Unit - III

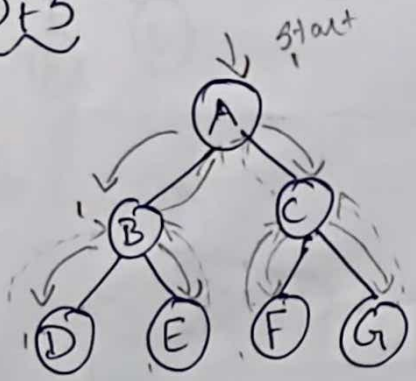
① Graph Traversal:

- ⇒ Traversal → process of visiting all nodes in a graph.
- ⇒ Doesnot have any root & assume any node as starting node & visit all the nodes any repetition.
- ⇒ Nodes in graph is visited only once

② Types:

- ⇒ Depth First Search (DFS) → Based on depth
- Breath First Search (BFS) ⇒ Based on level.
- ⇒ DFS - Stack
- BFS - queue.

DFS



⇒ A B D E C F G
⇒ A C G F B D E

Step 1: A — DFS

Step 2: C, B — A

Step 3: G, F, B — A C

Step 4: D, B — D C G F

Step 5: E, D — A C G F B

Step 6: — A C G F B E D

DFS Algorithm

⇒ Each vertex of graph
- Visited
- Not visited

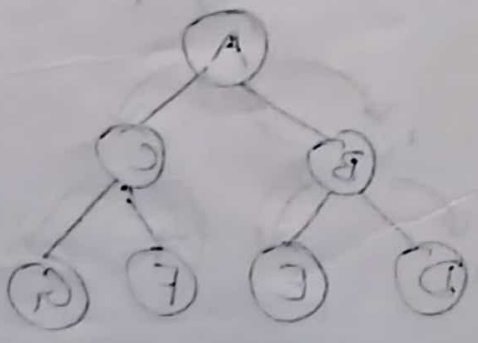
⇒ The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

⇒ ① Start by putting any one of the graph's vertices on top of a stack

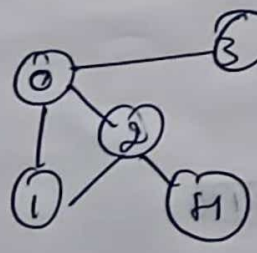
⇒ ② Take the top item of the stack & add it to the visited list.

③ Create a list of that vertex adjacent nodes. Add the ones which aren't in the visited list to the top of the stack

④ Keep repeating steps 2 & 3 until the stack is empty



Step : 1



Visited :

adjacent vertex Stack :

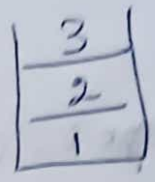
Start from vertex 0.

Step 2: visited

0				

stack

1	2	3		



top of stack. visited adjacent nodes.

Step 3: visited

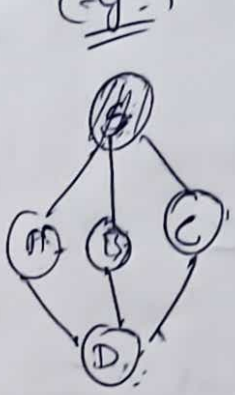
0	1			

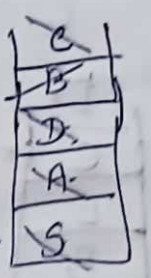
stack

2	3			

DFS

Eg: 1



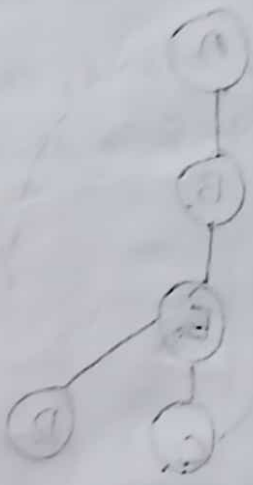


top

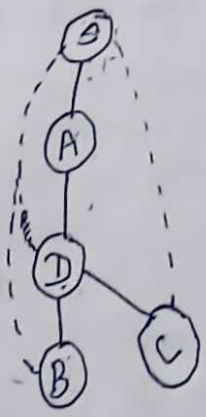
indicates the stack

stack.

B C S



2)



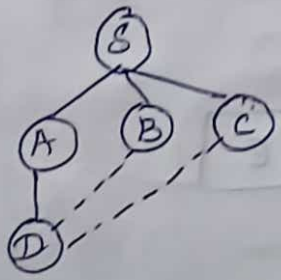
—— tree edge
---- Back edge

→ SADBC

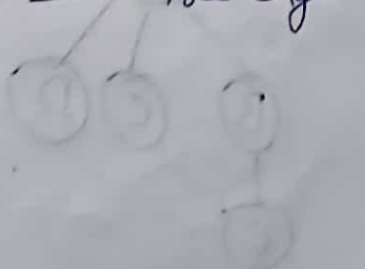
BFS

S	A	B	C	D	

2)

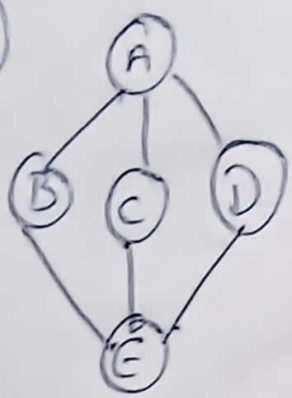


--- Cross edge
—— tree edge

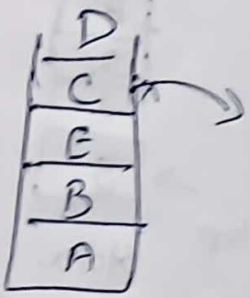


Eg: 2

(6)

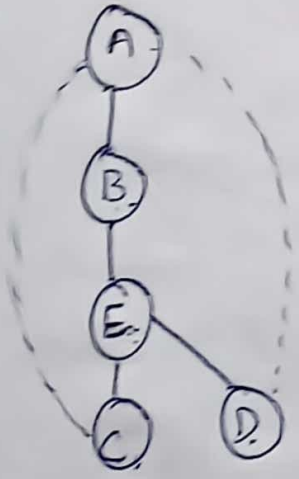


DFS Stack.



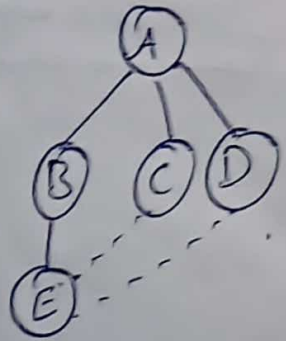
Pushing order

⇒ ABECD



BFS Queue.

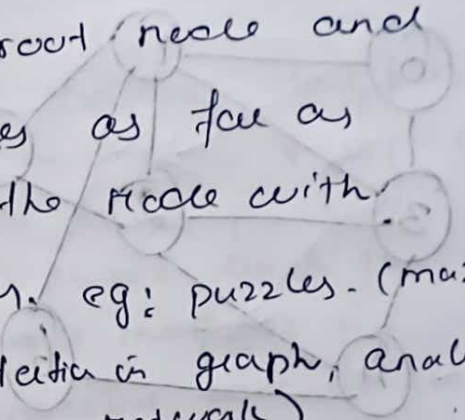
Soln



A	B	C	D	E

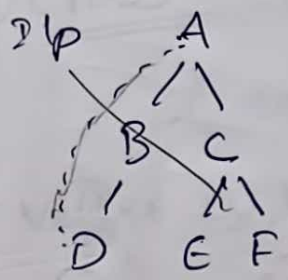
(3) DFS:

DFS is a traversal approach in which traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes. eg: puzzles. (mazes, sudoku, cycle detection in graph, analyzing network)

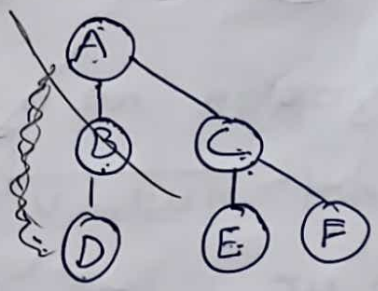


(4) BFS: (level order traversal)

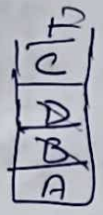
BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level. eg: finding shortest path in graph. Gps navigation systems.



DFS:

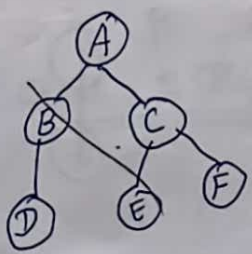


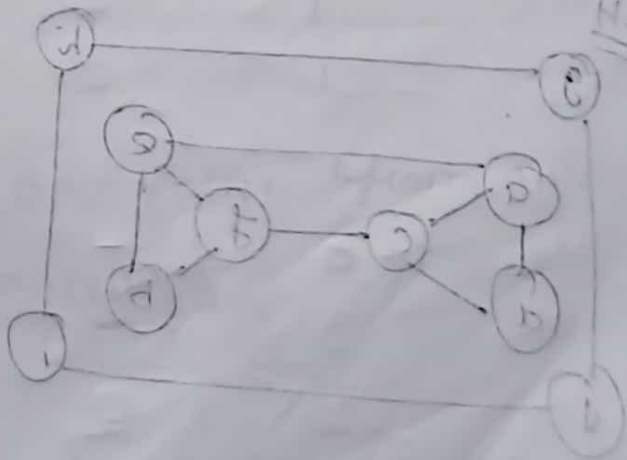
ABCDEF



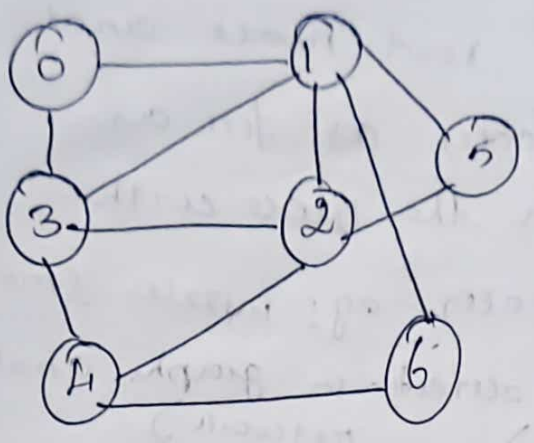
ABCDEF

DB

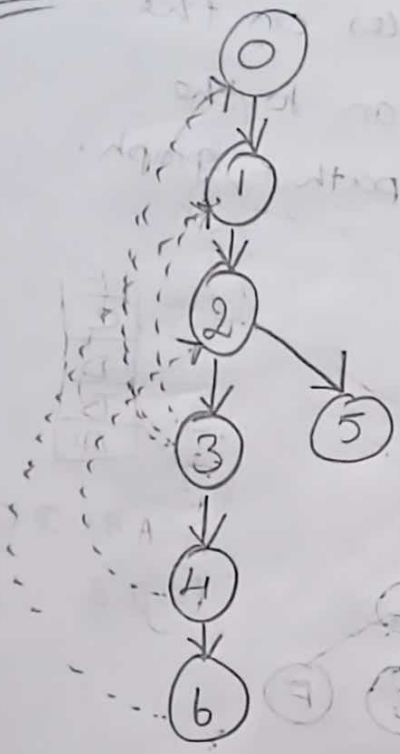




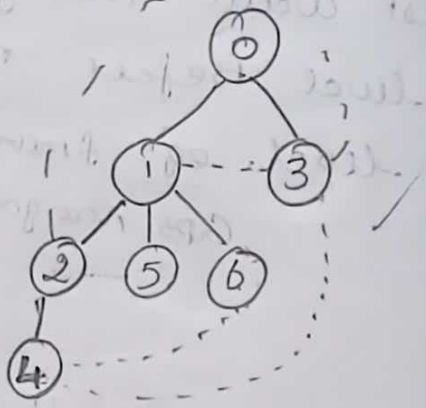
Eg: 3

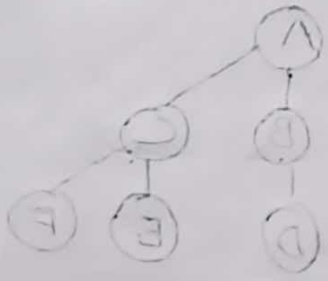


DFS

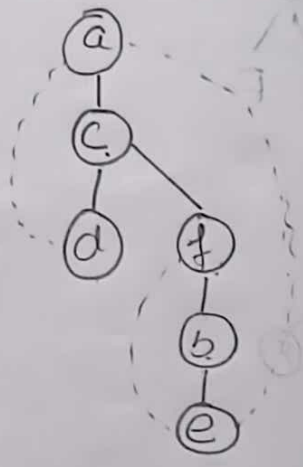


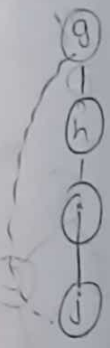
BFS



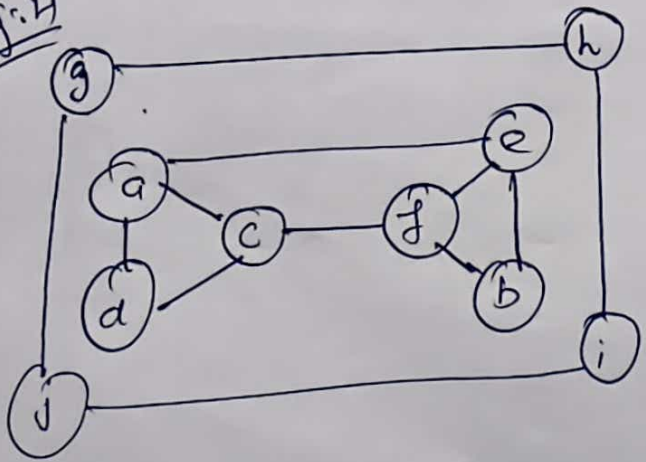


DFS

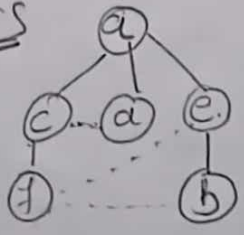


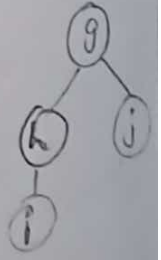


Eg: 4



BFS





(1) BFS

1) Vertex based technique

2) Visit all the nodes of the same level first & then move to next level

3) Queue data structure is used

4) Recursion or backtracking concept

5) BFS is slower

DFS

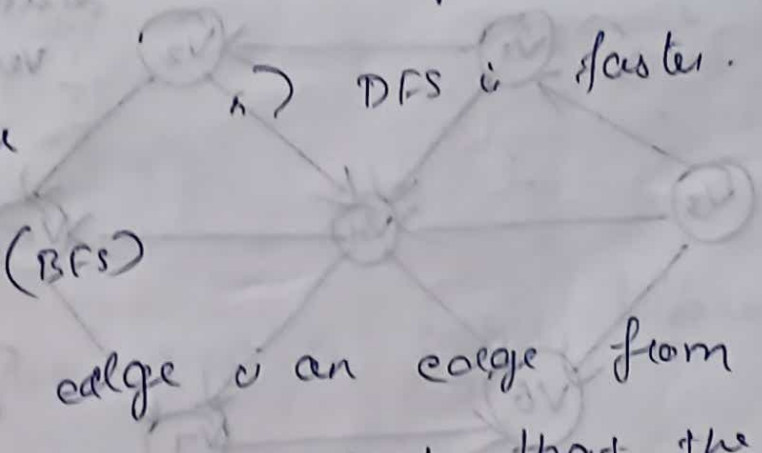
1) Edge based technique

2) Traverse start from root node & explore as far as possible until we reach the node that has no unvisited adjacent nodes

3) Stack data structure is used

4) Use backtracking concept

5) DFS is faster.



