Module 2

Self Balanced Tree :-

Self balancing tree is a tree that automatically keeps its height minimal after insertions or deletion. The height is typically maintained in order of $\log n$. ie $O(\log n)$

Eg :- B-Tree, Red-Black Tree, AVL Tree, Splay tree, Treap, 2-3 tree.

Height Balanced Tree

A height balanced tree is one in which the difference in the height of the two subtrees for any node is less than or equal to some specified amount.

One type of height balanced tree is the AVL tree named after its originator

- Adelson Velski and Landis

AVL tree :-

In AVL tree the height difference may be nomore than 1.

Definition :-

* An empty binary tree B is an AVL tree
* If B is the non empty binary tree with B_L and B_R are its left and right subtrees then B is an AVL tree if and only if :
  (i) B_L and B_R are AVL trees. and
  (ii) $|h_L - h_R| \leq 1$, where h_L and h_R are heights of B_L and B_R subtrees respectively

Balance Factor :-

To Implement an AVL tree each node must contain a balance factor, which indicates its states of balance relative to its subtrees. if balance is defined by

(Height of left subtree) - (Height of right subtree)

then balance factors in a balanced tree can only have values $-1, 0$ or 1.

If a node 'x' has balance factor 1 then that means left subtree is at greater height than the right subtree, then it is said to be left balanced.

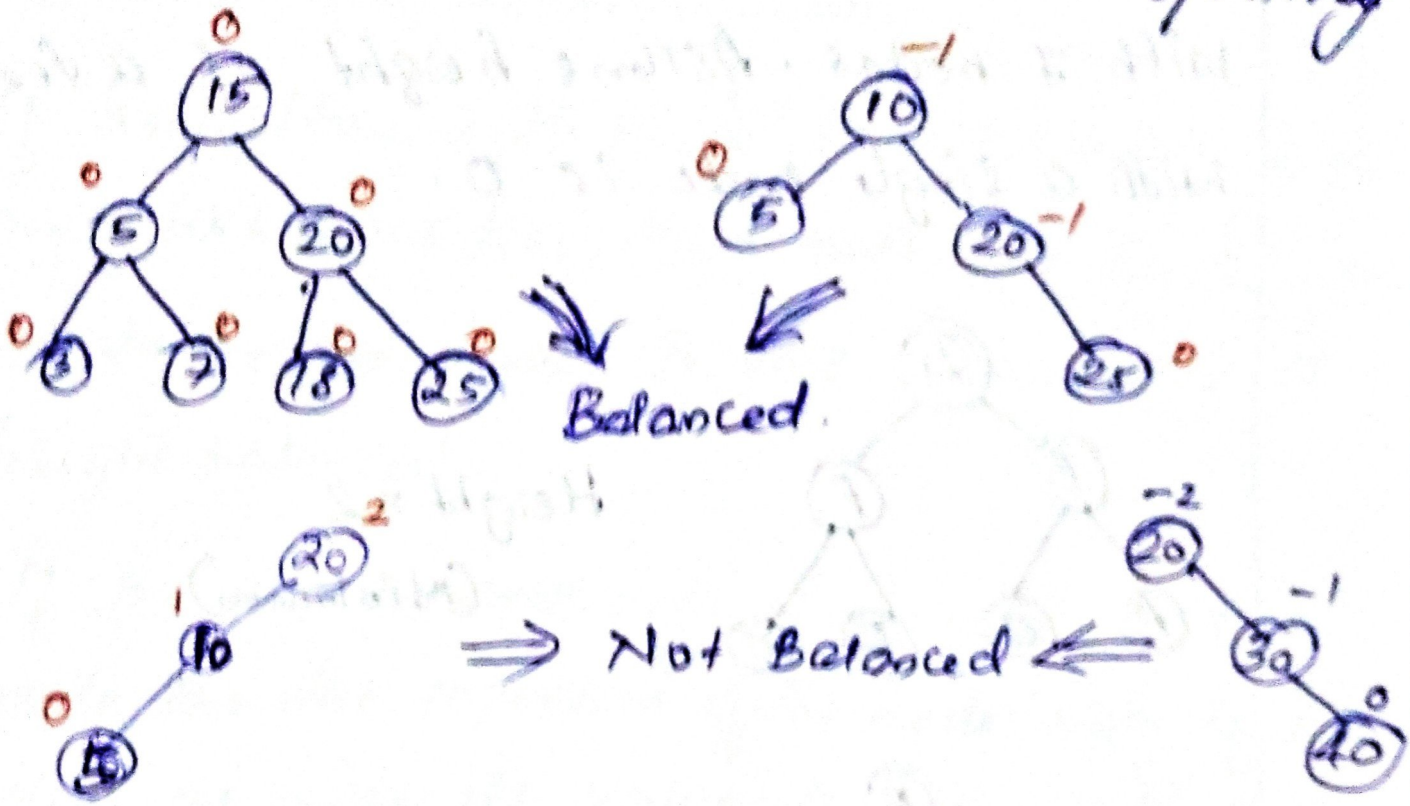
If a node 'x' has balance factor -1 then that means, right subtree is at greater height than the left subtree then it is said to be right balanced. For balance factor 0 the tree is said to be (completely) balanced.

If binary tree is binary search tree then it is AVL search tree iff:

1) Binary search tree B_L and B_R are AVL search tree where B_L and B_R are left and right subtree respectively.

2) $|h_L - h_R| \leq 1$ where h_L and h_R are

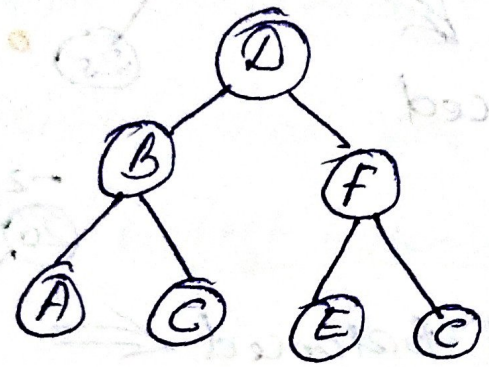
heights of left and right subtrees, respectively



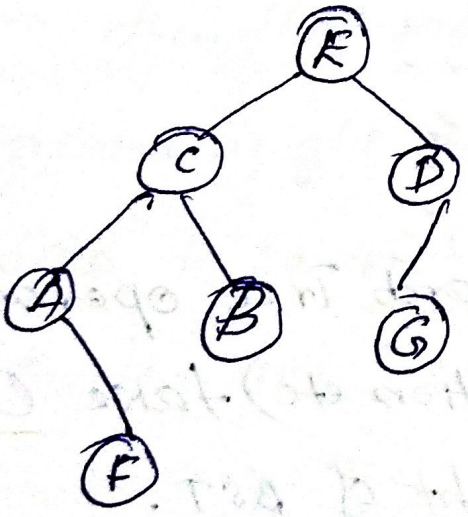
Why AVL tree?

