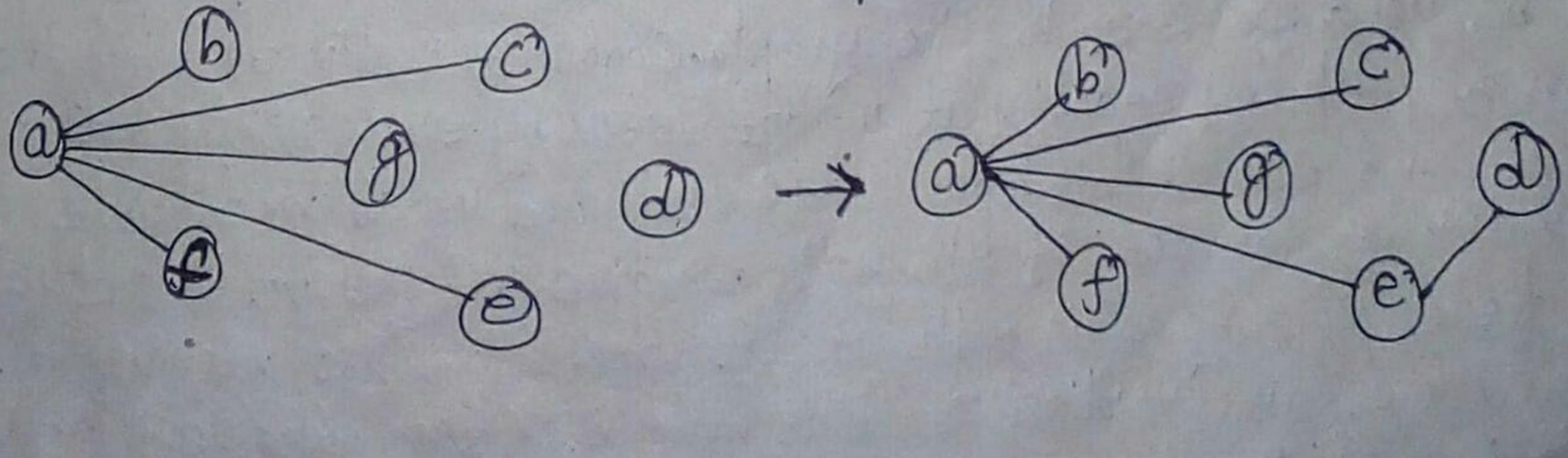


Solution:-

We choose 'a' as initial vertex then we have.



⇒ Order of vertices of level 1 i.e $\{b, c, g, e, f\}$.
Say now the order be $\{e, f, g, b, c\}$.



Algorithm For BFS

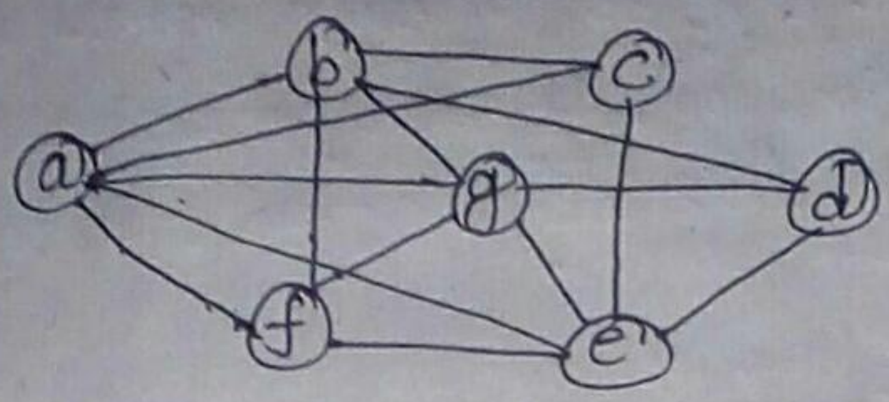
```
BFS(G,s) //s is start vertex.
{
    T={s};
    L=ϕ; //an emphy queue.
    Enqueue(L,s);
while (L!=ϕ)
    {
      v=dequeue(L);
      for each neighbour w to v
      if (w∉L and w∉T)
        {
          enqueue(L,w);
          T=TU{w}; //put edge {v,w} also.
        }

    }
}
```

Analysis:- From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjcent vertices are looked for and this can be done in O(n) time (for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue. i.e, n, produce complexity of an algorithm as O(n2).

2) Depth-First Search (DFS): This is an another technique that can be used to search the graph. Choose a vertex as a root and form a path by starting at a root vertex by successively adding vertices and edges. This process is continued until no possible path can be formed. If the path contains all the vertices then the tree consisting this path is DFS tree. Otherwise, we must add other edges and vertices. For this move back from the last vertex that is met in the previous path and find whether it is possible to find new path starting from the vertex just met.