(2) Cross edge: (BFS)

A cross edge is an edge from vertex u to a vertex v such that the subtrees rooted at u and v are disjoint.

Back edge: (DFS)

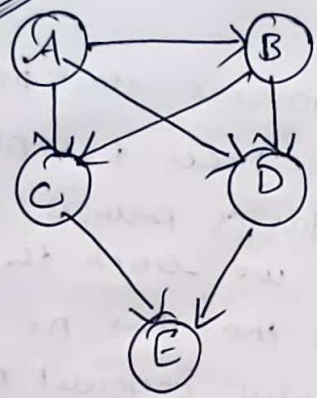
A back edge is an edge from a vertex u to one of its ancestors.

Tree edge:

Edge present in tree obtained after performing DFS & BFS

Directed Acyclic graph.

Topological Sort

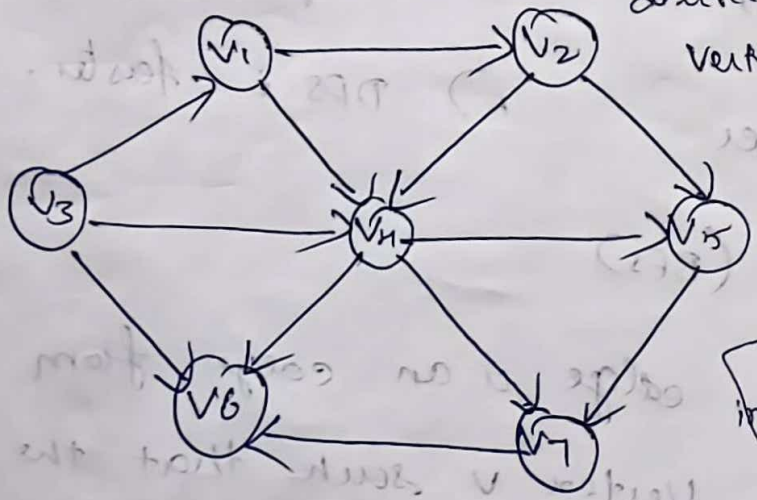


→ Directed Graph
→ Undirected Graph

DAG: Directed acyclic graph (DAG) is a digraph that has no cycles.

↳ set of vertices, when all edges are directed from one vertex towards another.

⇒



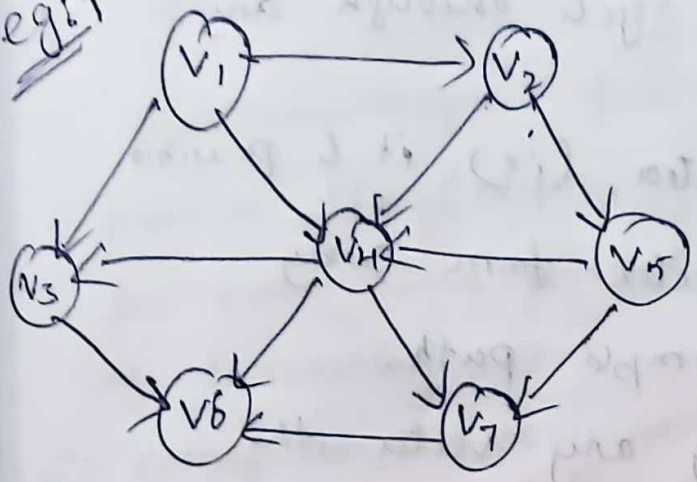
indegree changes.

indegree = 0

$V_3 \rightarrow V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7$

Vertex	1	2	3	4	5	6	7
V_1	1	0	–	–	–	–	–
V_2	1	1	0	–	–	–	–
V_3	0	–	–	–	–	–	–
V_4	3	2	1	0	–	–	–
V_5	2	2	2	1	0	–	–
V_6	3	2	2	1	0	1	0
V_7	2	2	2	1	1	0	–

eg:1



1) Scheduling the tasks
2) ordering matters

Applications of DFS:

1) undirected Graph
2) Biconnectivity
3) Euler circuits
4) Finding Strongly connected components for directed graph.

1) Biconnected Graph:

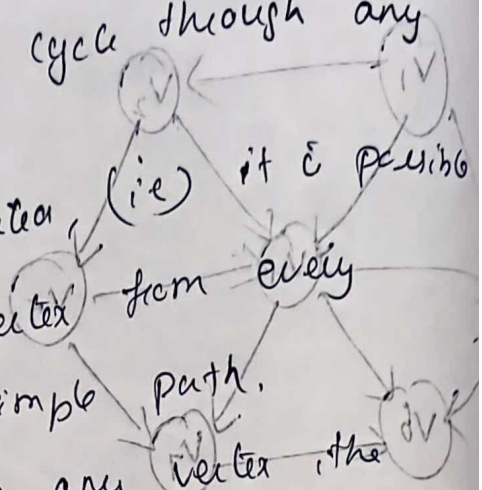
An undirected graph is called biconnected if

(i) It is connected (i.e there is a path from each vertex to every other vertex)

(ii) No articulation points. (Articulation points state if any vertex is removed, the graph becomes disconnected).

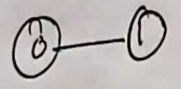
⇒ there is a simple cycle through any two vertices.

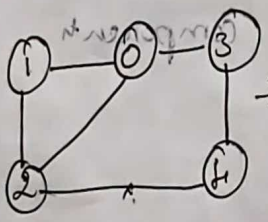
⇒ the graph is connected, (i.e) it is possible to reach every vertex from every other vertex by a simple path.

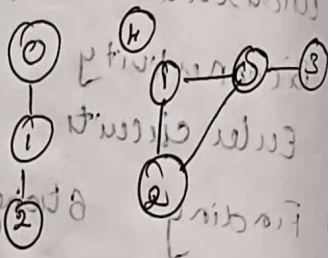


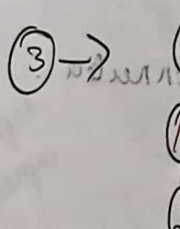
⇒ even after removing any vertex, the graph remains connected.

Examples:

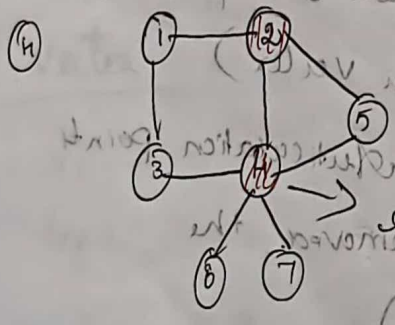
①  → Biconnected

②  → Biconnected

③  Not a biconnected.



→ Articulation point

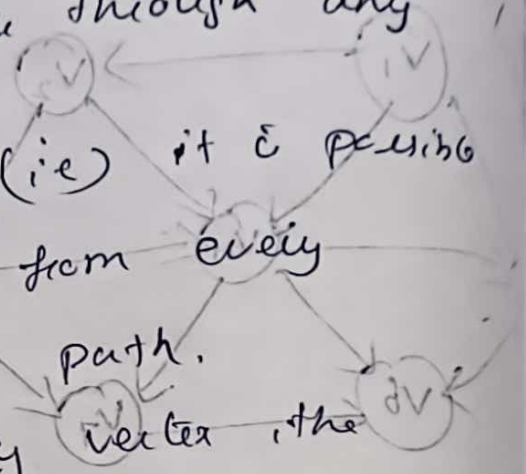
④ 

→ Articulation point

⇒ there is a simple cycle through any two vertices.

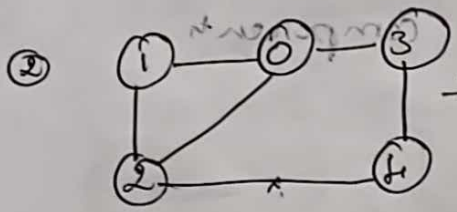
⇒ the graph is connected, (i.e) it is possible to reach every vertex from every other vertex by a simple path.

⇒ even after removing any vertex, the graph remains connected.



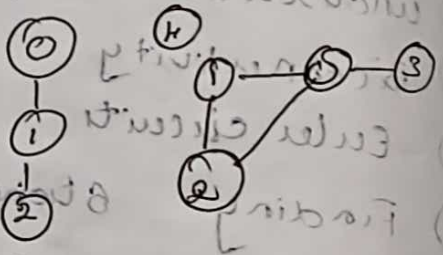
Example:

① → Biconnected

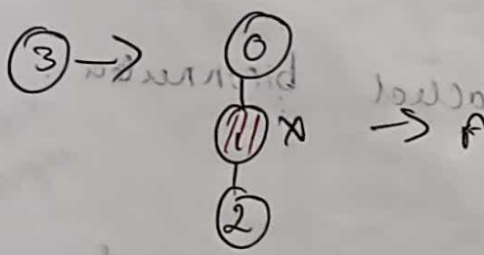


② → Biconnected

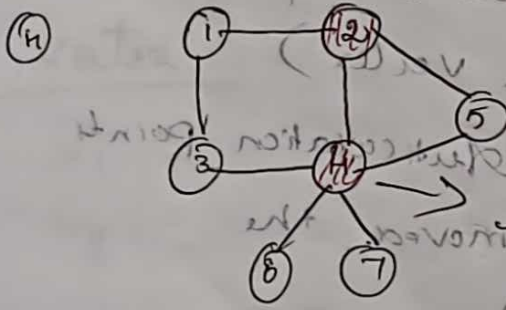
③



Not biconnected.



→ Articulation point



Articulation point

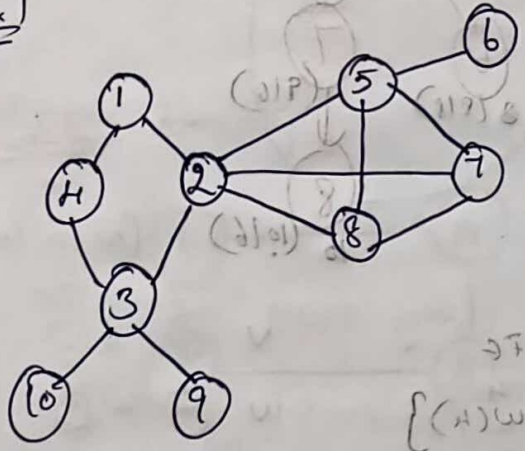
Finding an articulation point:

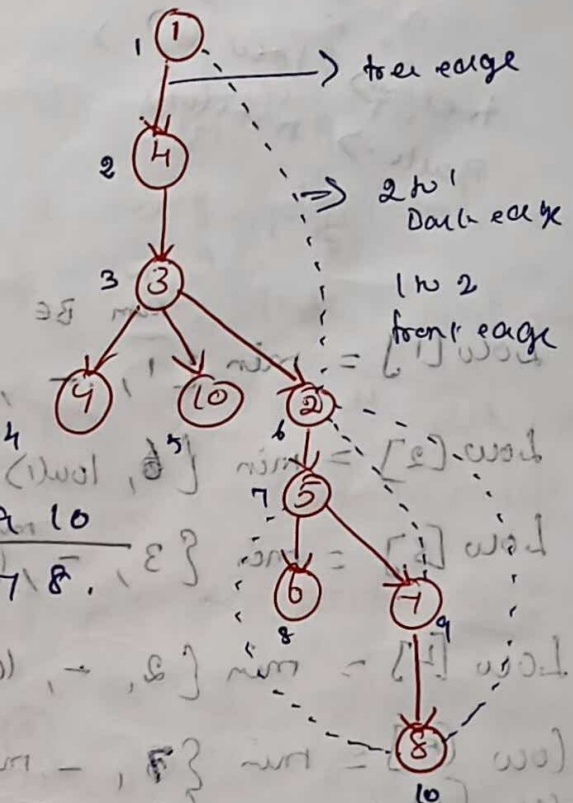
=> Construct Depth first search tree. and provide d/n (depth first number) for each node according to the order of traversing.

=> Edges can be divided as,

a) Tree edges
b) Back edges
c) Front edges

eg:-





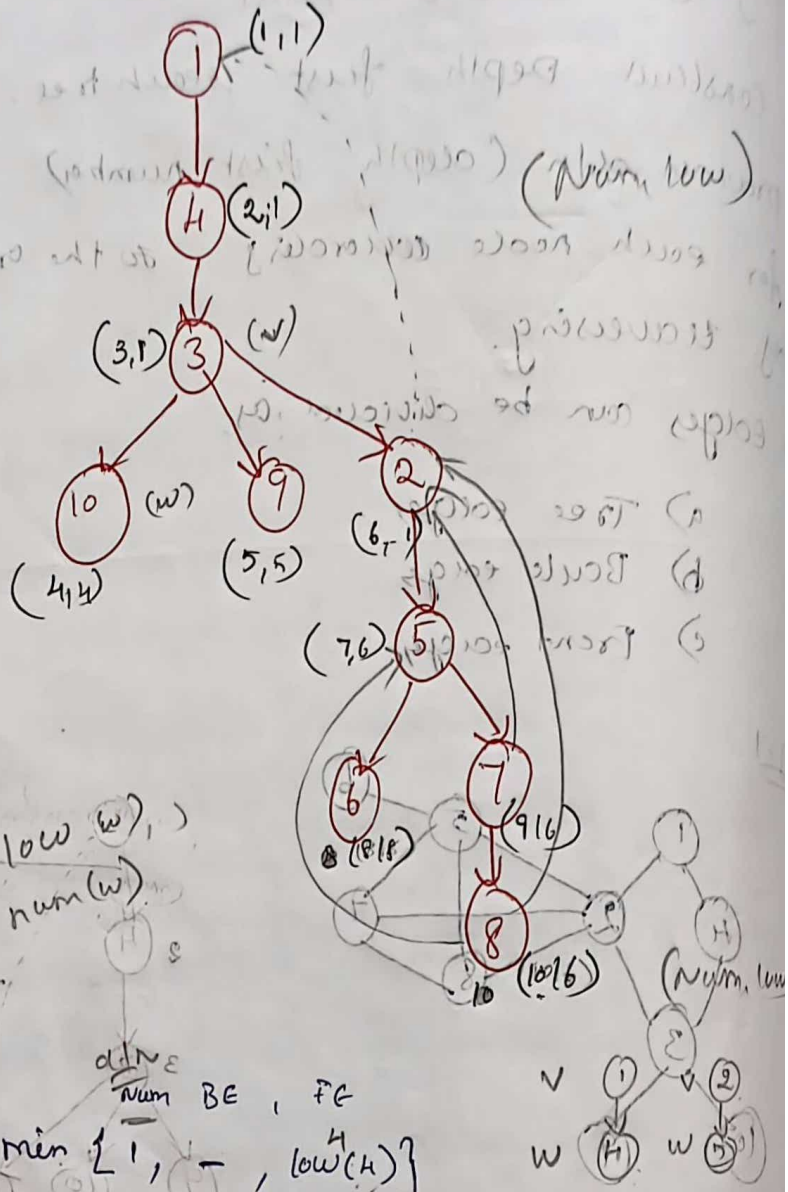
1	2	3	4	5	6	7	8	9	10
1	4	3	9	10	2	5	6	7	8.