* Most of the Binary Search Tree operations (eg:- search, insertion, deletion etc) take $O(h)$ time where 'h' is height of BST.
* Minimum height of BST is $\log n$.
* Height of AVL tree is always $O(\log n)$ where n is number of nodes in tree.
* So time complexity of all AVL tree operations are $O(\log n)$

eg) Minimum & Maximum height of an AVL tree with 7 nodes. Assume height of a tree with a single node is 0.

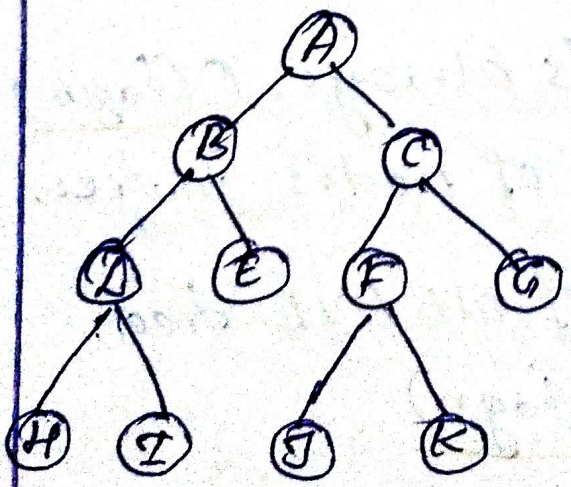


Height = 2
(Minimum)



Height = 3
(Maximum).

eg: For 11 node AVL tree

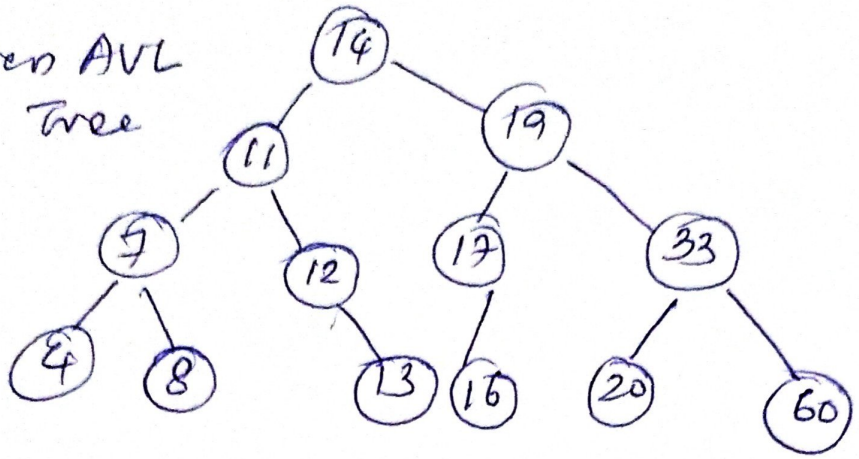


Height = 3

Minimum & maximum is same in this case.

Deletion in AVL Tree

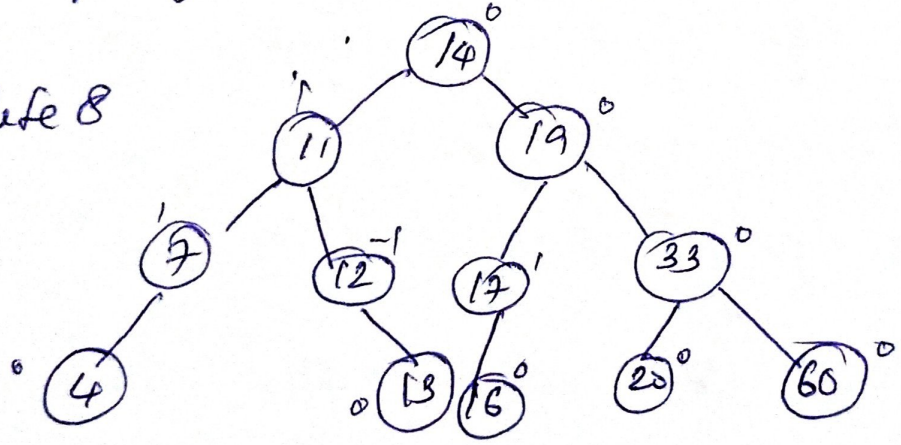
Given AVL Tree



Delete 8, 7, 11, 14, 17

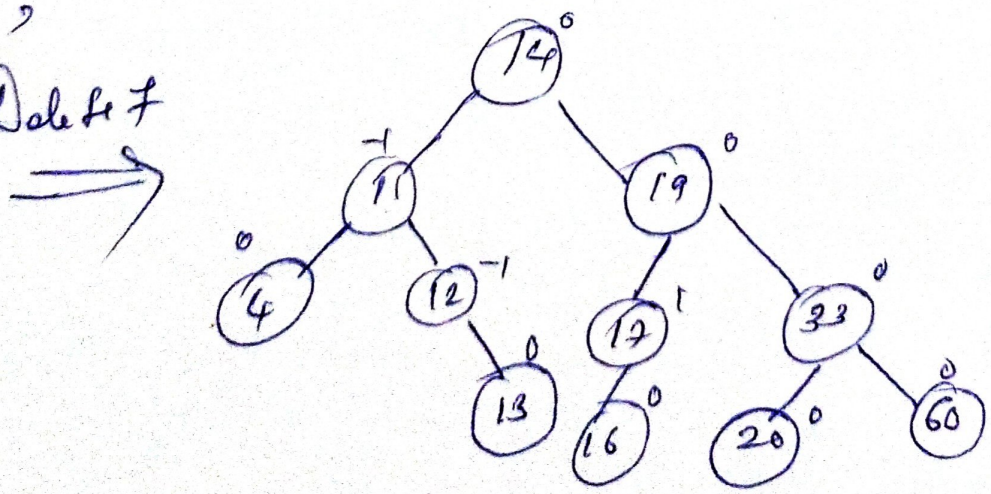
Deletion is same as that of BST. In addition we have to check balance factor of each node and perform rotation if needed.