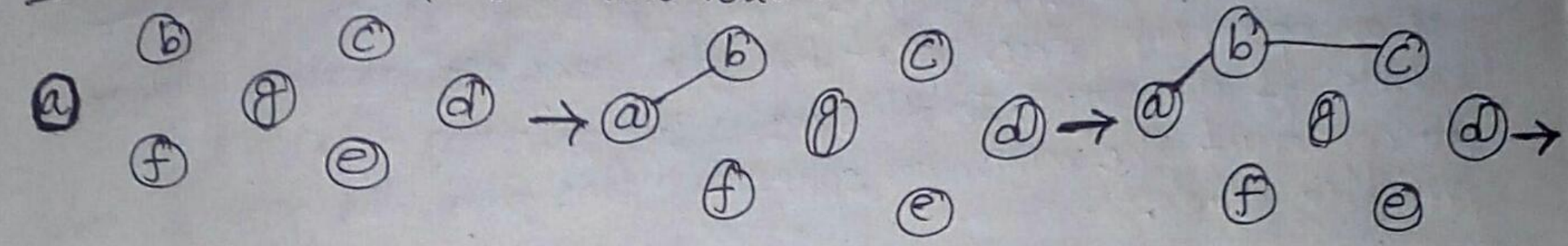
If there is such a path continue the above process. The whole process is continued until all the vertices are met. This method of search is also called backtracking.

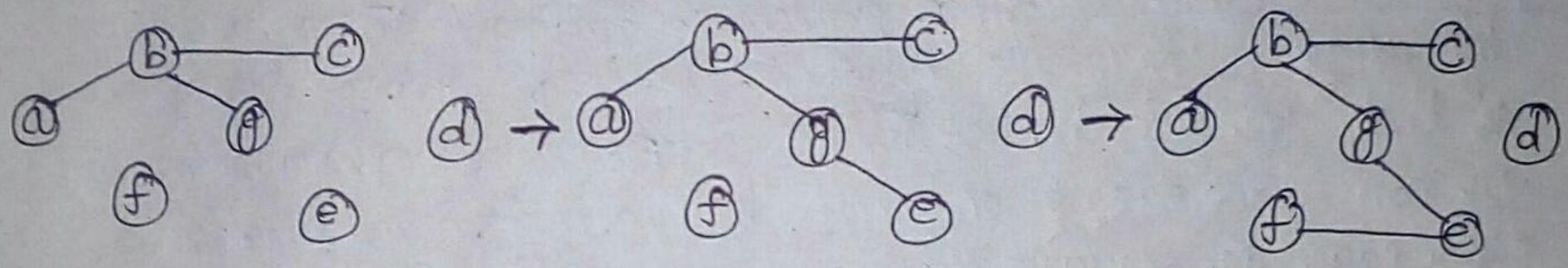
Example:

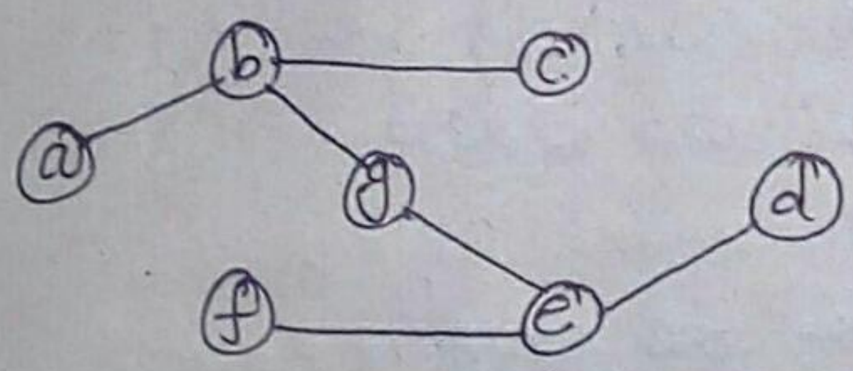


Use depth first search to find a spanning tree of this graph.

Solution: choose a as initial vertex:







This is the required spanning tree of graph by DFS.

Algorithm:

```
DFS (G,s)
  { T={s};
    Traverse (s);
  }
Traverse (v)
  { for each w adjcent to v and not yet in T
    { T=TU{w}; //put edge {v,w} also
      Traverse (w);
    }
  }.
```