DBS

Low [v] is the minimum of.

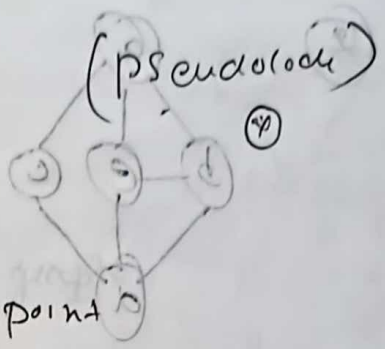
1. num(v)
2. the lowest num(w) among all back edges (v, w)
3. the lowest low(w) among all tree edges (v, w)

(Num, low)



1. Low[1] = min{1, -, low(4)}
1. Low[2] = min{2, low(1), low(5)}
1. Low[3] = min{3, -, (low(10), low(9), low(2)}
1. Low[4] = min{2, -, low(3)}
6. Low[5] = min{7, -, min(low(6), low(7))}
8. Low[6] = min{8, -, -} = 8
6. Low[7] = min{9, num(2), low(8)}
6. Low[8] = min{10, min(num(2), num(5))}, -}
⇒ min(10, 6) ⇒ 6
5. Low[9] = min{5, -, -} ⇒ 5
4. Low[10] = min{4, -, -} ⇒ 4

To find articulation point



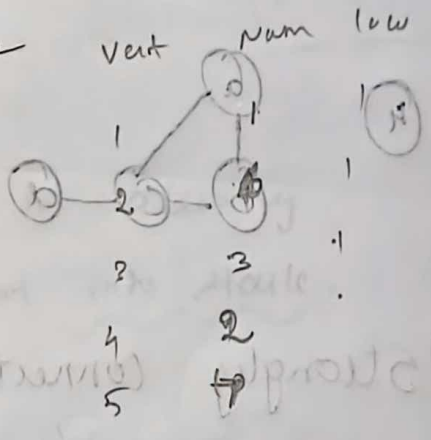
$low(w) \geq num(v)$

eg: 3, 2, 5 are articulation point

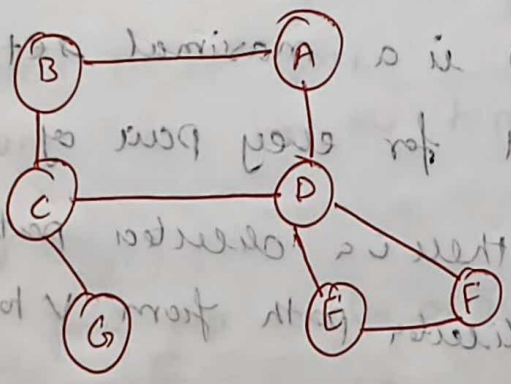
3 => $4 \geq 3$ $5 \geq 3$ ✓

2 => $6 \geq 6$ ✓

5 => $8 \geq 7$ ✓
$6 \geq 7$ ×

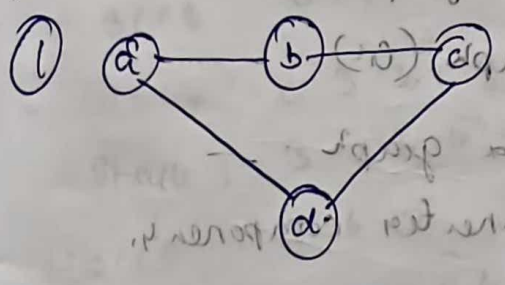


Example:

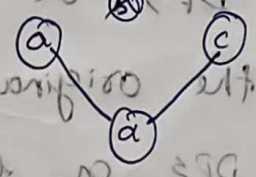


=> Check the following graphs are biconnected

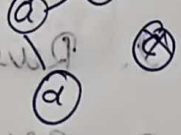
(i)



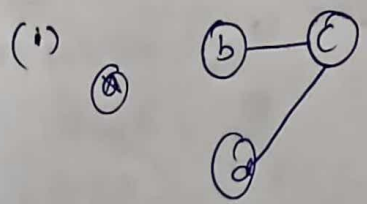
(ii)

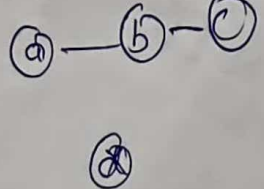


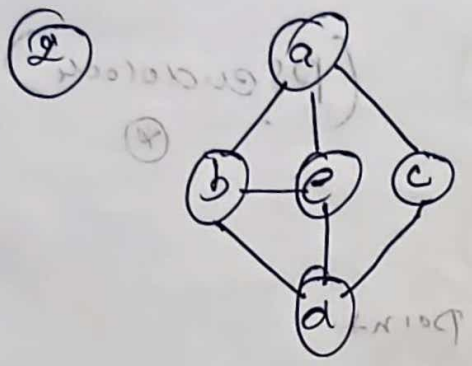
(iii)

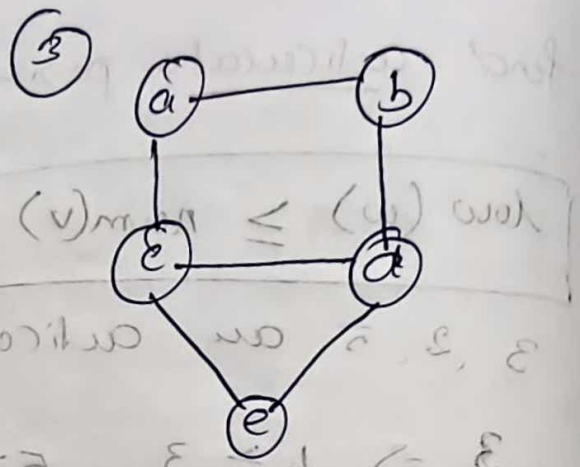


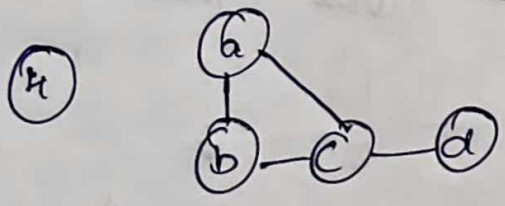
(iv)









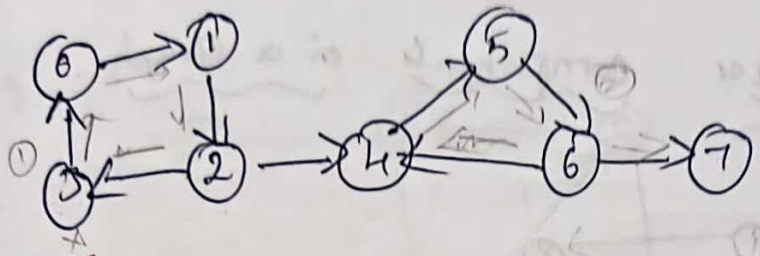


7) Strongly connected components

A strongly connected component of a directed graph (G) is a maximal set of vertices such that for every pair of vertices u and v, there is a directed path from u to v & directed path from v to u.

Steps: [Kosaraju's Algorithm] based on a DFS implementation twice.

1. Perform DFS on given graph (G)
2. Reverse the original graph (G')
3. Perform DFS on reversed graph
4. print the strongly connected components.



Step 1: Perform DFS on given graph

Stack :

Visited : 0 1 2 3

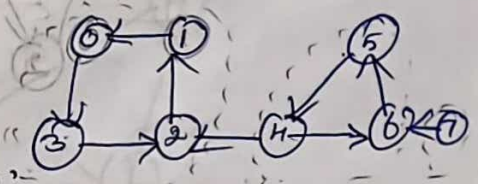
when a vertices leads to already visited vertex then push that into stack.

① Stack : 3 7 6 5 4 2 1 0

Visited : 0 1 2 3 4 5 6 7

Step 2: Reverse the graph.

⇒ Perform DFS from the top of the stack.



Step 3:

⇒ Start from top vertex of stack.

① Visited 0 3 2 1

Stack : 3 7 6 5 4 2 1 → go to top vertex in stack, if it is already visited pop.

① SCCs : 0 1 2 3

② Visited : 0 3 2 1 4 6 5

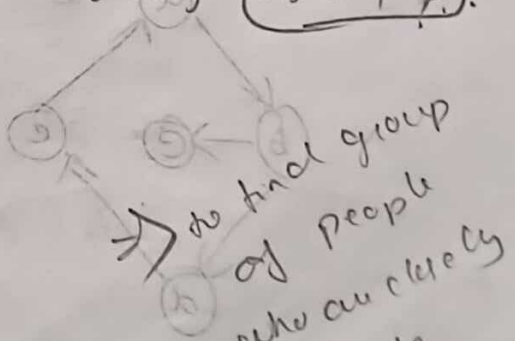
Stack : 3 7 6 5

② SCCs : 4 6 5

③ Visited : 0 3 2 1 4 6 5 7

Stack :

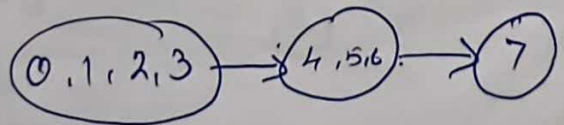
SCCs : 7



⇒ to find group of people who are directly connected

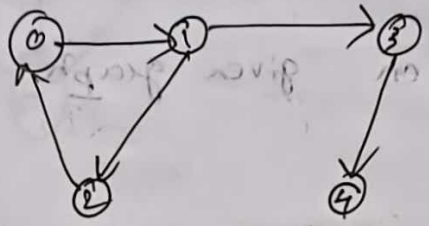
Step 4:

SCCs :



Strongly connected components in a graph.

eg: 1



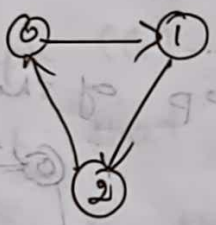
⇒ can reach from 0 to 1 & also 1 to 0

1-2 & also 2-0

0-2 & also 2-0

⇒ But 1-3 can be reached, but not 3->1

similarly 3-4 reached but not 4-3

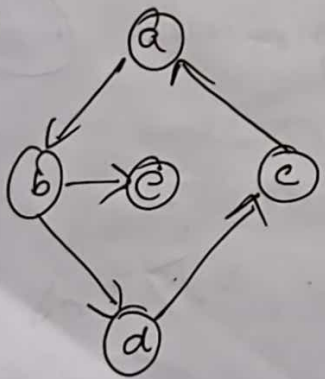


3

4

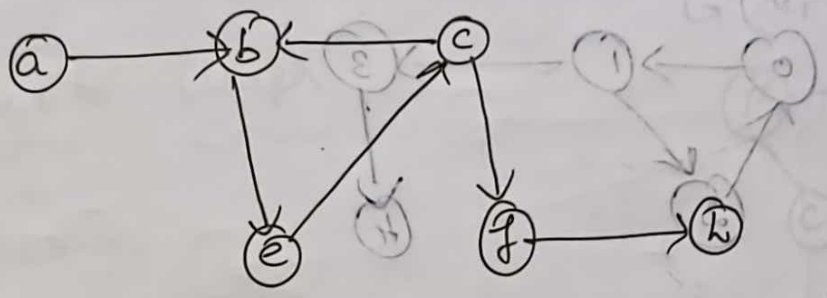
∴ This graph contains 3 strongly connected components.

eg: 2



[2 strongly connected components]

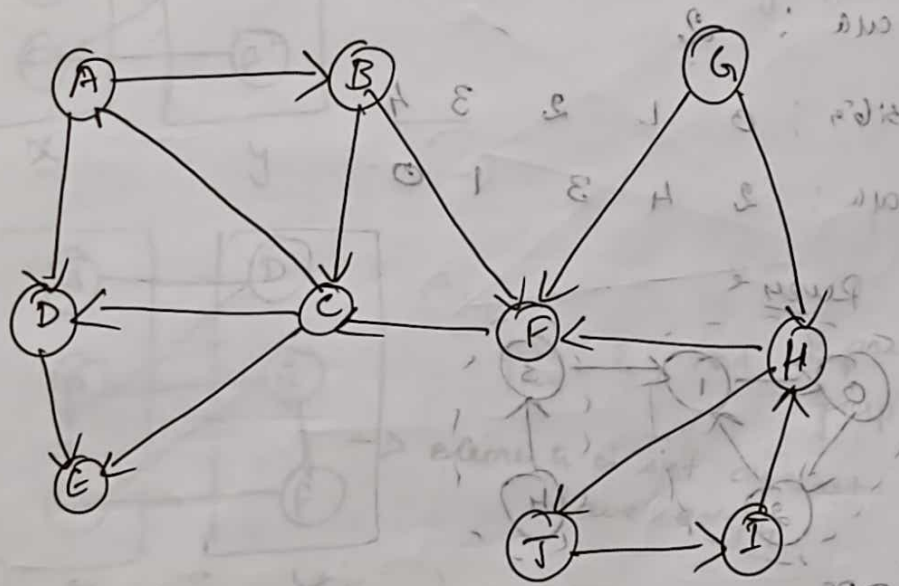
eg: 3

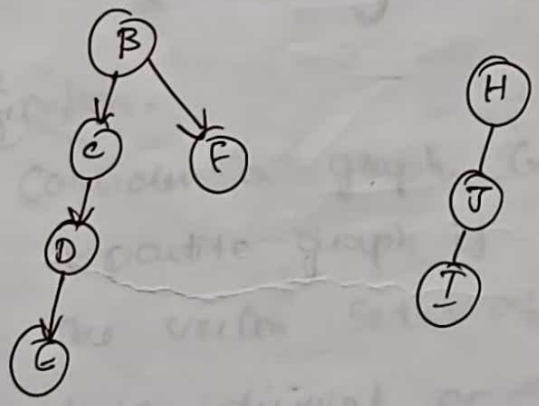


4 Strongly connected components

eg: 4.

(1)

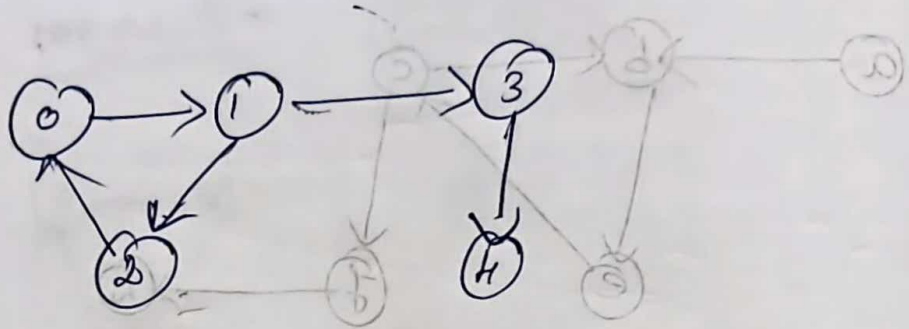




Strongly connected components are.