Delete 8



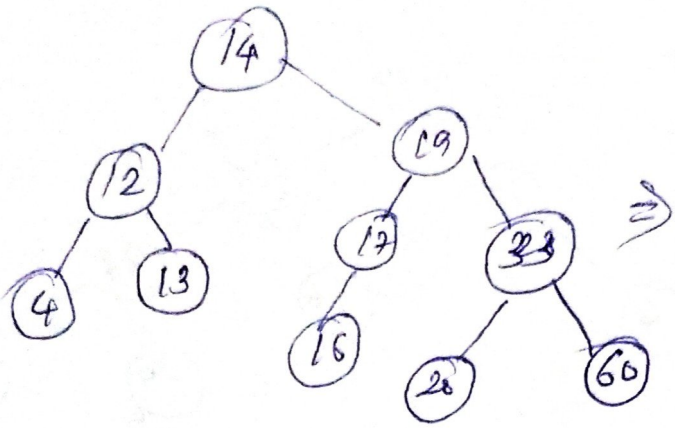
Check ~~the~~ balance factor. Here it is balanced

Delete 7



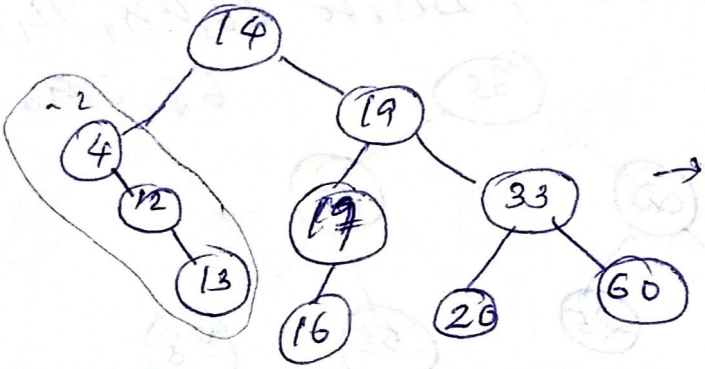
Here also balanced.

Delete 11

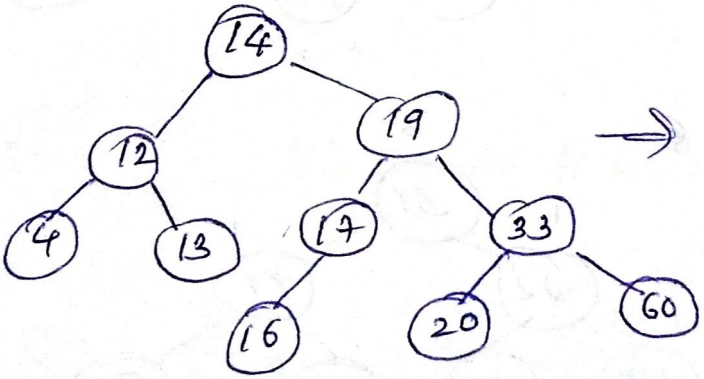


⇒ Balanced

or

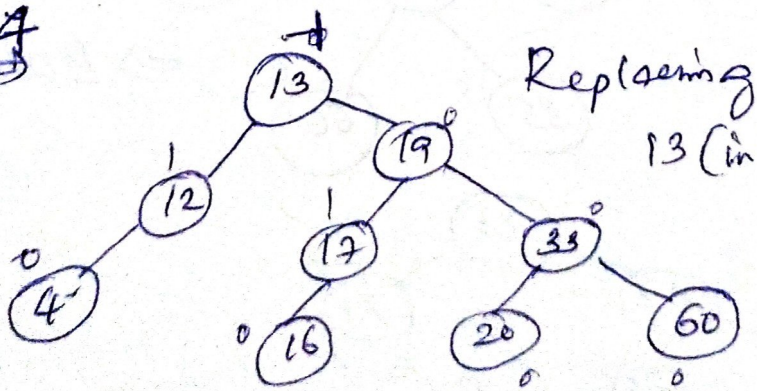


→ Not balanced



→ Balanced.

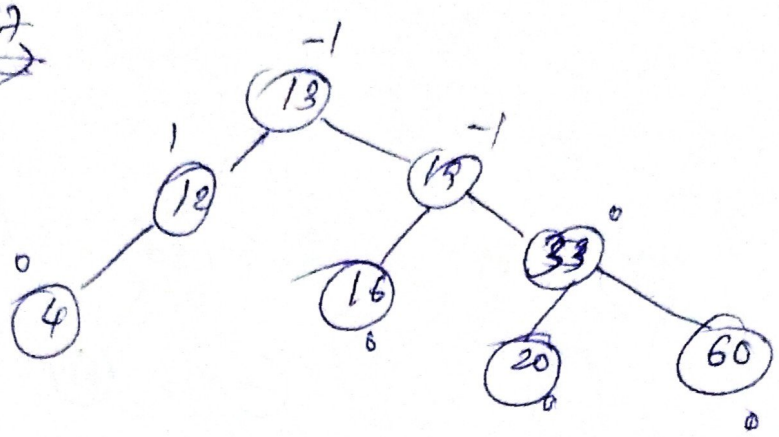
Delete 14



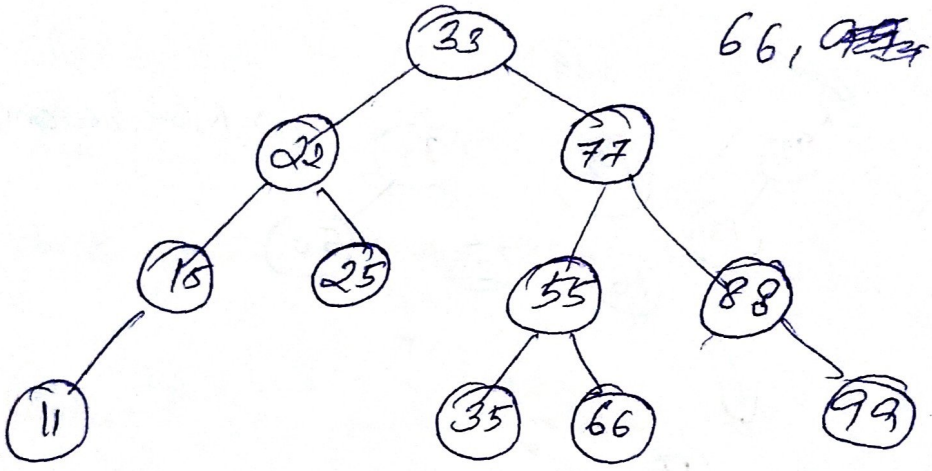
Replacing 14 with 13 (inorder predecessor)