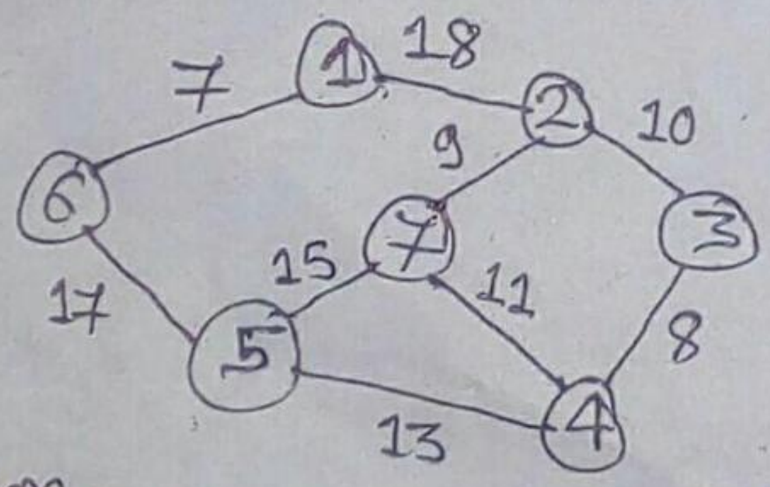
Analysis:
The complexity of the algorithm is greatly affected by Traverse function we can write its running time in terms of the relation $T(n)=T(n-1)+O(n)$, here $O(n)$ is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find the complexity of an algorithm is $O(n)$.

⊛. Minimum Spanning Trees:-

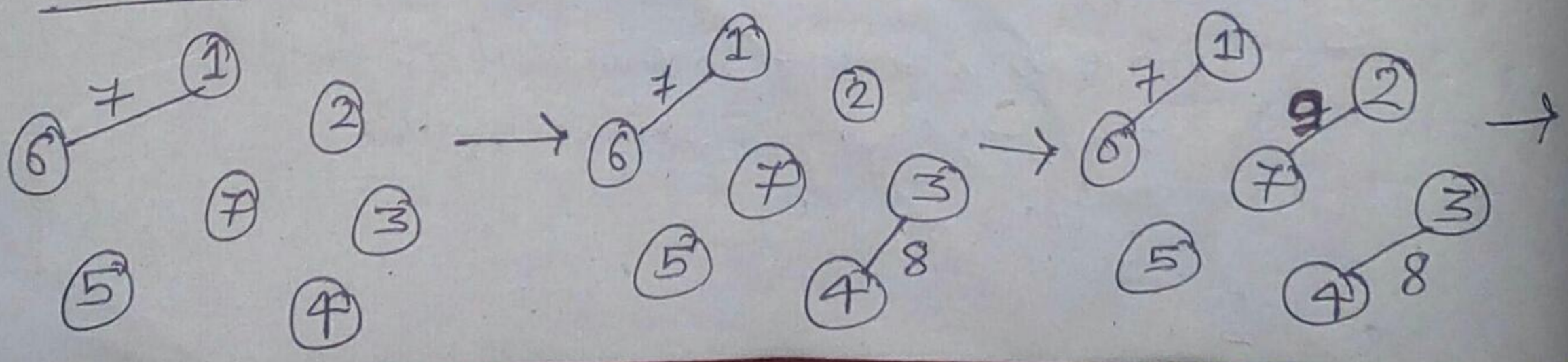
A minimum spanning tree is a connected weighted graph ~~in~~ is a spanning tree that has the smallest possible sum of weights of its edges. In this we study following algorithms used to construct the minimum spanning tree from the given connected weighted graph.

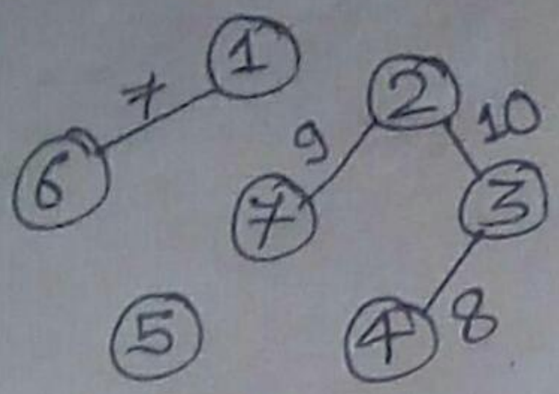
1) Kruskal's Algorithm:- The problem of finding MST can be solved by using Kruskal's algorithm. The idea behind this algorithm is that you put the set of edges from the given graph $G1=(V,E)$ in non decreasing order of their weights. The selection of each edge in sequence then guarentees that the total cost that would ~~from~~ form will be the minimum. Note that we have G1 as a graph, V as a set of n vertices and E as a set of edges of graph G1.

Example:- Find the MST and its weight of the graph.