(1) BACF
(2) HIJ
(3) G
(4) D
(5) E.

1) eg: 1



Step 1: DFS

* Visited : 0 1 2

  Stack :

* Visited : 0 1 2

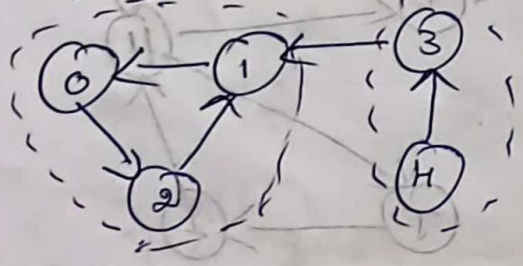
  Stack : 2

Backtrack

* Visited : 0 1 2 3 4

  Stack : 2 4 3 1 0

Step 2: Reverse



Step 3: DFS

* Visited : 0 2 1

  Stack : 2 4 3 1

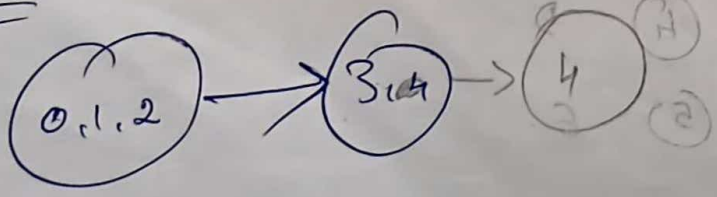
  SCCs : 0 1 2

* Visited : 0 2 1 3 4

  Stack : 2 4 3

  SCCs : 3 4

Step 4:

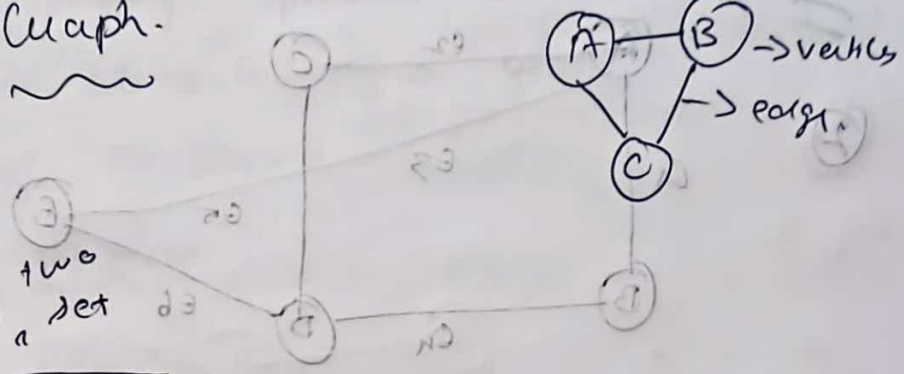


Application of BFS

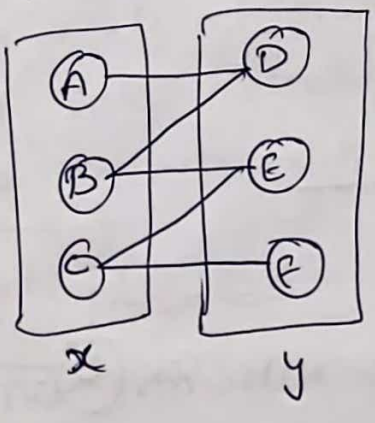
Graph
$G(V, E)$

Bipartite Graph.

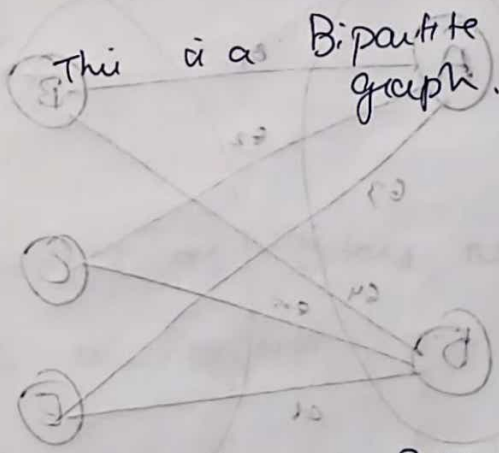
two partition.



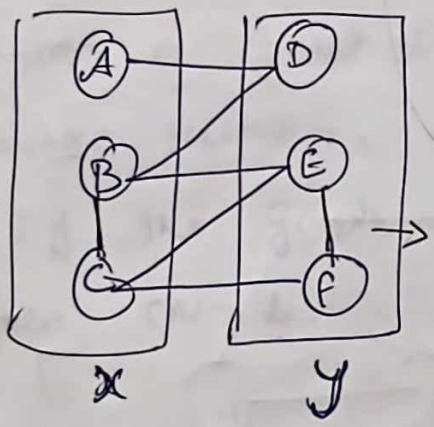
=> Graph into two set



This is a Bipartite graph.



=>



This is not Bipartite graph

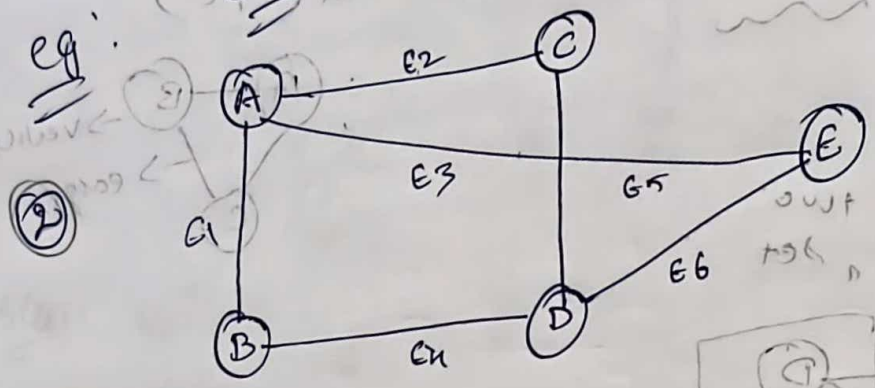
elements in set does not have edges.

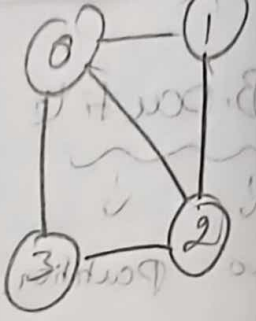
Definition.

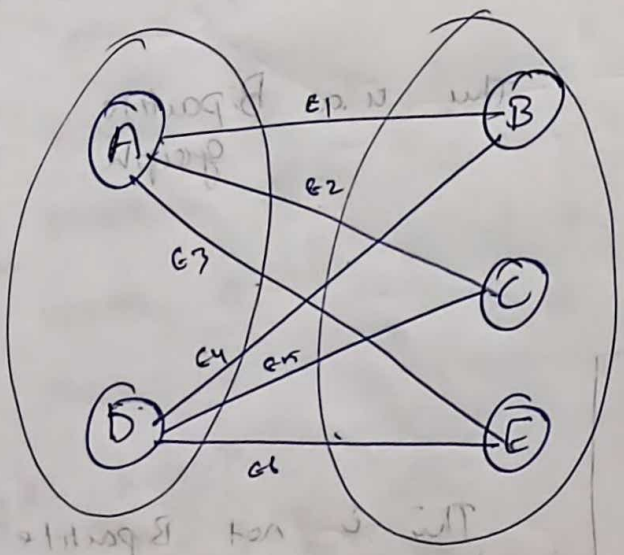
Consider a graph $G(V, E)$. The graph G is a bipartite graph if

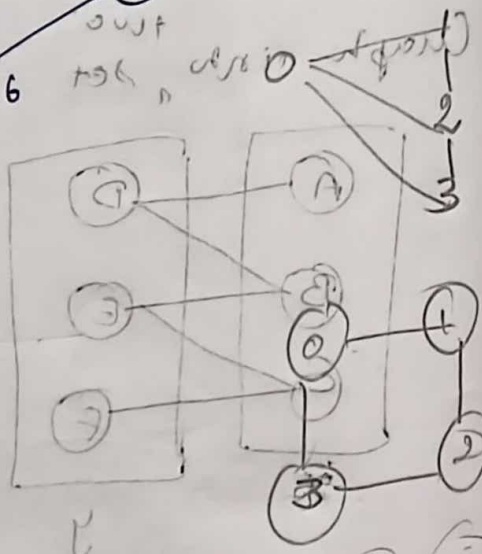
* The vertex set V of G can be partitioned into two disjoint and independent set V_1 & V_2.
* All the edges from edge set E have one endpoint vertex from the set V_1 & another endpoint vertex from set V_2.

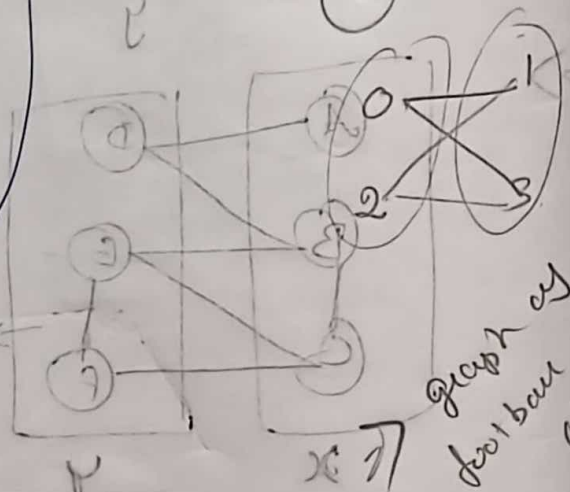
eg:











$V_1 = (A, D)$

$V_2 = (B, C, E)$

∴ Each edge has one endpoint in V_1 & another endpoint in V_2. There is no edges whose both endpoints belong to V_1 or V_2.

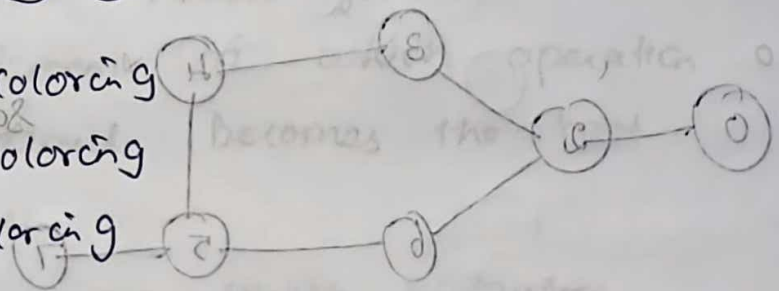
∴ we conclude that graph G_1 is a bipartite graph.

Graph coloring

Graph coloring problem is to assign colors to certain elements of a graph subject to certain constraints.

Graph coloring problem:

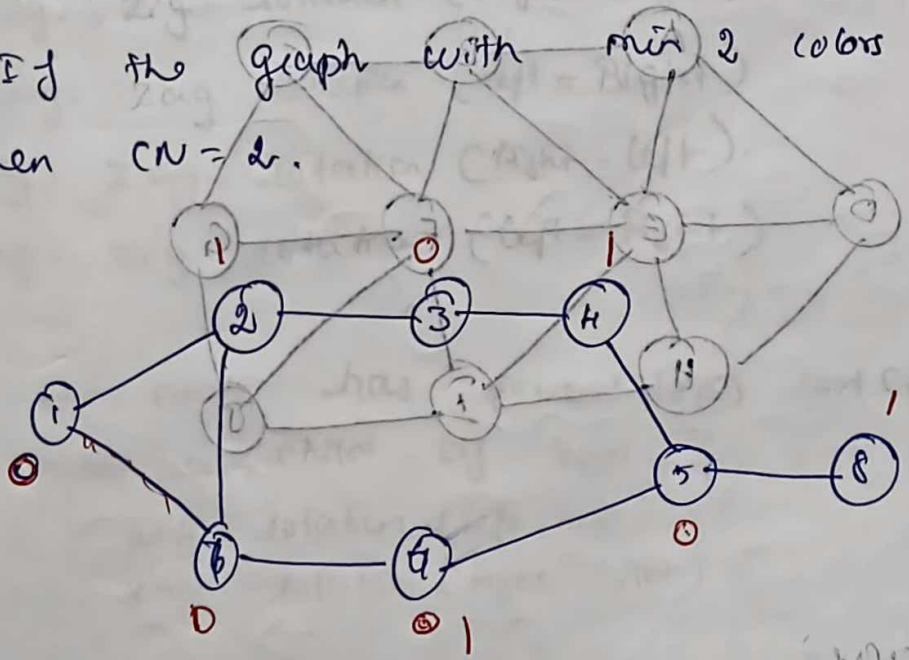
* Vertex coloring
* Edge coloring
* Face coloring

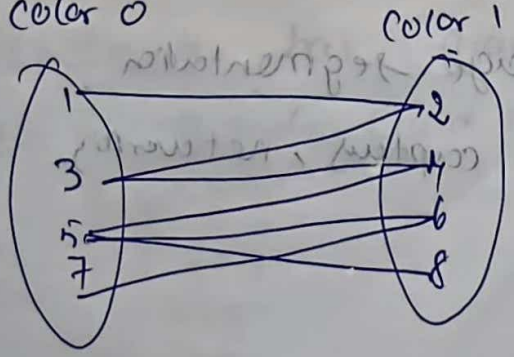


Chromatic Number:

* The smallest number of colors needed to color a graph G is called its chromatic number.
* If the graph with min 2 colors then CN = 2.

eg:



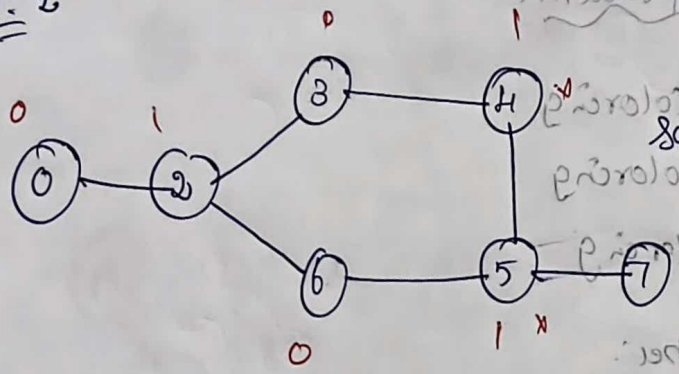


0 - red
1 - blue.