⇒ Balanced

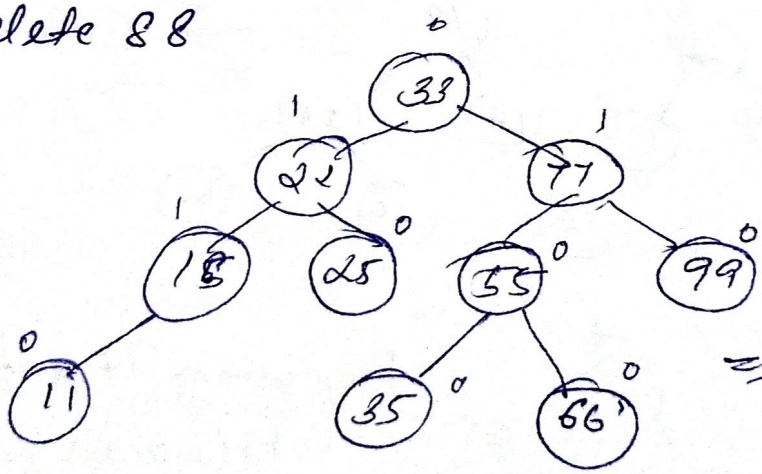
Delete 17 ⟹



Q) Given tree, Delete 88, 99, 24, 33, 11, 15, 35, 66, ~~77~~ 71, 55

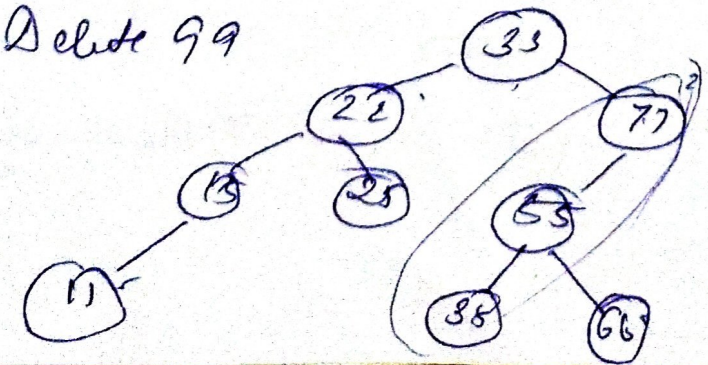


Delete 88



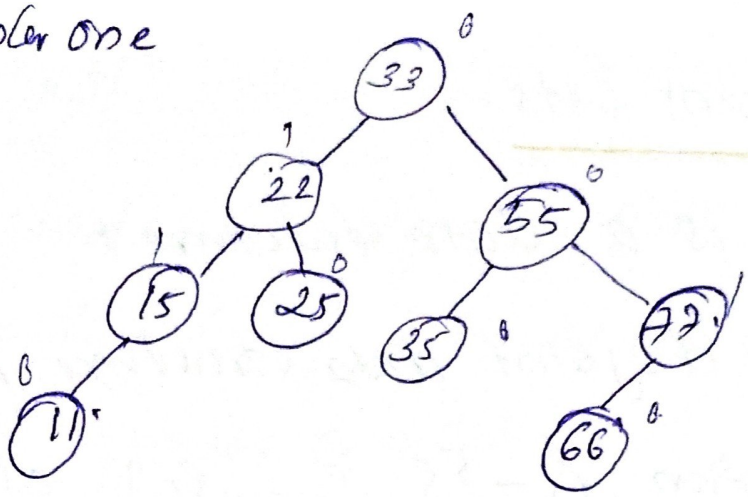
⟹ Balanced

Delete 99



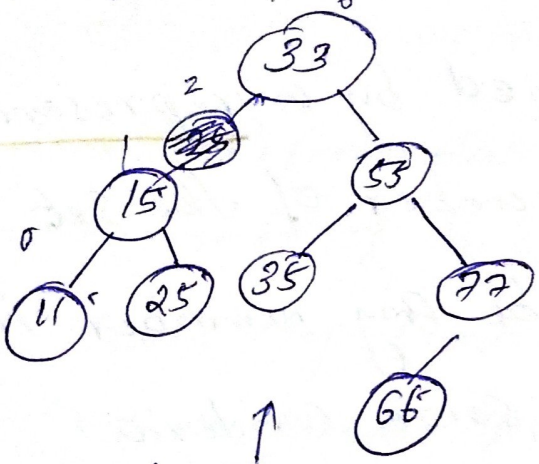
⟹ Imbalance
In this case you can do RR or LR Rotation

Taking simple one RR Rotation.



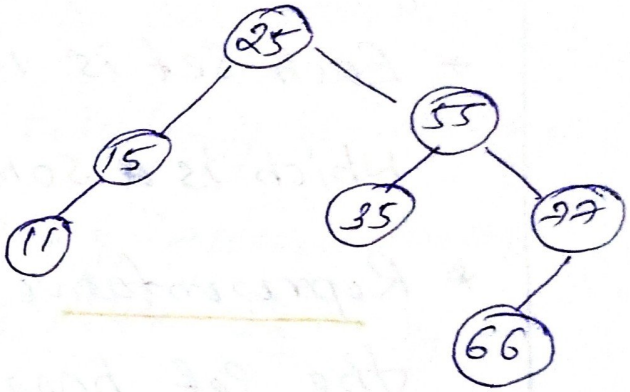
Balanced.

Delete 22.

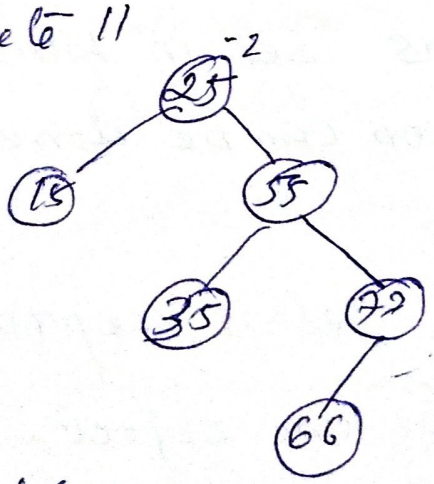


~~Not~~ balanced.

Delete 33.

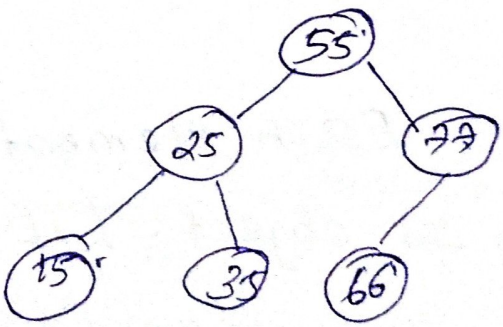


Delete 11

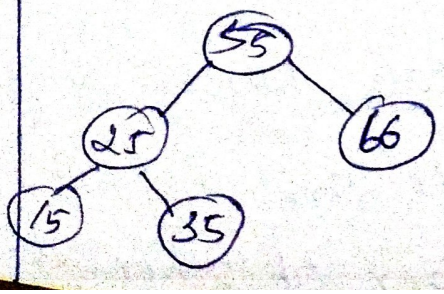


Not balanced.