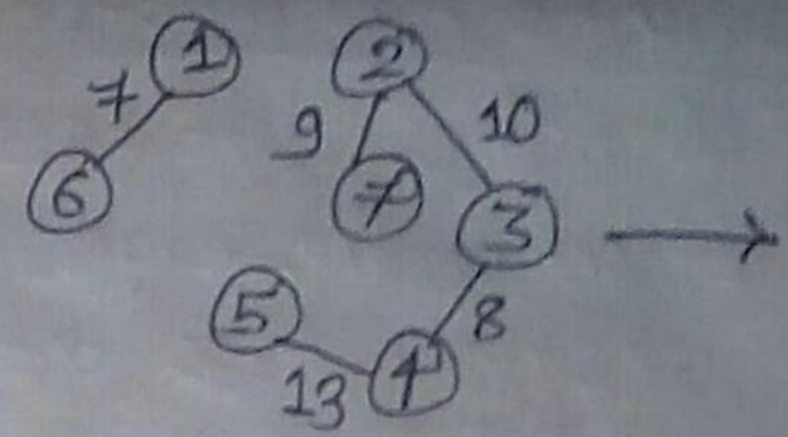


Solution:-

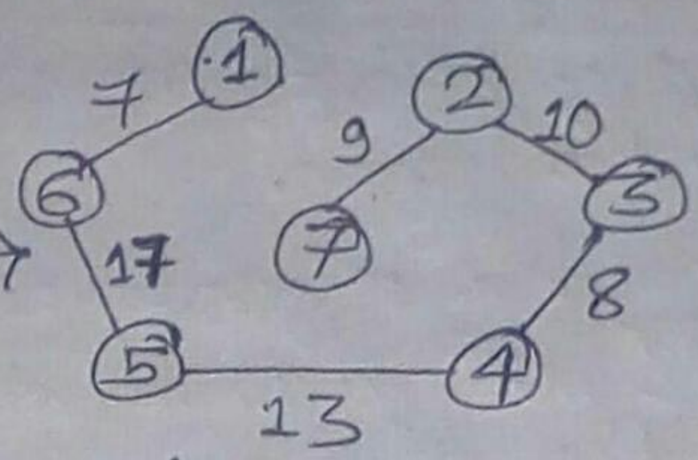




→ Edge with weight 11 forms cycle so discard it. →



Edge with weight 15 forms cycle so discard it →



Hence, the total weight of MST is 64.

Algorithm:

```
Kruskal MST(G)
{ T={V} // forest of n nodes.
  S= set of edges stored in non decreasing order of weight.
  while (|T|< n-1 and E!= ∅)
  {
    Select (u,v) from S in order.
    Remove (u,v) from E
    if((u,v). does not create a cycle in T)
        T= T.U{(u,v)}
  }
}
```

(Union of (u,v))

Analysis: In the above algorithm the n tree forest at the beginning takes (V) time, the creation of S take O(E log E) time and while loop execute O(n) times and the steps inside the loop take almost linear time. So the total time taken is O(E log E).

2) Prim's Algorithm:- It is a another minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which:

→ form a tree that includes every vertex.

→ has a minimum sum of weights among all the trees that can be formed from the graph.

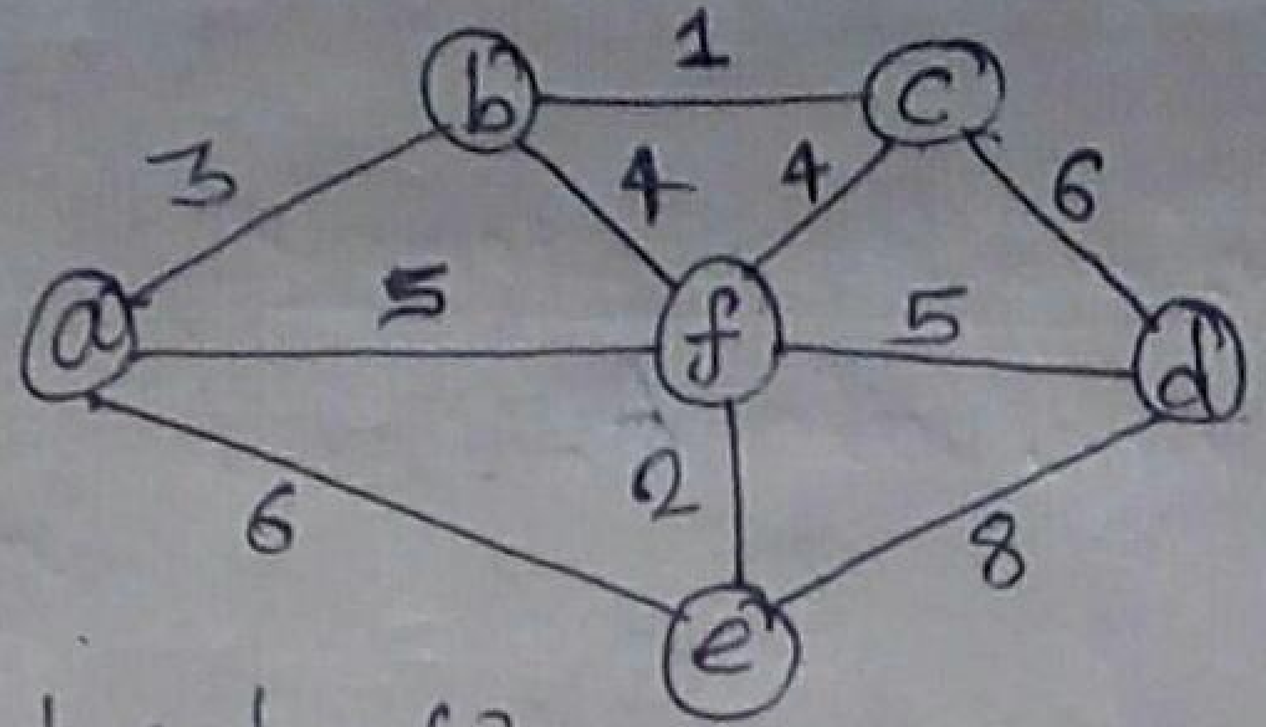
The idea behind this algorithm is just to take an arbitrary vertex and choose the edge with minimum weight incident on the choosen vertex. Add the vertex and continue the above process until all the vertices are not added to the list. Remember that the cycle must be avoided.