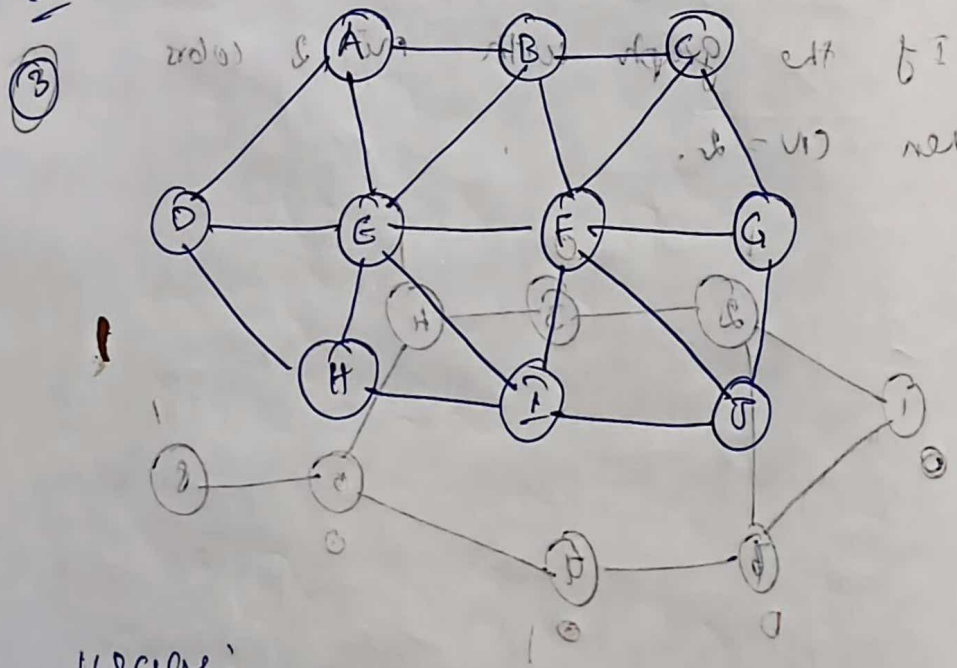
Properties:

1. Bipartite graph can be colored by 2 colors.
2. Contain even cycles.

eg: 2



eg: 3



Usage:

Data mining, image segmentation, clustering, image capturing, networking.

Unit - IV

Advanced Trees

balancing.

* Splay tree is defined as the self-adjusted tree in which any operation performed on the element would rearrange the tree. so that the element on which operation has been performed becomes the root node of the tree.

* Splay trees are roughly balanced.

Six types of rotations:

1. Zig rotations (Right)
2. Zag rotations (Left)
3. Zig - Zig rotation (Right - Right)
4. Zag - Zag rotation (Left - Right)
5. Zig - Zag rotation (Right - Left)
6. Zag - Zig rotation (Left - Right)

Case: 1

If the node has parent (ie) root (or) search node is child of root. (No grand parent).

Zig rotation (Left child)
Zag rotation (Right child)

Case: 2

If the node has grand parent (or) search node is grand child of root

Zig - Zig
Zag - Zag
Zig - Zag
Zag - Zig

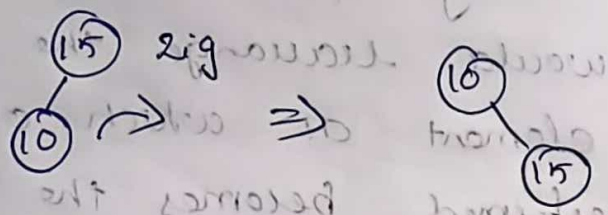
Splay Insertion:

15, 10, 14, 7, 13, 16

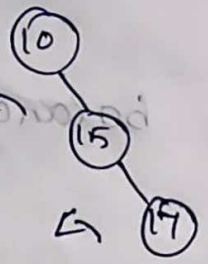
Zig - right
Zag - left

Step 1:

Step 2:

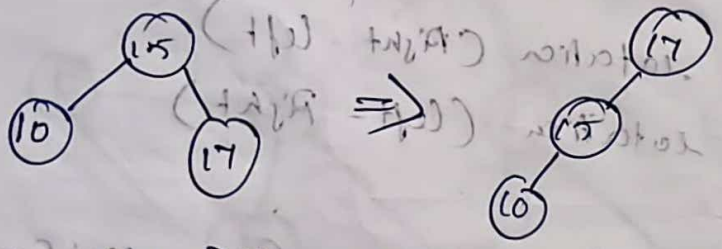


Step 3:



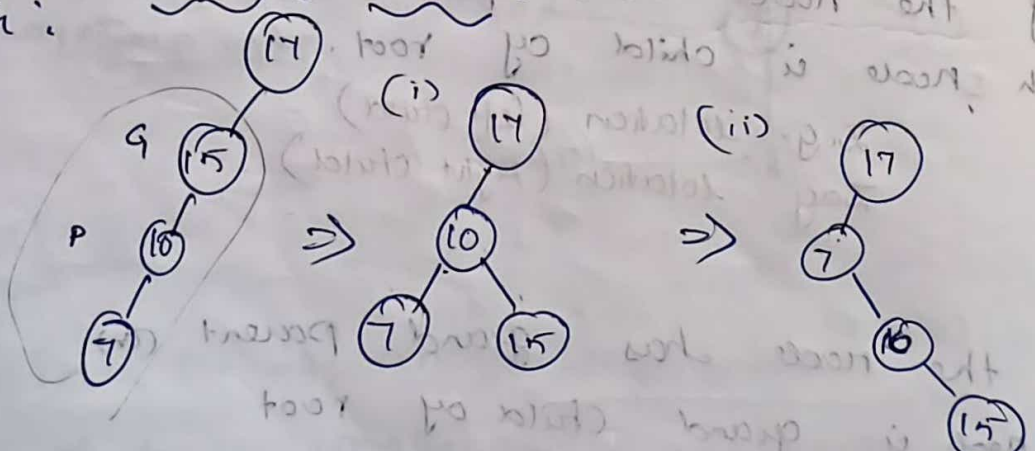
Bring 14 to root ⇒ Zag-Zag

Pull down the grandparent first, then pull down the parent

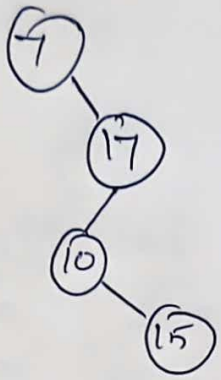


Step 4:

Zig-Zig rotation.



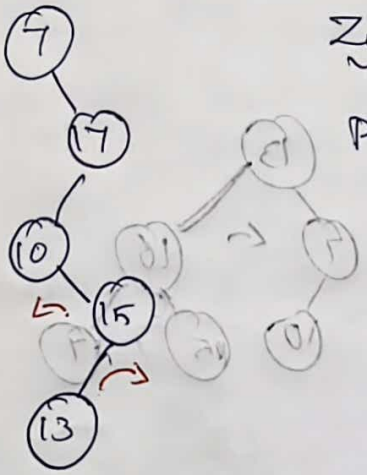
(iii)



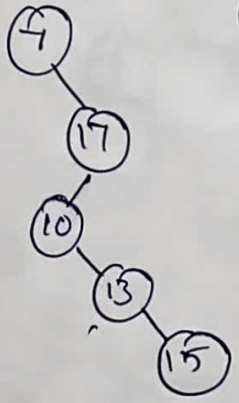
Step 5 :

Zig - zag Rotation.

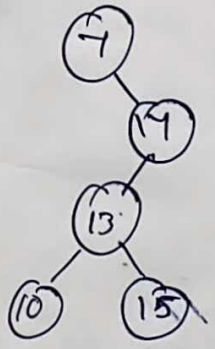
pull down parent first then grandparent.



(i)

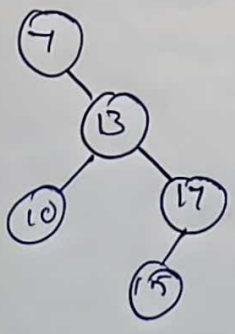


(ii)

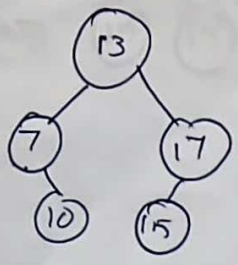


Again zig - zag

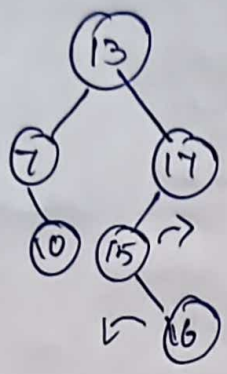
(i)



(ii)



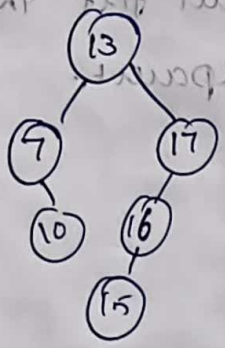
Step 6:



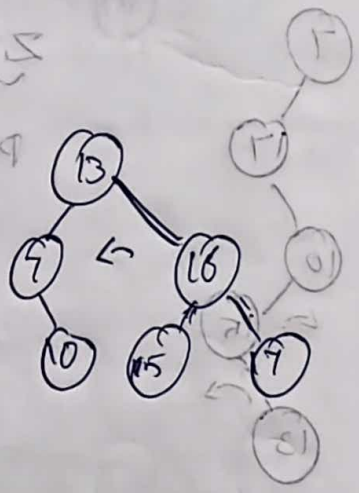
(ii)

Zag-zig rotation.

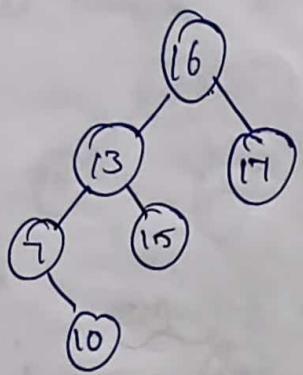
(i)



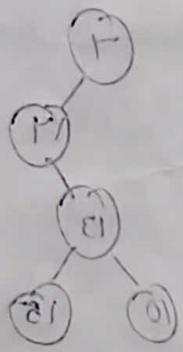
(ii)



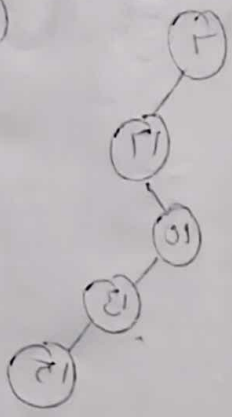
Zig-zag rotation.



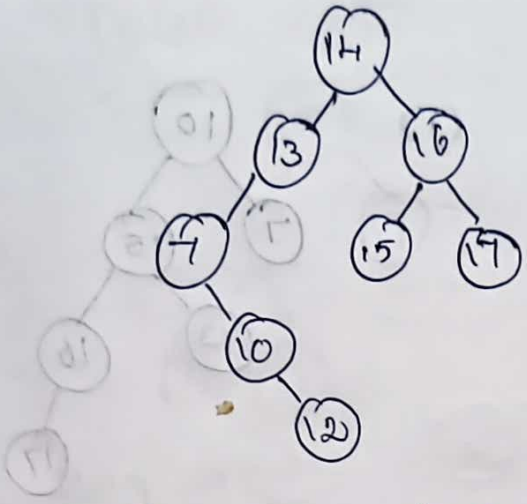
(i)



(ii)



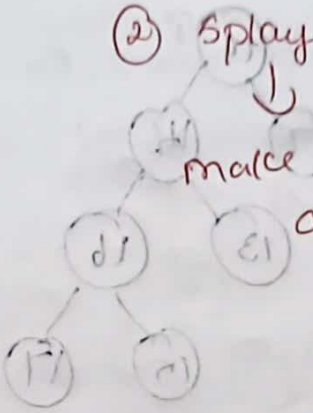
Splay Deletion:



① Standard BST Deletion

② Splaying

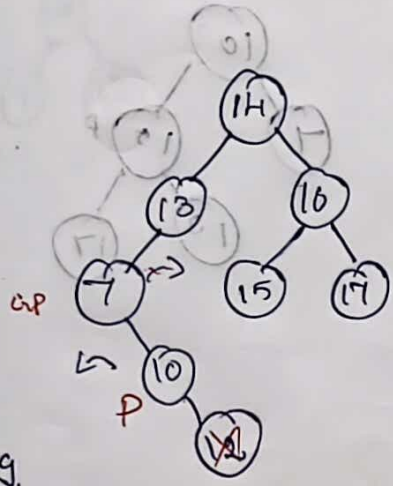
make parent node as root node.



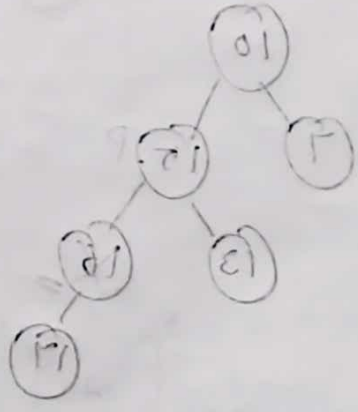
Delete 12, 14, 16, 20, 17

Step 1:

Delete leaf node "12"

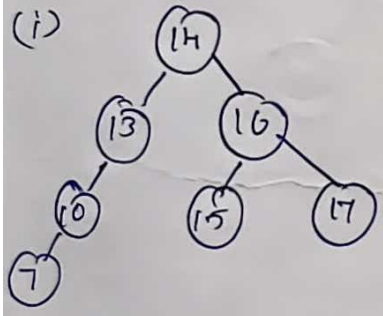


(i)

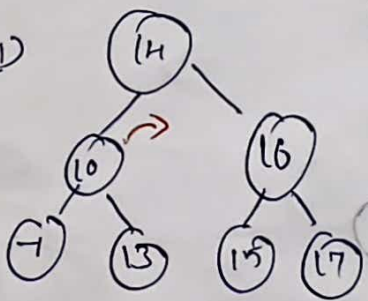


Zag-Zig

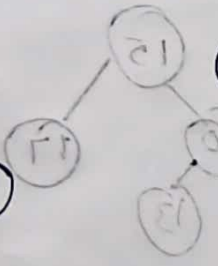
(i)



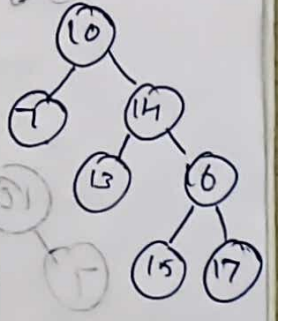
(ii)



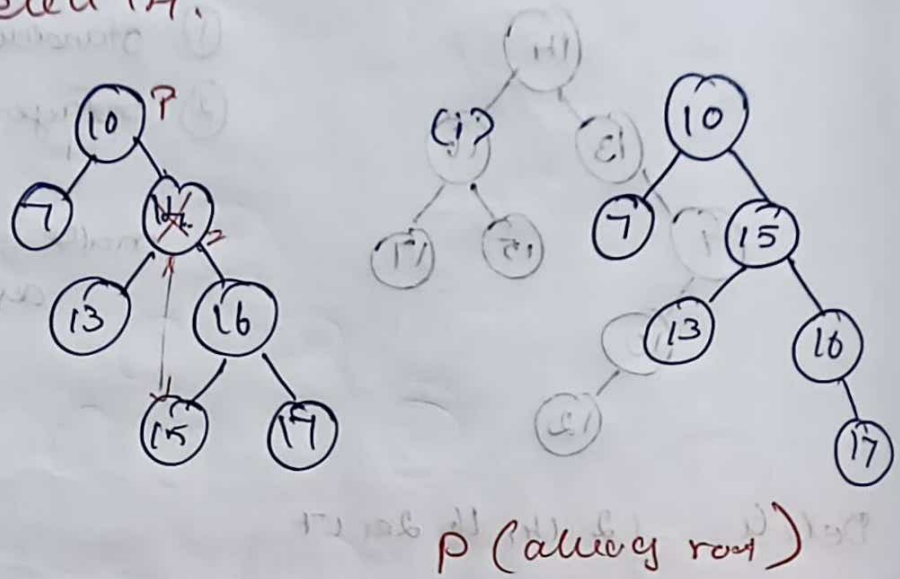
(iii)



Zig

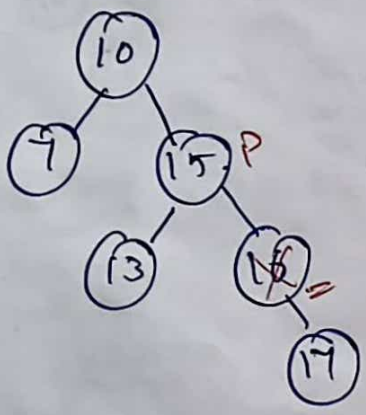


Step 2: Delete 14.