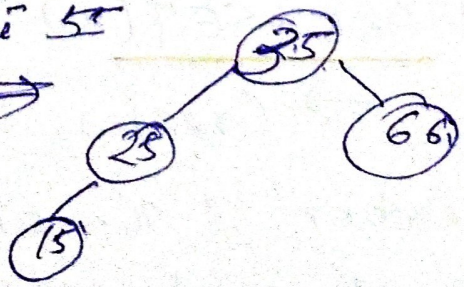
RL rot or ~~RR~~ RL rot



Delete 77



Delete 55



Disjoint Sets.

* It is a data structure.
* A disjoint data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots S_k\}$ of disjoint dynamic set.
* Each set is identified by a representative which is a some member of the set
* Representative can be any member in the set based on some criteria
  Eg: Smallest one.

[NB: Dynamic set means set in which insertion and deletion can be done]

Each element of set is represented by an object. Let x be the object. Following operations can be done on the set.

+) MAKE-SET(x)

→ Creates a new set whose only member is x. Since sets are disjoint, x should not be member of other set.

2) UNION(x,y)

↳ Unites the dynamic sets that contains x and y. Say S_x & S_y into a new set that is the union of these two sets.

⇒ The two sets are assumed to be disjoint prior ~~to the~~ to the operation.

⇒ Representative of resulting set is any member of $S_x \cup S_y$ [Many implementation chose representative of either S_x or S_y as new representative].

⇒ After union, remove S_x & S_y from the set since it is ~~disjoint~~ disjoint

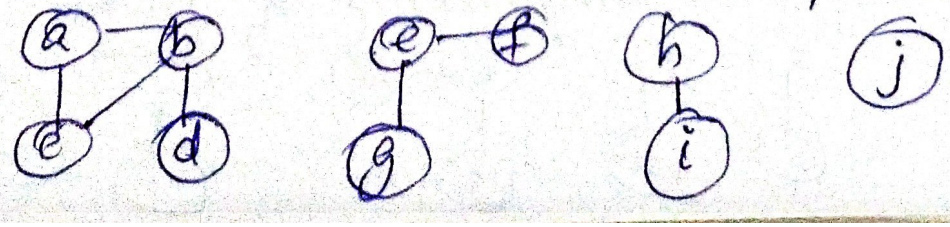
3) FIND-SET(x).

↳ Returns a pointer to the representative of the set containing x.

Application of disjoint set data structure

* Determining the connected components of an undirected graph.

eg :- Graph with 4 connected components.



Let G be the graph, set of vertices is denoted by V(G) and set of edges E(G)

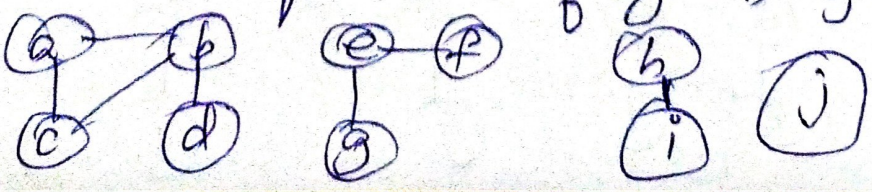
eg:- $V(G) = \{a, b, c, d, e, f, g, h, i, j\}$

$E(G) = \{(b,d), (e,g), (a,c), (h,i), (a,b), (e,f), (b,c)\}$.

Find connected components.

Edge processed	Collection of disjoint sets								
Initial sets.	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i} {j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i} {j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i} {j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i} {j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}	{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}	{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}		{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}		{h,i}		{j}

We got 4 disjoint sets. which are 4 connected components of given graph.



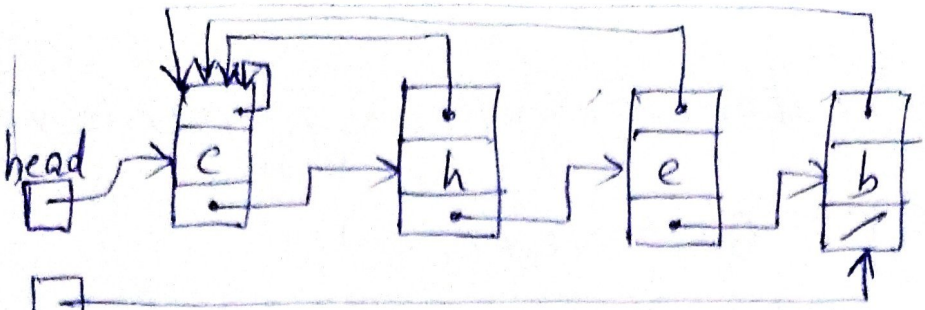
This can be done by following procedure.

```
CONNECTED-COMPONENT(G)
1   for each vertex v ∈ V(G)
2       do MAKE-SET(v)
3   for each edge (u,v) ∈ E(G)
4       do if FIND-SET(u) ≠ FINDSET(v)
5           then UNION(u,v)
```

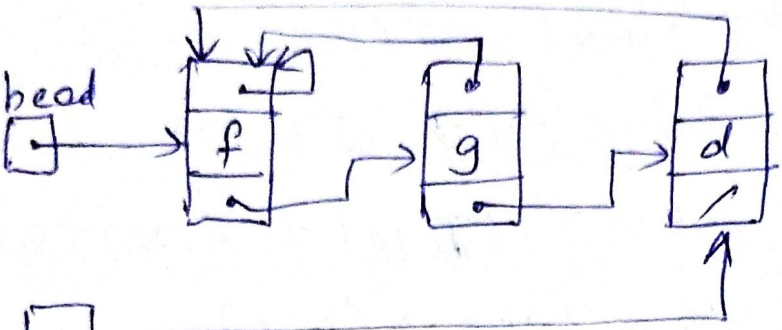
Linked list representation of disjoint set:-

A disjoint data structure can be represented using linked list.