Pseudo code / Algorithm:-

```
PrimMST(G)
{ T = ϕ;    //T is a set of edges of MST
  S = {s};  //s is randomly choosen vertex and S is
              set of vertices.
while (S != V)
  {
    e = (u,v) an edge of minimum weight incident to
        vertices in T and not forming simple circuit
        in T if added to T i.e, u ∈ S and v ∈ V-S.
    T = T U {(u,v)};
    S = S U {v};
  }
}
```

Analysis:- In above algorithm while loop execute O(V). The edge of minimum weight incident on a vertex can be found in O(E), So the total time is O(EV). We can improve the performance of the above algorithm by choosing better data structures as priority queue and normally it will be seen that the running time of prism's algorithm is O(E log V)!.

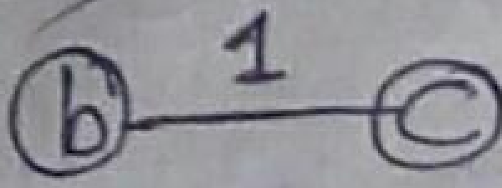
<u>Example 1</u>: Find the minimum spanning tree of given graph by using Prim's algorithm.



<u>Solution</u>:-

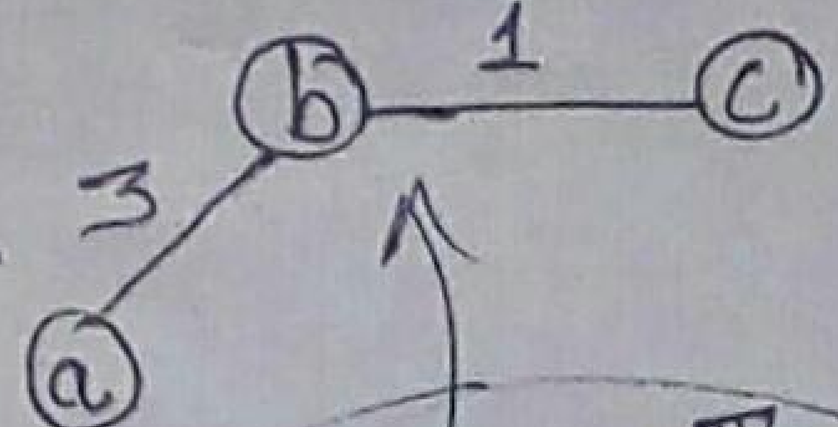
V = {a, b, c, d, e, f}

Let we start with vertex b.



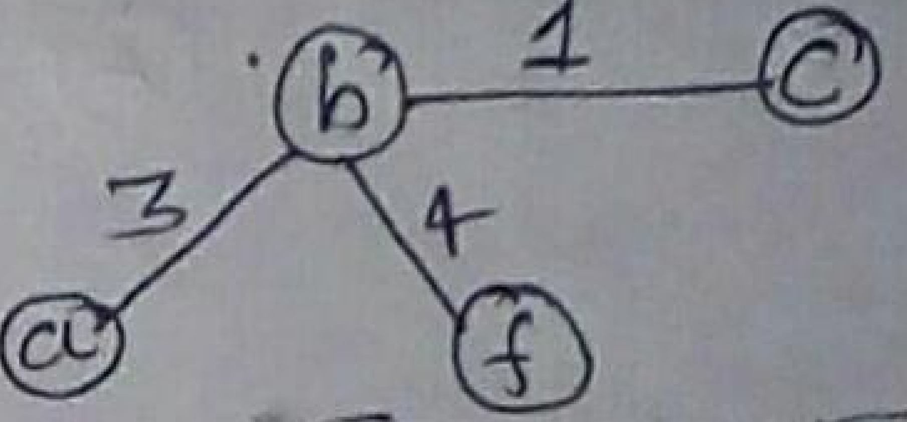
लिइएको vertex बाट minimum भएको edge हेर्नें