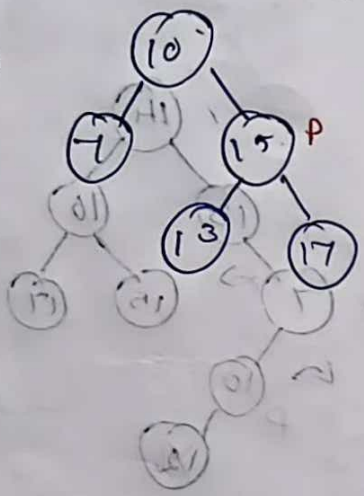


P (already root)

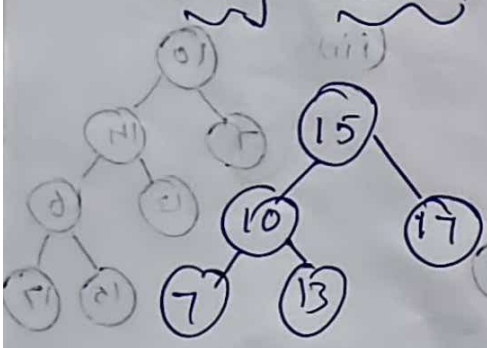
Step 3: Delete 16.



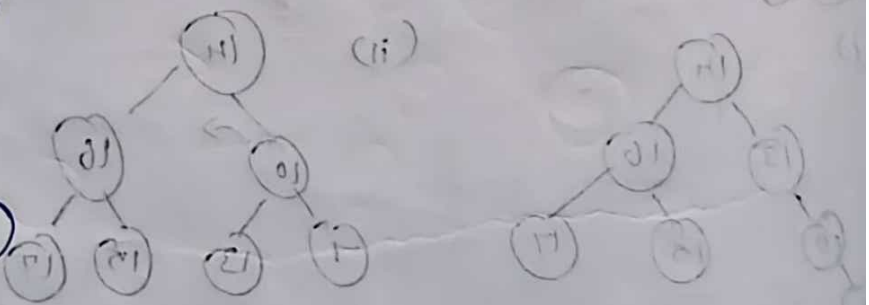
(i)



Zag rotation

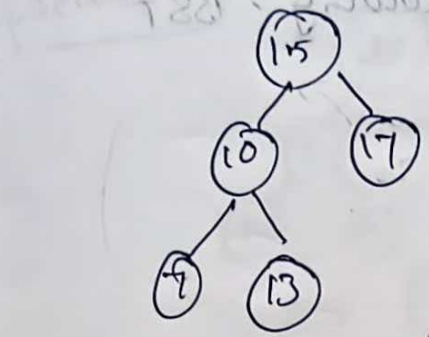


(ii)



Step 4:

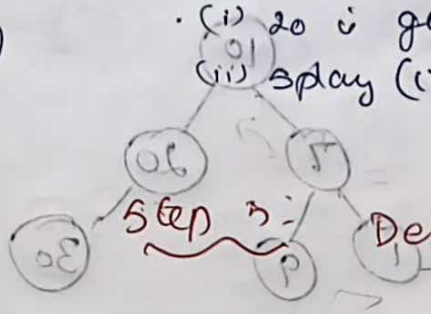
Delete 20



=> Search for 20 (Not available)

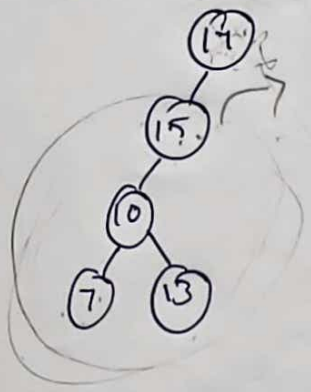
(i) 20 is greater than 17

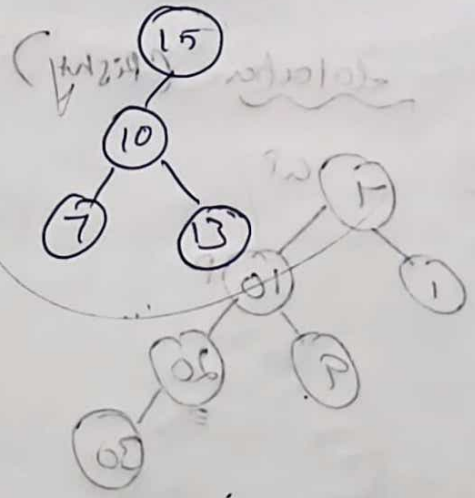
(ii) Splay (17)



Step 3: Delete 17

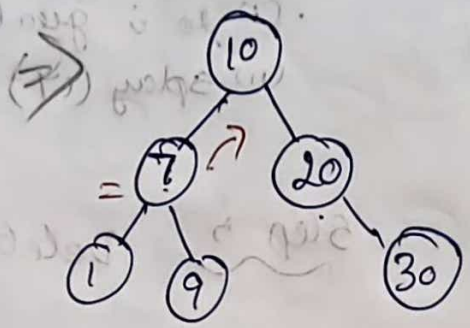
Zag rotation.



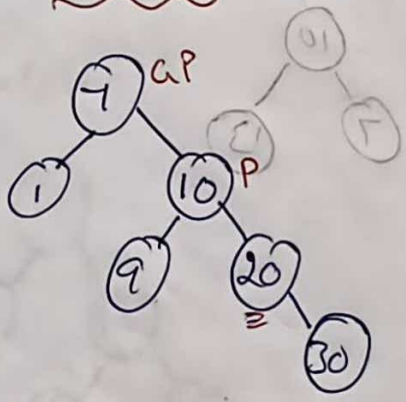


Searching in Splay Tree.

I) Search 7 in the following BST

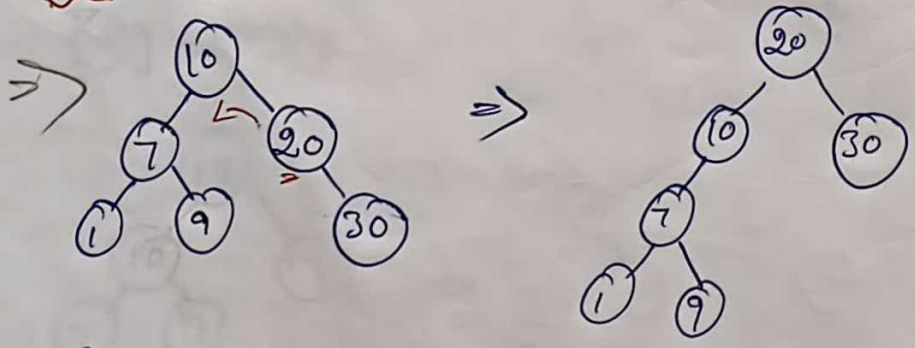


1. Zig rotation (Right)



II) Search 20 in the above BST.

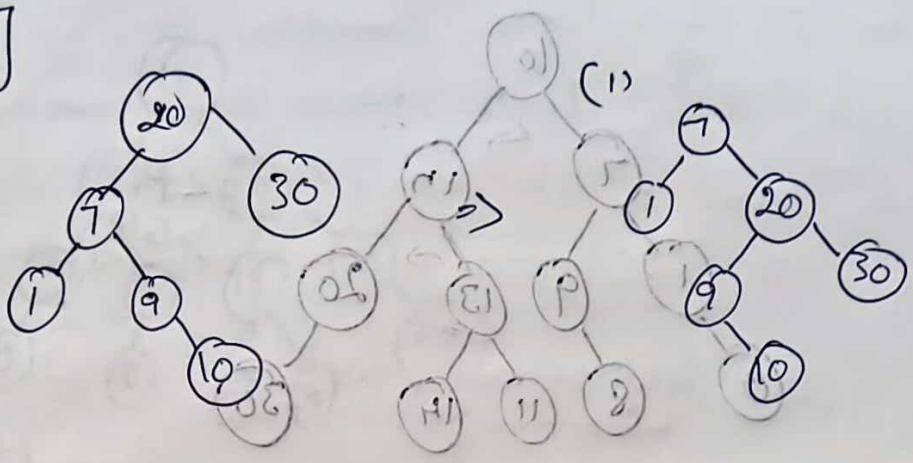
2. Zag rotation (Left)



III) Search 1 in the above BST.

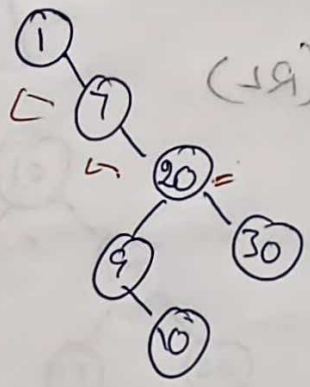
3. Zig - Zig (RR)

Search 1



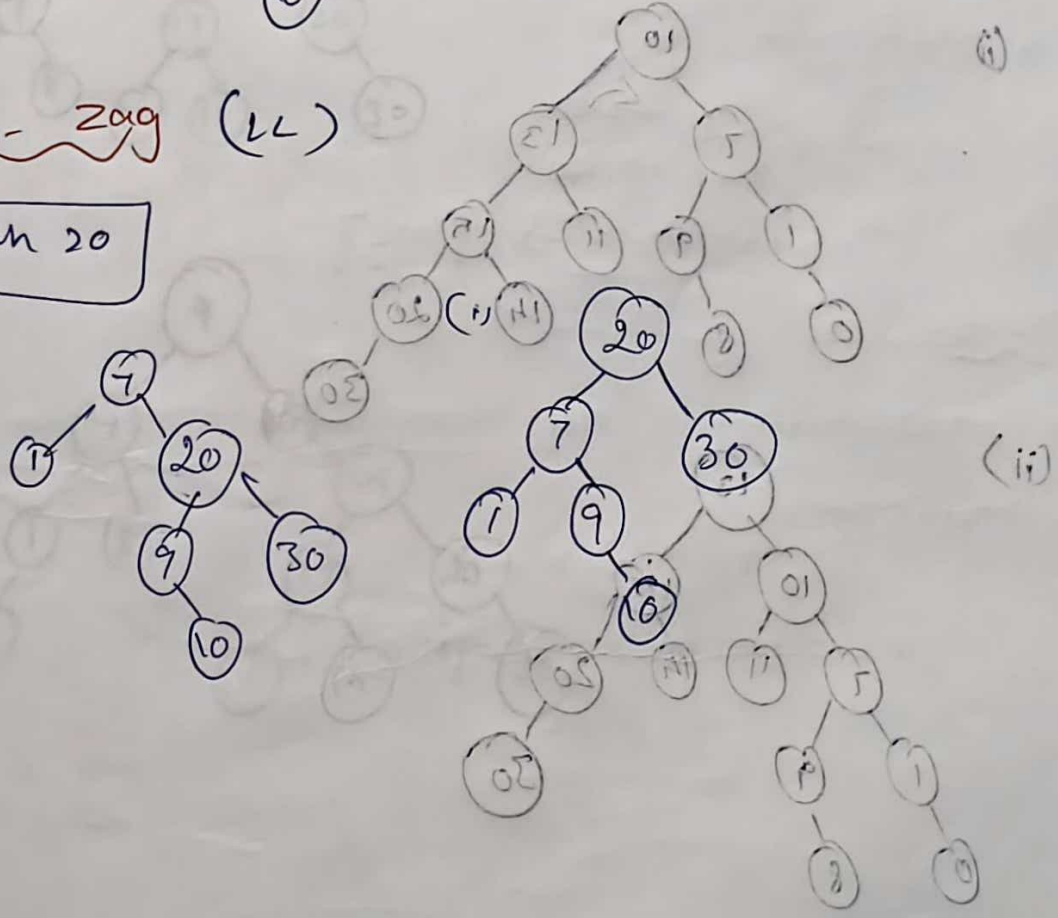
(i)

=> (ii)



4. Zag - Zag (LL)

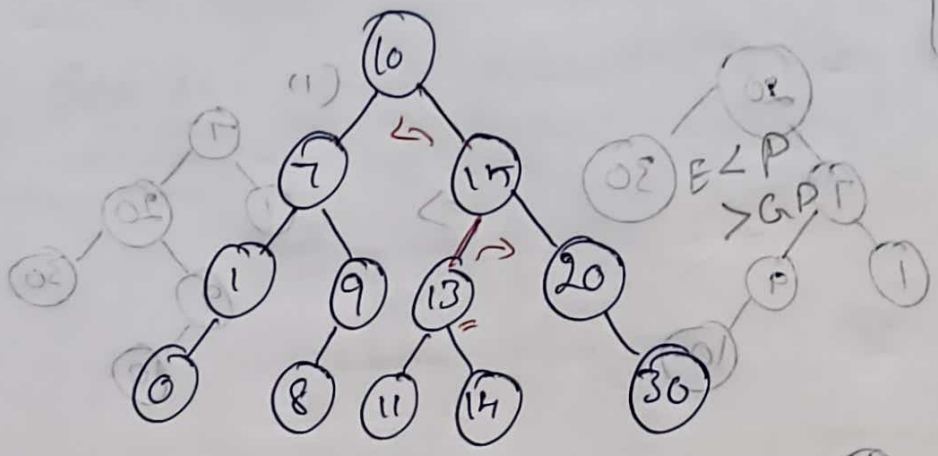
Search 20



(i)

(ii)

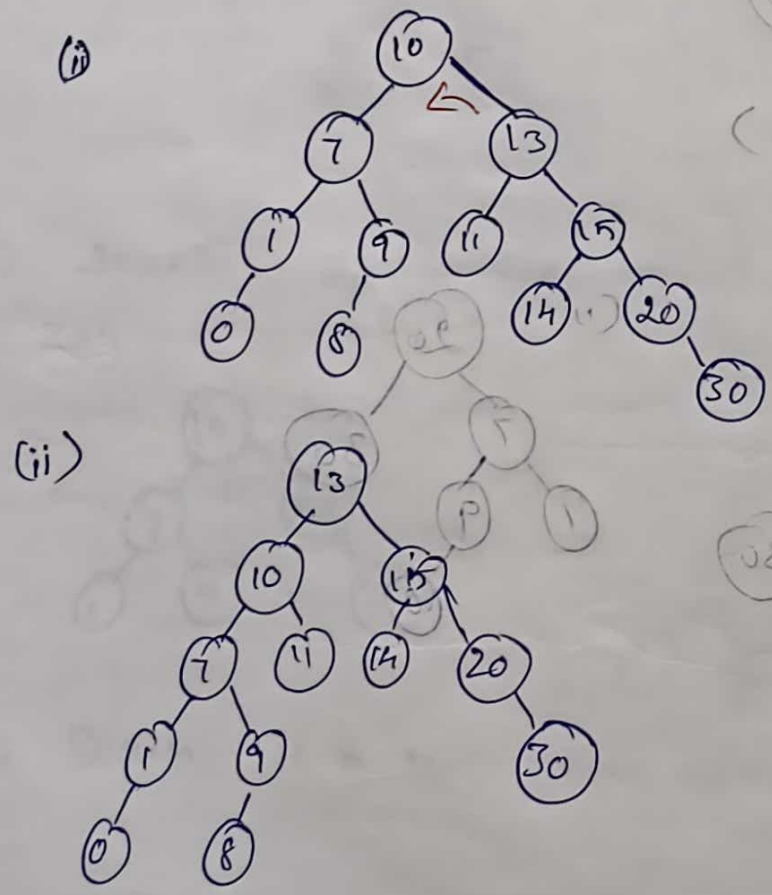
I RBT



1) Zig-zag rotation. (RL)