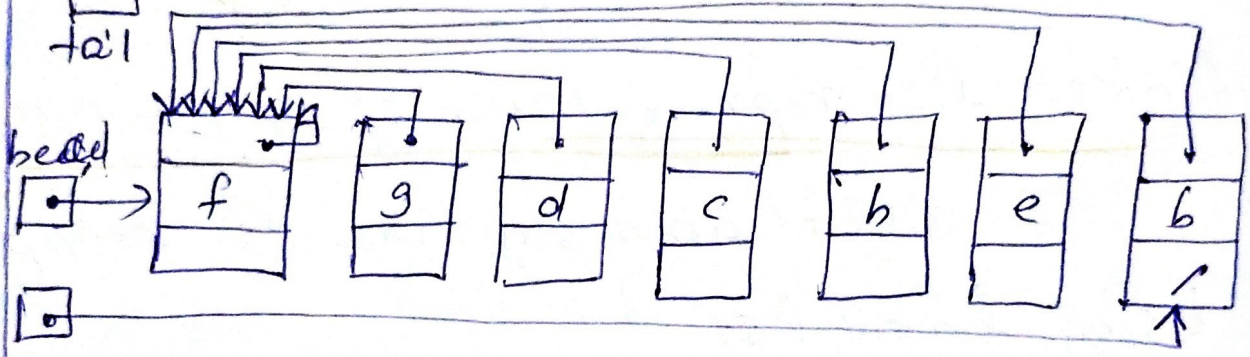
* Each set is represented as linked list
* First object is each linked list serves as its set's representatives.
* Each object in linked list contains a set member, a pointer to the object containing next set member and pointer back to the representative.
* Each list maintains pointers head, to the representative and tail to the last object in the list.
* Within each list object may appear in any order



(a)



(b)



(c).

First linked list representation represents a set with objects b, c, e, h with c as representative.

(b) => Represents another set with objects f, g, d with f as representative.

Each object on the list contains a set member, a pointer to next object on list and a pointer back to first object on the list.

Each ~~list~~ list has pointers head and tail to first and last objects.

(c) ⇒ The result of UNION(e, g). ~~The rep~~ The representative of resulting set is f.

Suppose we have n objects, say $x_1, x_2, \cdots x_n$. and we execute sequence of n MAKE-SET operations followed by n-1 UNION operation, total no. of operations will be 2n-1 and total no. of objects updated will be,

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

A weighted union heuristics/rule

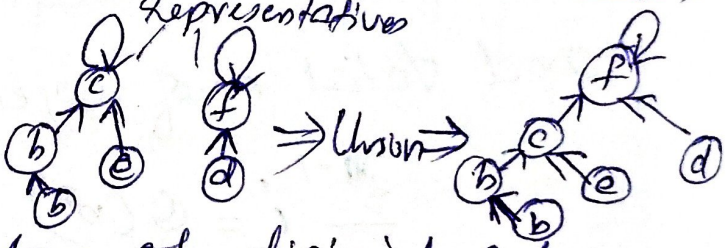
In worst case UNION will take $\Theta(n)$ time per call because we may be appending a longer list onto shorter list, we must update the pointer to the representative for each member of longer list.

Suppose each list also includes length of list,

We can easily append smaller list into longer one. With this simple ~~heuristic~~ weighted-union heuristic also, it may take $\Omega(n)$ time for UNION operation, if both sets have $\Omega(n)$ members.

However, a sequence of m MAKESET, UNION and FIND-SET operations, n of which are MAKESET operation, it takes $O(m+n\lg n)$ time.

Disjoint set forests (eg)



* Another implementation of disjoint sets.

In disjoint forest, each member points only to its parent. The root of each tree contains representative and is its own parent.

* This is the fastest implementation when compared to linked list

Two heuristics to improve running time are (a) Union by Rank (b) Path compression.

(a) Union by Rank

* Here root of tree with fewer nodes point to the root of tree with more nodes. We maintain a rank which is upperbound of height.

In union by rank, the root with smaller rank is made point to root with larger rank during union operation.

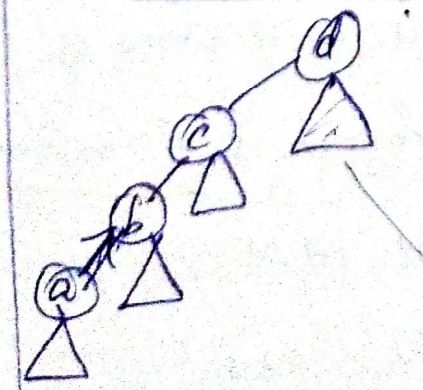
(b) Path Compression:

* Simple and more effective.

* During FINDSET operation, make each node x points directly to root node.

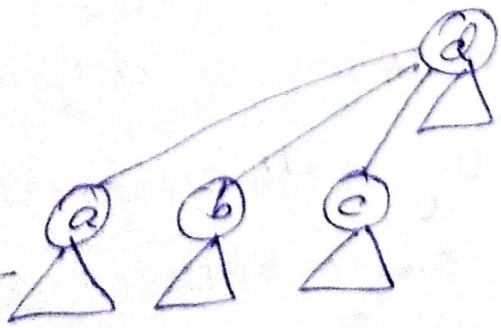
Path compression during FINDSET operation is shown below.

Before



Subtrees.

After.



Pseudocode for disjoint forest

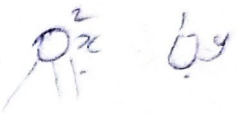
$p[x] \Rightarrow$ parent of x

```
MAKESET(x)
1   p[x] ← x
2   rank[x] ← 0

UNION(x,y)
1   LINK(FIND-SET(x), FIND-SET(y))

LINK(x,y)
1   if rank[x] > rank[y]
2       then p[y] ← x
3       else p[x] ← y
4            if rank[x] = rank[y]
5            then rank[y] ← rank[y] + 1

FINDSET(x)
1   if x ≠ p[x]
2       then p[x] ← FINDSET(p[x])
3   return p[x]
```



Running time

By union by rank, the running time of operations, of disjoint ~~for~~set forest is $O(m \log n)$

By path compression, for n MAKESET, and f ~~FINDSE~~ FINDSET operations, it takes

$O(n + f \cdot (1 + \log_{2+f/n} n))$

When using both it takes $O(m\alpha(n))$ where $\alpha(n)$ is a very low growing function. and normally $\alpha(n) \leq 4$.

Depth First Search :-

* Graph traversal method.
* Algorithm starts from an arbitrary node and explore as far as possible in the graph before backtracking.
* After backtracking, it repeats the same process for all remaining vertices which have not been visited till now.
* In DFS, the predecessor subgraph produced may composed of several trees, because search may be repeated from multiple sources.
* The predecessor subgraph of DFS forms a depth first forest composed of several depth first trees. The edges are called as tree edges.

Besides creating depth first forest, the DFS also timestamps each vertex.

* Each vertex will have 2 timestamps.
  1. d[v] → When v is first discovered
  2. f[v] → When search finishes examining v's adjacency list.
* These timestamps are integers between 1 & 2|V|
* For each vertex d[u] < f[u]
* Vertex v is WHITE before time d[u], GRAY between time d[u] and f[u] and black thereafter.