→



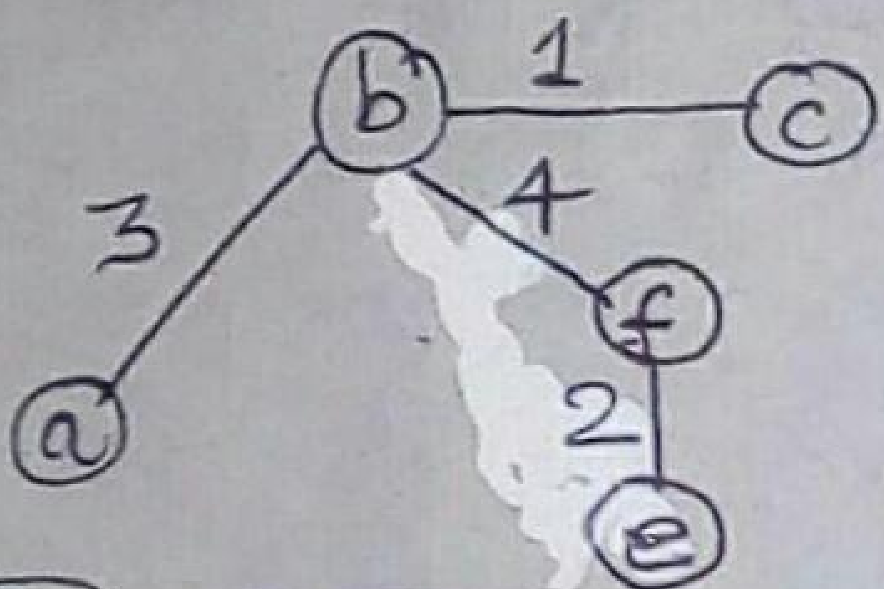
अब b, र c बाट minimum हेर्नें, continue this process.

→



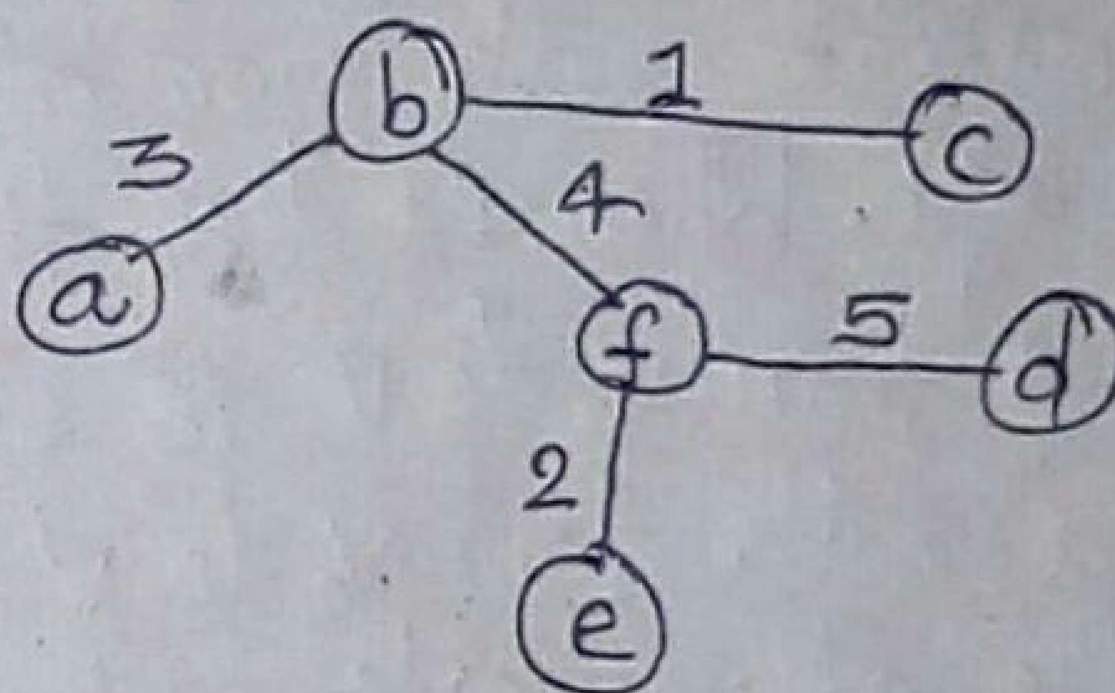
→

Here we look from a, b, c minimum. b to f and c to f are minimum so we can take any one of them. Let I took b to f.