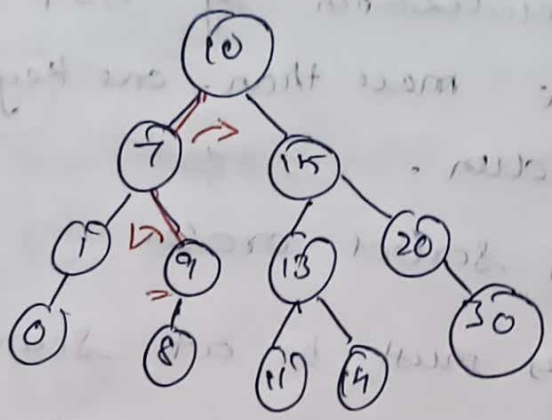
Find 13

pull parent 1st



(ii)

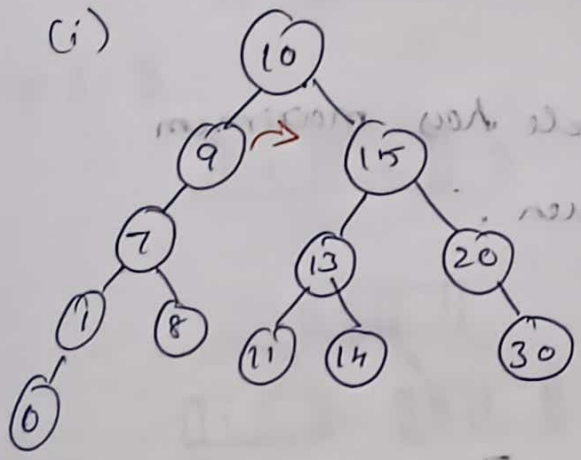
2) Zag - zig rotation. (LR)



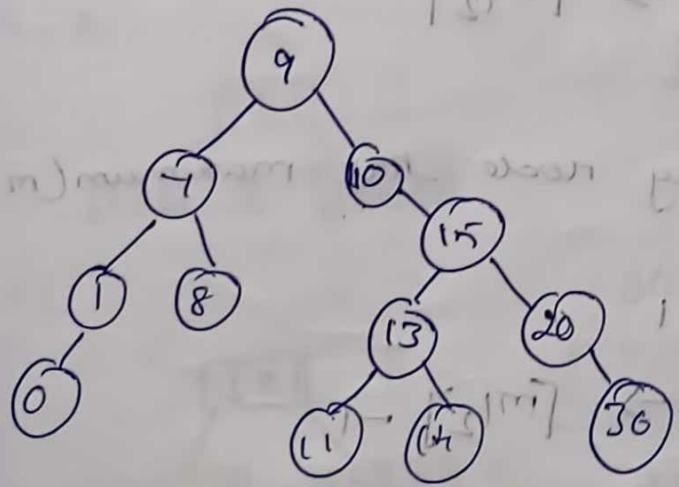
E > P
< GP

Search 9

(i)



(ii)



B-Tree.

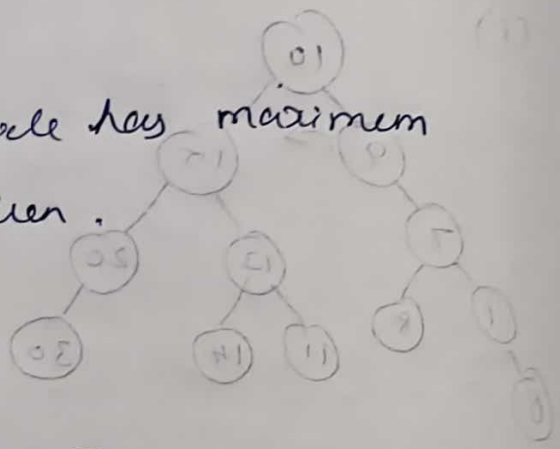
- It is a generalization of BST in which a node can have more than one key and more than 2 children.
- It maintains sorted-order
- All leaf nodes must be at same level.

Properties:

⇒ Children:

* Maximum = Every node has maximum "m" children.
* Minimum

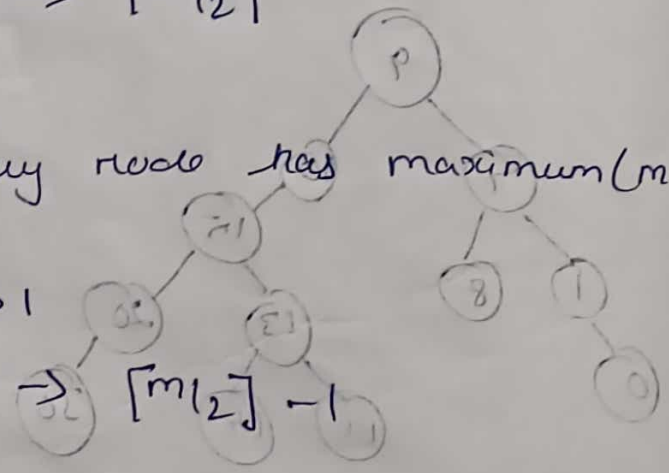
leaf → 0
root → 2
Internal node → $\lceil m/2 \rceil$



⇒ Keys:

* maximum = Every node has maximum $(m-1)$ keys
* minimum

root → 1
All other nodes → $\lceil m/2 \rceil - 1$



1) Insertion in B-Tree.

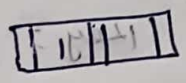
Insert elements in B-Tree of order-3 from 1 to 10.

m = 3 (children)

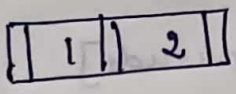
maximum keys = $m-1 = 3-1 = 2$

Key = 2

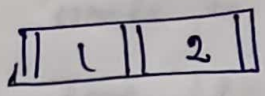
Step 1: 1



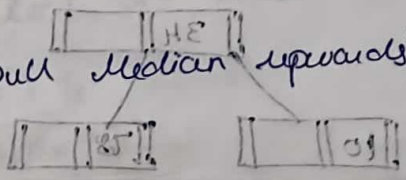
Step 2: 2

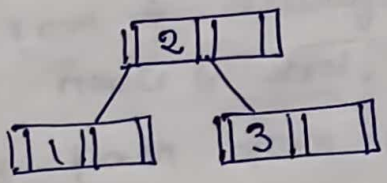


Step 3: 3

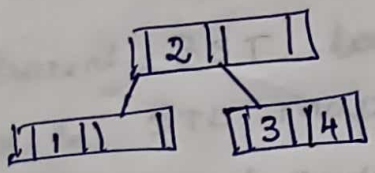


[push Median upwards]

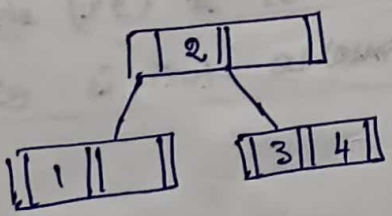




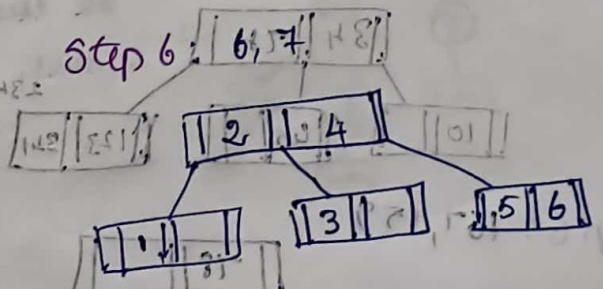
Step 4: 4



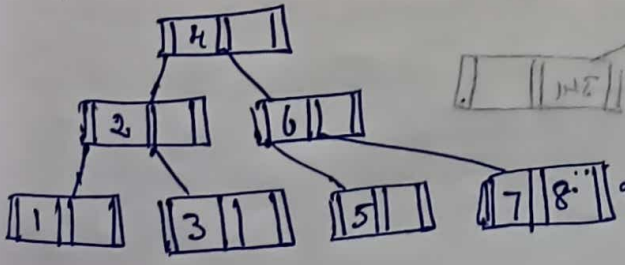
Step 5: 5



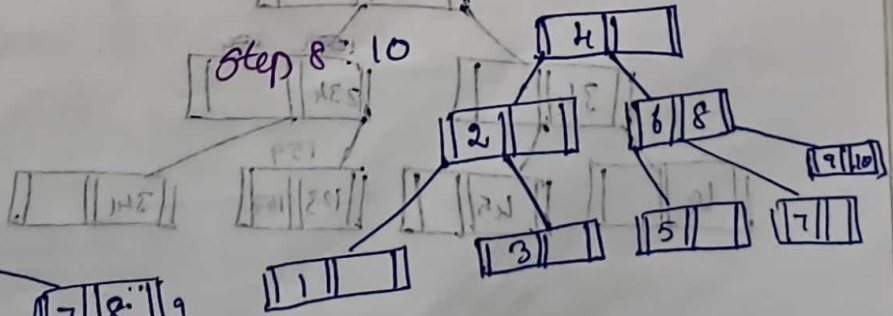
Step 6:



Step 7: 8, 9



Step 8: 10



Example 3:

Construct a B-tree of order 3 by inserting the following numbers.

10, 34, 78, 45, 123, 341, 234, 167, 159, 52, 83

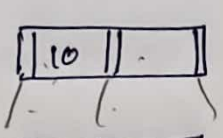
max children (order) = 3

Key = 2

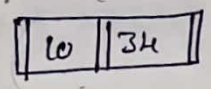
min = ⌈3/2⌉ − 1 = 1

children = ⌈3/2⌉ = 2

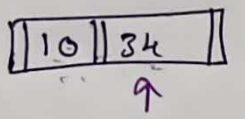
Step 1: 10



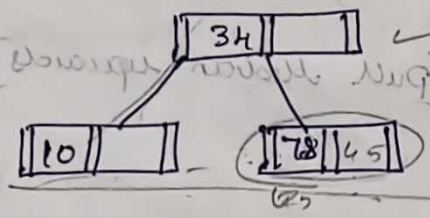
Step 2: 34



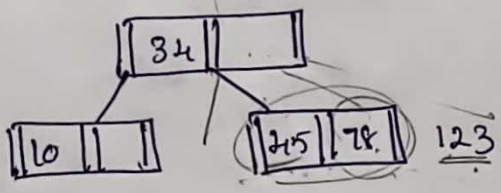
Step 3: 78



[pull upwards]



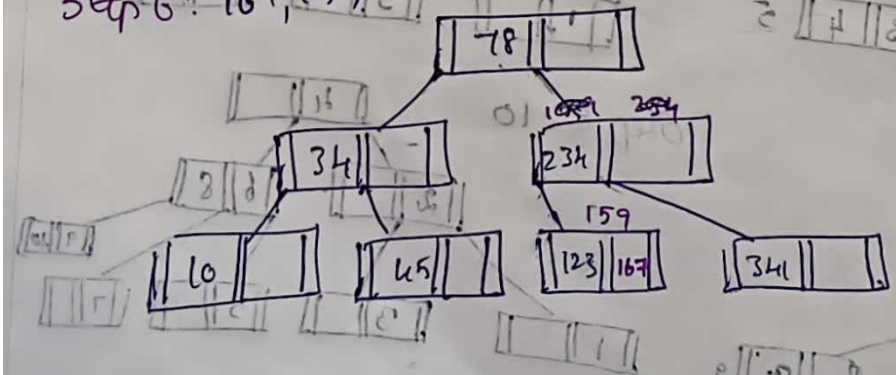
Step 4: 45, 123



[pull upwards]

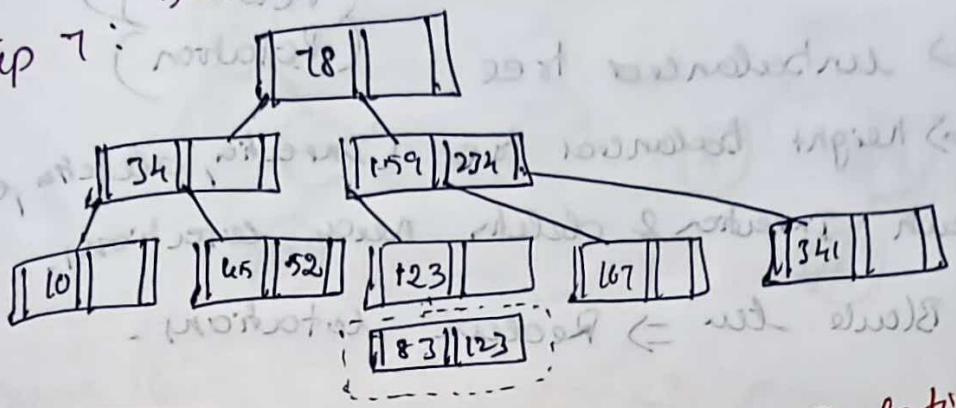
Step 5: 341, 234

Step 6: 167, 159



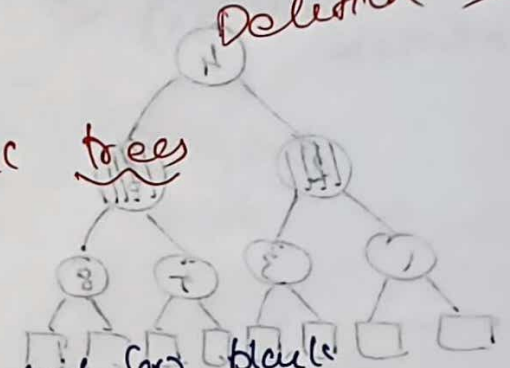
52, 83

Step 7 :



Deletion →

Red - Black trees



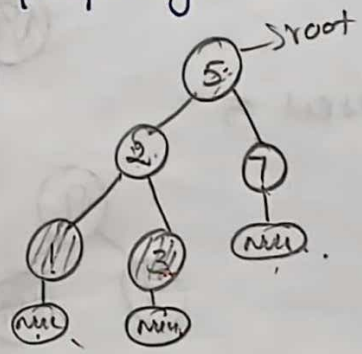
property

1. Every node is either red (or) black
2. Every leaf (NULL Pointer) is black.
3. The root is always black
4. If a node is red, then both children are black.
5. Every path from node to descendent leaf contain the same number of black nodes.