Pseudocode :

```
DFS(G)
1   for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4   time ← 0
5   for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7          then DFS-VISIT(u)
```

DFS-VISIT(u)

1 Color[u] ← GRAY ⇒ White vertex u just been discovered

2 time ← time+1

3 d[u] ← time

4 for each v ∈ Adj[u] ⇒ Explore edge (u,v)

5 do if Color[v] = WHITE

6 then π[v] ← u

7 DFS-VISIT[v]

8 Color[u] ← BLACK ⇒ Blacken u; it is finished

9 f(u) ← time ← time+1

Running time :-

Lines
Loops 1-3 & 5-7 takes $\Theta(V)$ time.

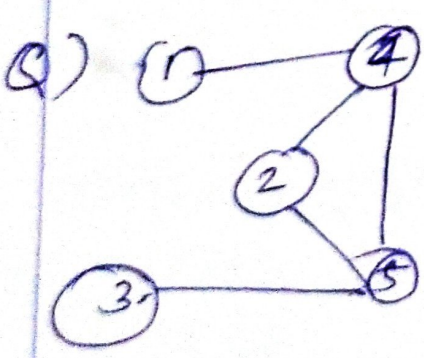
During visit of v, ie, execution of DFS visit(), the loop on lines 4-7 is executed $|Adj[v]|$ times.

ie Total cost $\sum_{v \in V} |Adj[v]| = \Theta(E)$

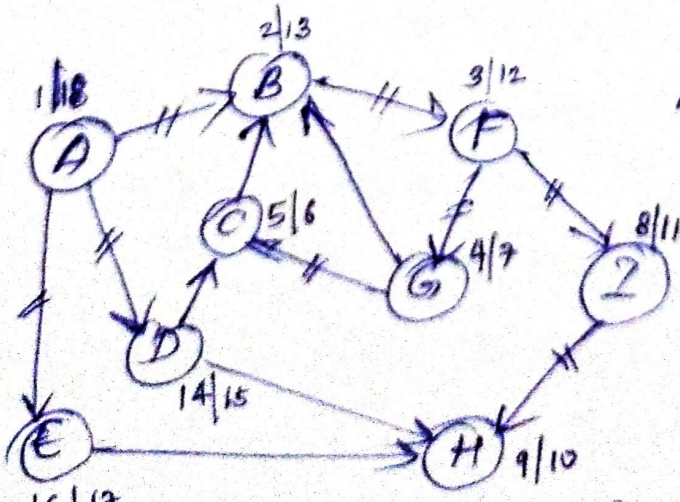
$\therefore$ Running time of DFS is $\Theta(V+E)$

~~Breadth first Search~~

~~* Graph traversal method~~

Q) 

$\Rightarrow$ 1 4, 2, 5, 3



A - B F G C I H D E

Tree edges :-
AB, BF, FG, GC, FI, IH, AD, AE

Back edges
CB, [redacted], GB, [redacted]

Fwd
EA, DH, No fwd edges

Cross
DC, DH, EH

Classification of Edges

The edges in a graph can be classified by using DFS. The four type of edges are :-

① **Tree edges**

- Edges in depth first forest G_π. Edge (u,v) is a tree edge if v was discovered by exploring edge (u,v).

② **Back edges**:-

→ Edges (u,v), which are connecting a vertex u to an ancestor of v in a depth first tree. Self loops which may appear in directed graphs are considered to be back edges [if v appear before u & there is path from v to u]

③ **Forward edges**:-

→ Are those nontree edges (u,v) connecting a vertex u to descendant v in a depth first tree. [if v appears after u & there is path from u to v]

④ **Cross edges**:-

→ All other edges are cross edges. They can go between vertices in same depth first tree.