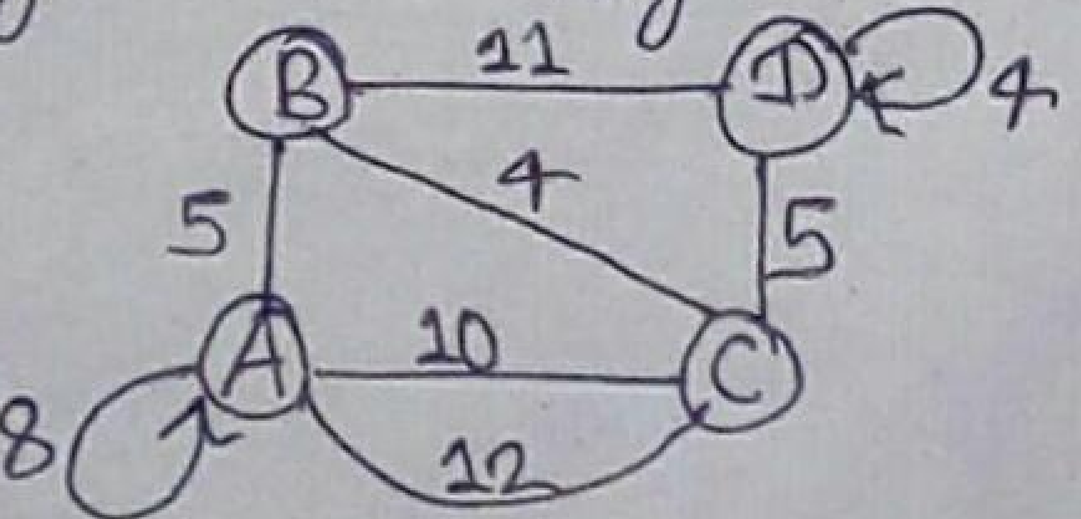
Now c to f is minimum but it is forming loop so we ~~loo~~ neglect it and take next minimum f to e.

→



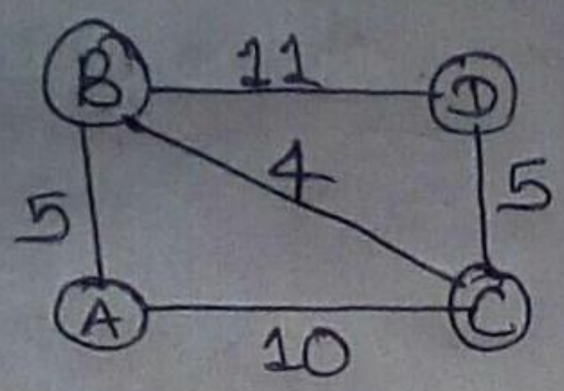
Hence the total MST = 15.

<u>Example 2</u>:- Find the minimum spanning tree of given graph by using Prim's algorithm.



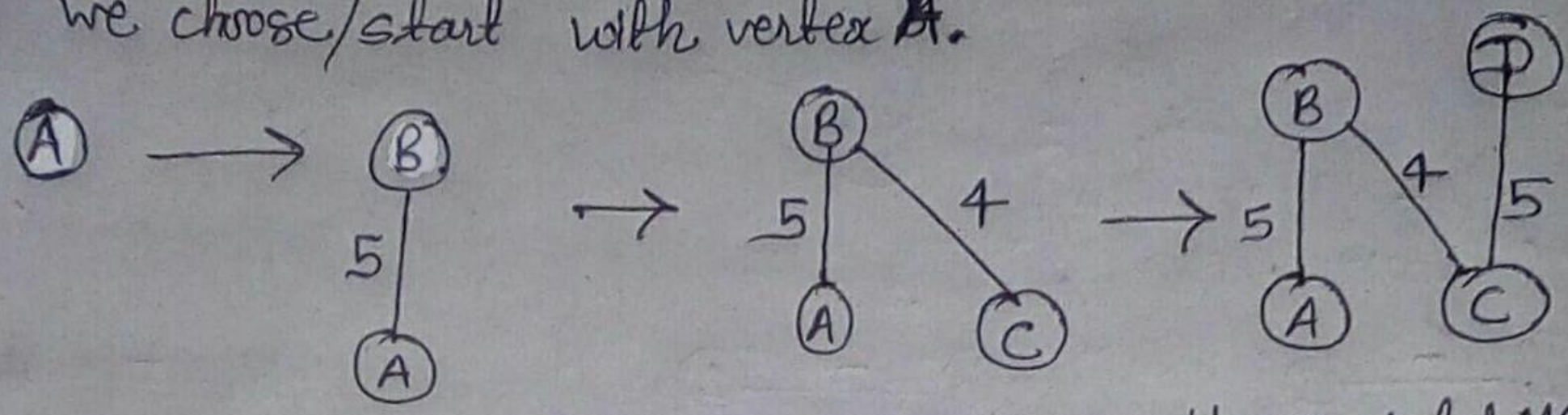
<u>Solution</u>:-

To create minimum spanning tree of this graph containing loop and parallel edge (i.e, two edges at AC) we have to first remove loops and also remove parallel edges ~~and proceed~~ whose weight is greater. Now the figure becomes:



Now we proceed as we did before example.