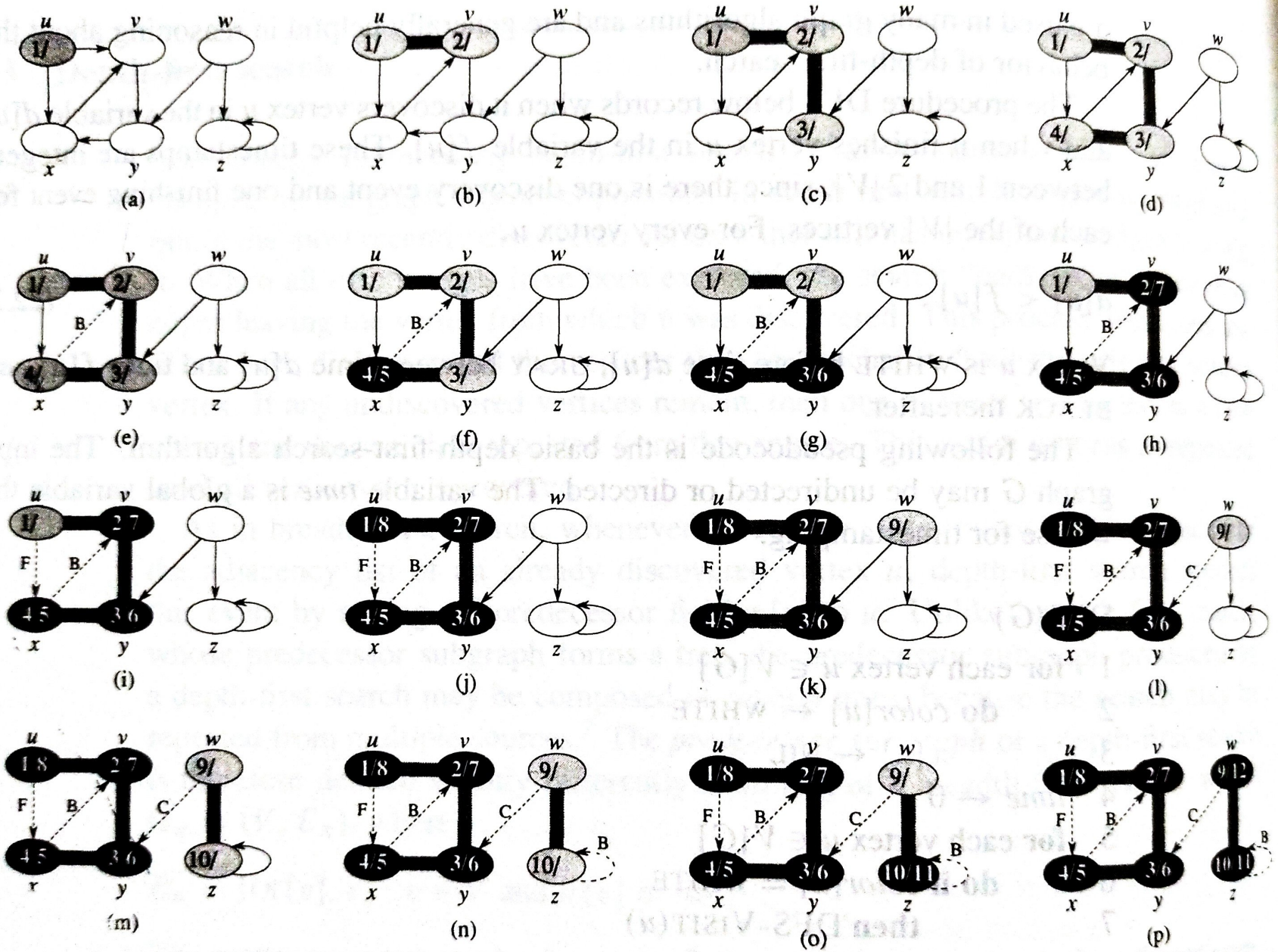


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

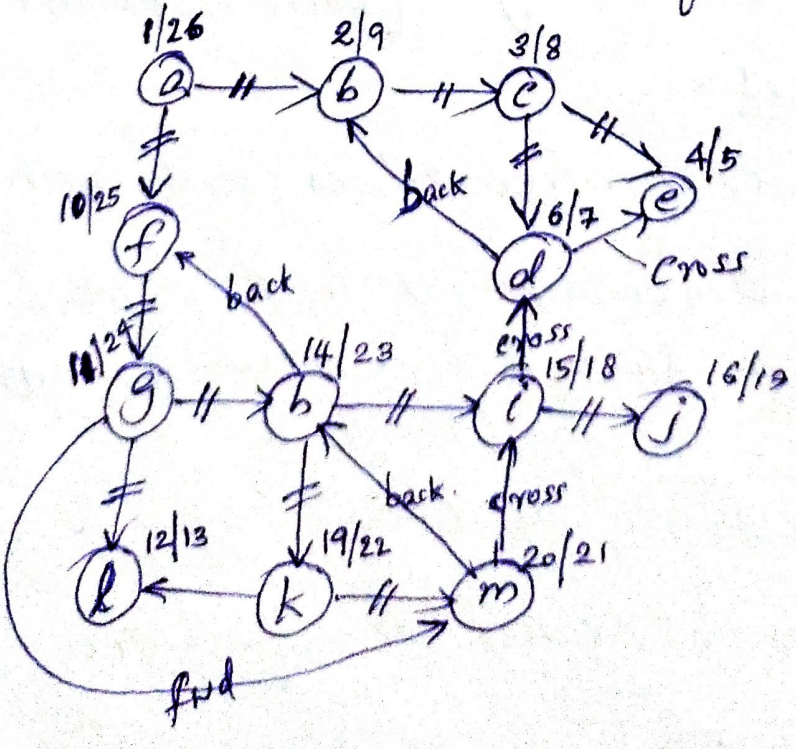
No.6 In previous figure of DFS traversal, it shows various types of edges.

If the edges are shaded it is tree edges. and dashed otherwise. Nontree edges are labelled B, C, or F according to whether they are back, cross or forward edges.

The key idea is that each edge (u,v) can be classified by color of vertex v that is reached when the edge is first explored.

1) WHITE indicates a tree edge
2) GRAY indicates a back edge
3) BLACK indicates a forward or cross edge.

eg:



Marked edges are tree edges

Tree edges :- (a,b) (b,c) (c,e) (c,d) (a,f) (f,g) (g,h) (h,i) (i,j) (h,k) (k,m) (g,l)

Back edges :- (d,b) (h,f) (m,h)

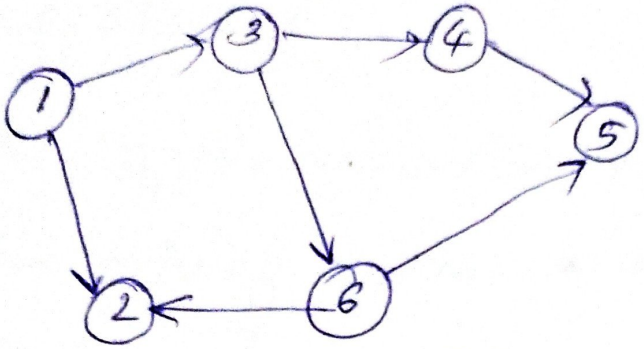
Fwd edges :- (g,m)

Cross edges :- (d,e) (m,i) (i,d) (k,l)

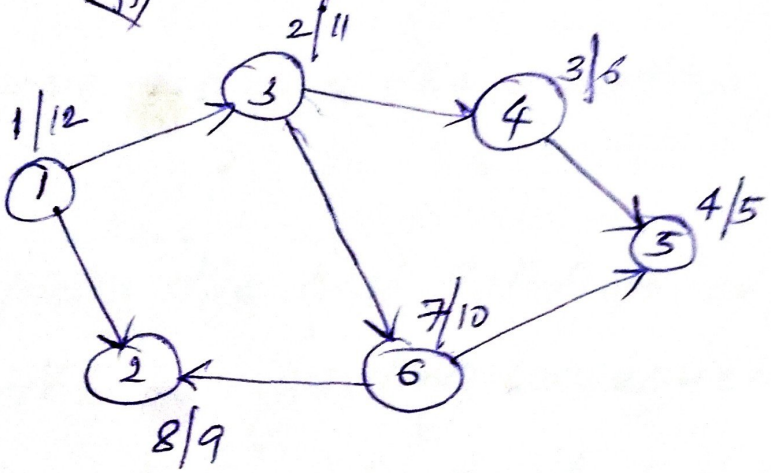
Topological Sorting Problem

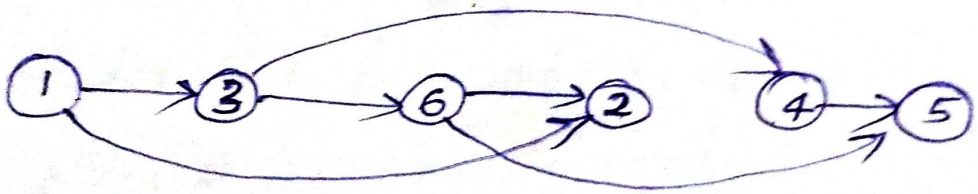
[For Algm & for ppt shared]

(1)



↓ DFS.





[By arranging nodes in decreasing order of finish time]

NB A graph can have more than one topological sorting order.