Let we choose/start with vertex A.



Hence total MST = 14

⊛ Shortest Path Algorithm: Dijkstra's Algorithm:-

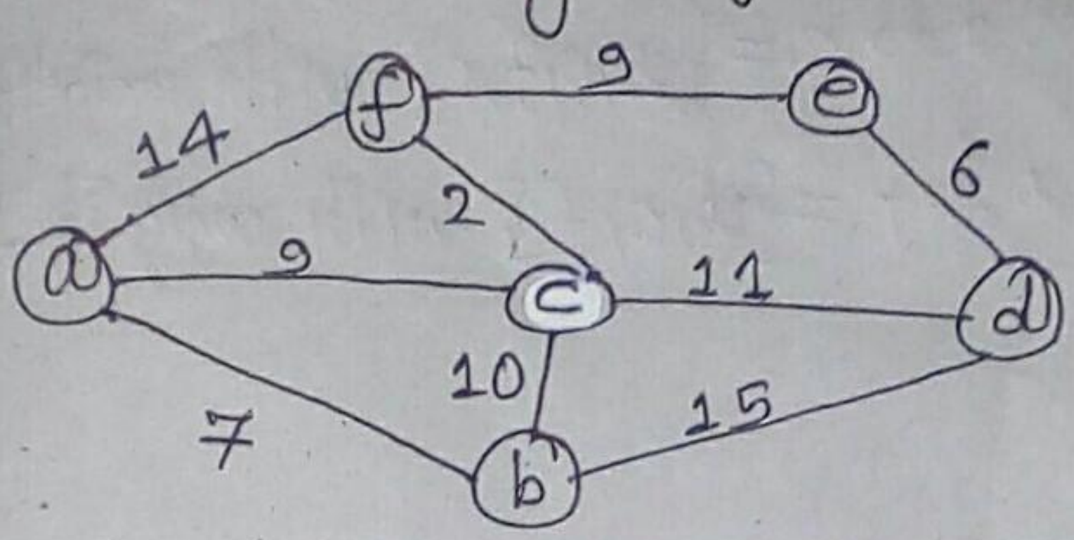
This is another approach of getting single source shortest paths. In this algorithm it is assumed that there is no negative weighted edge. Dijkstra's algorithm finds the shortest path from one vertex V_0 to each other vertex in a digraph. When it has finished, the length of the shortest distance from V_0 to V is stored in the vertex V, and the shortest path from V_0 to V is recorded in the back pointers of V and the other vertices along that path.

Algorithm:-

```
Dijkstra_Algorithm (G, w, s)
{ for each vertex v ∈ V
        d[v] = ∞
  d[s] = 0
  S = ∅
  Q = V
while (Q != ∅) {  u = Take minimum from Q and delete.
                  S = S ∪ {u}
                  for each vertex v adjcent to u
                   if d[v] > d[u] + w(u,v) then
                      d[v] = d[u] + w(u,v).
                  }
}
```

Analysis: In the above algorithm the first for loop block takes $O(V)$ time. Initialization of priority queue Q take $O(V)$ time. The while loop executes for $O(V)$ where for each execution the block inside the loop takes $O(V)$ times. Hence the total running time is $O(V^2)$.

Example: Find the shortest paths from the source node to all other vertices using Dijkstra's algorithm.



Solution:

Let source vertex be a, so we initialize initially source vertex 'a' with weight 0 and ∞ to all other remaning vertices. Now, we construct table for faster calculations.

Vertex selected ↓	a	b	c	d	e	f
a	[0]	∞	∞	∞	∞	∞
b		[7]	9	∞	∞	14
c			[9]	22	∞	14
f				20	∞	[11]
d				[20]	20	
e					[20]	

do other calculations on rough as below,

d[b] > d[a] + w[a,b]
∞ > 0 + 7
∞ > 7 [true]

So, d[b] = d[a] + w[a,b]
= 7.

Similarly for others

We construct now a solution table to find shortest path for each vertex as below.

Destination	a	b	c	d	e	f
Minimum Weight	0	7	9	20	20	11

Now, we have to find ~~minimum weighted path~~ shortest path from figure

in such a way that adding which - which vertices we get minimum weight as in the table.

destination vertex का कुल कुल edge जोड़ि table का minimum weight पाइ थाए path

The shortest path from a to b = {a, b} with weight 7.

too 7 forms third path

The shortest path from a to c = {a, c} with weight 9.

The shortest path from a to d = {a, c, d} with weight 20

Since 11+9=20

The shortest path from a to e = {a, c, f, e} with weight 20

The shortest path from a to f = {a, c, f} with weight 11.