⇒ Balanced BST. based on color property.

⇒ Used in STL map operations.

⇒ Children of red node must be black (i.e) 2 consecutive red nodes is not allowed

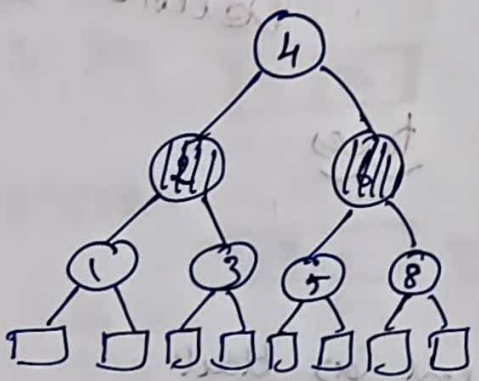


→ BST → unbalanced tree !! (Recoloring) (Rotation)

→ AVL ⇒ height balanced tree (Insertion, deletion, etc)

→ ↳ each Insertion & deletion need rotations.

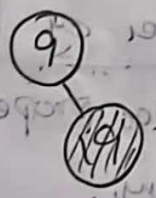
⇒ Red Black tree ⇒ Recolor rotations.



Creation on Red Black tree

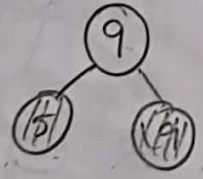
eg: 1 9, 19, 5, 15, 18, 25, 40, 90

Insert 9, 19

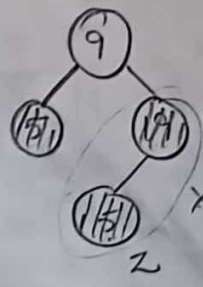


> If a new node comes it must be colored.

Insert 5

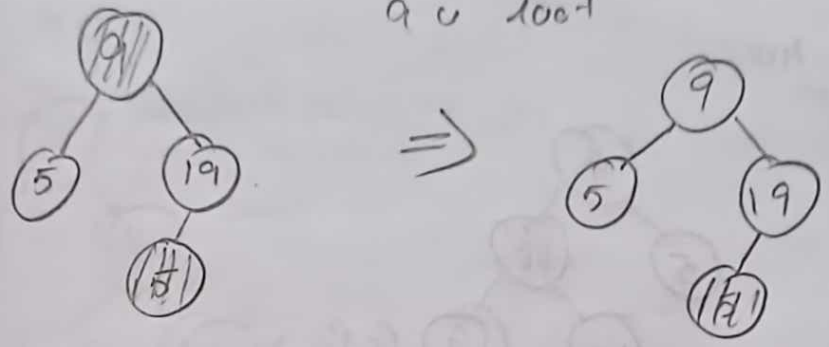


Insert 15

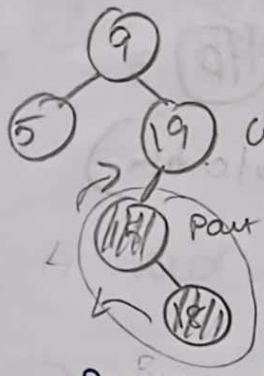


* Recoloring.
* If uncle is red
  - Set parent color = black
  - uncle color = black
  - Grand parent = red

9 is root



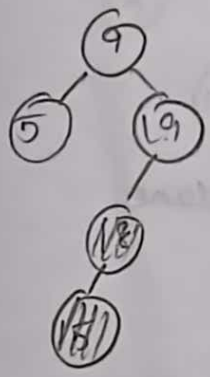
Insert 18

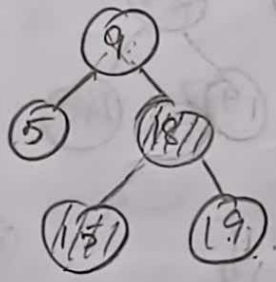


=> (no uncle rotation)

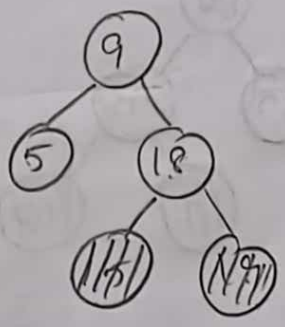
2 aq 2 : 9
L R

After Left - Right Rotation.



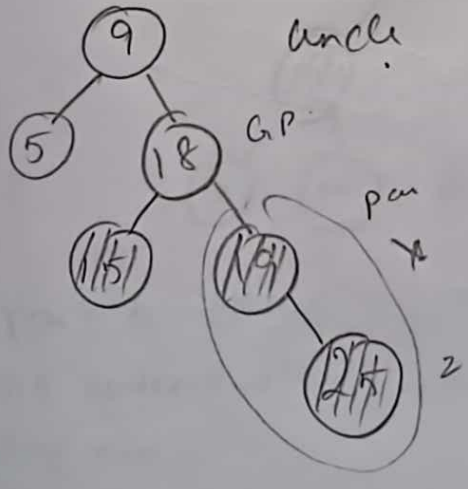


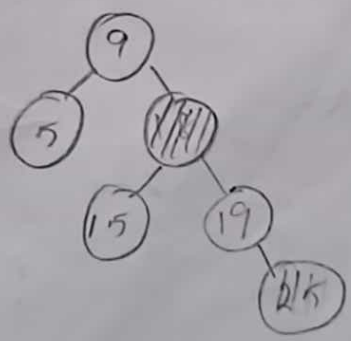
Inserted to balance tree we violating children.



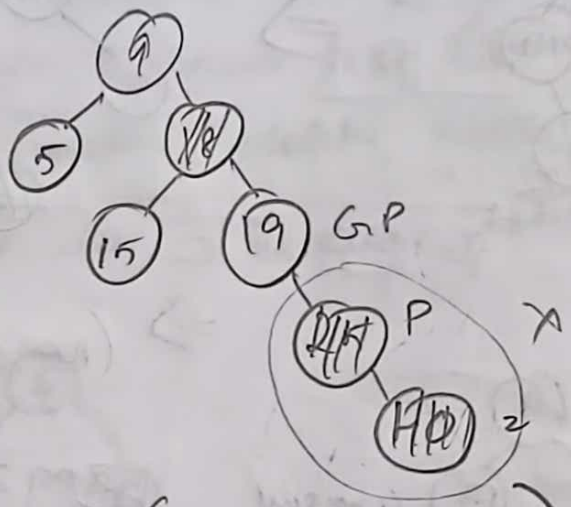
Insert 25

=> check for uncle.

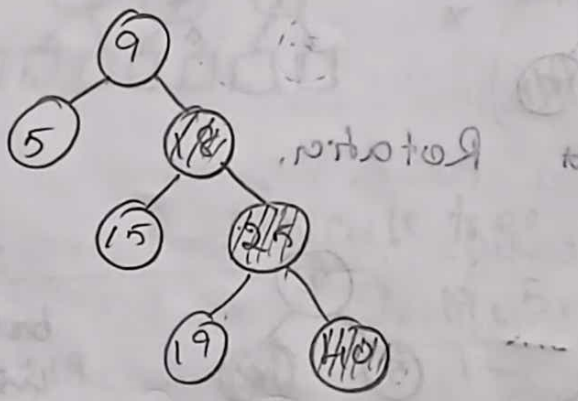




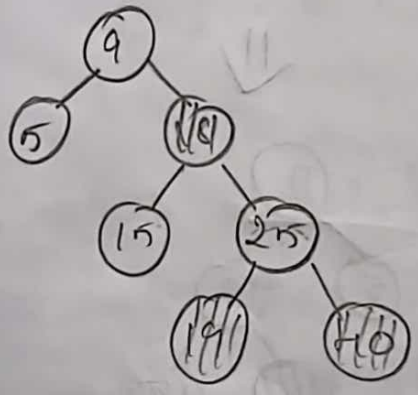
Insert 20:



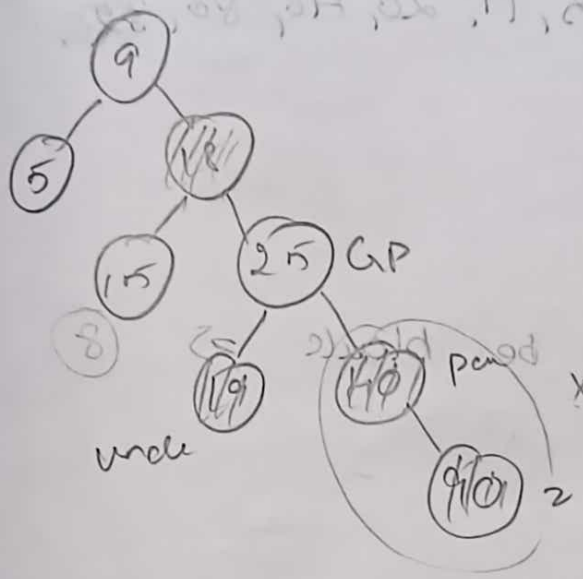
=> No uncle (LL Rotation)



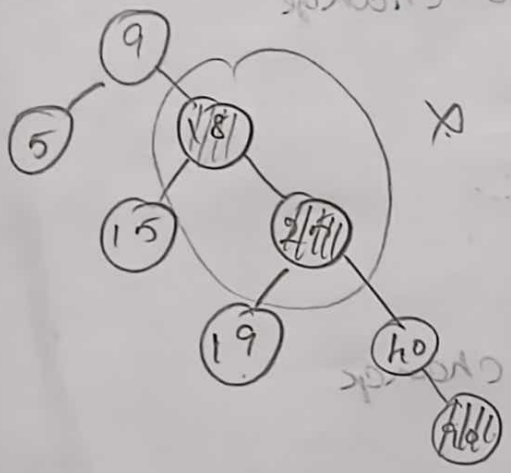
Inorder no balance



Insert 90

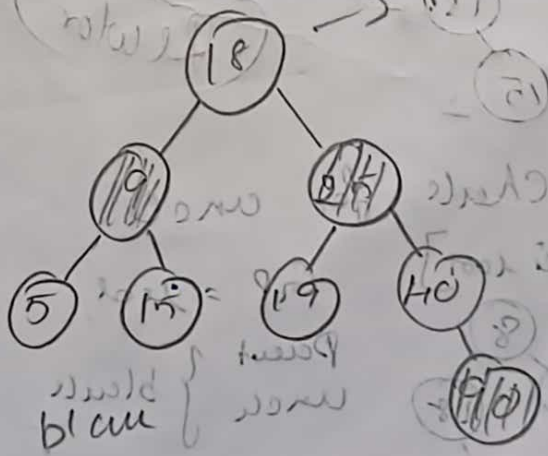


=> uncle is red -> Recoloring



⇓

after left rotation



=> root is black

=> no consecutive red nodes

leaf nodes

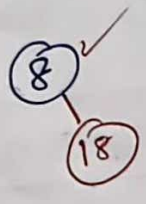
Insertion of:

8, 18, 5, 15, 17, 20, 40, 80, 30.

Insert 8:

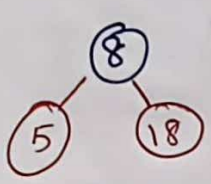
leaf node must be black => (8)

Insert 18:



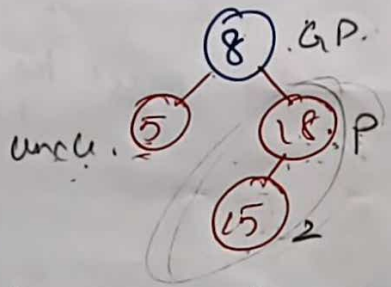
Parent black no change

Insert 5:



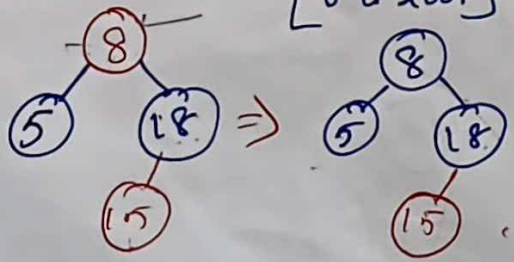
Parent black no change

Insert 15:



(1) Parent is red, check for uncle

[8 is root]



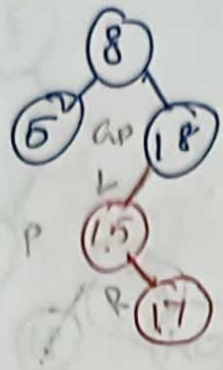
GP = red.

Parent } black

uncle }

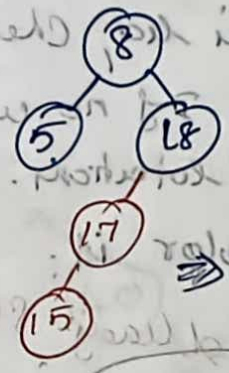
z = red

Insert 17 :



(Rotation + Recolor)

* Parent is red, check for uncle.
* No uncle, make rotations. 2rg 2rg

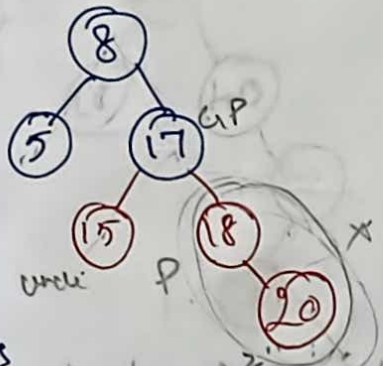


Right Rotate

RR

* After recoloring

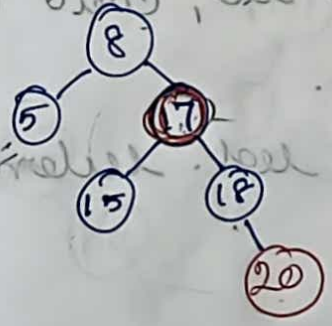
Insert 20:

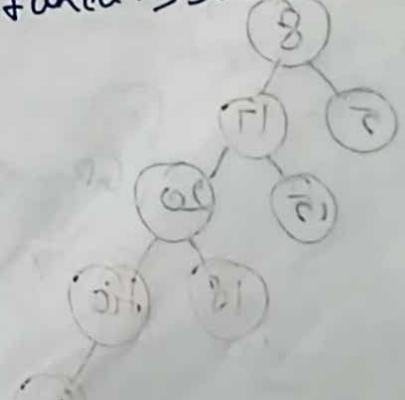


L Parent is red, check for uncle.

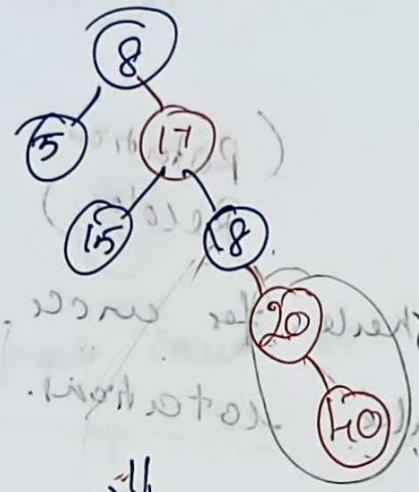
* uncle is red - recoloring.
  G.P red
  P + uncle -> black.

Recoloring





Insert 40:

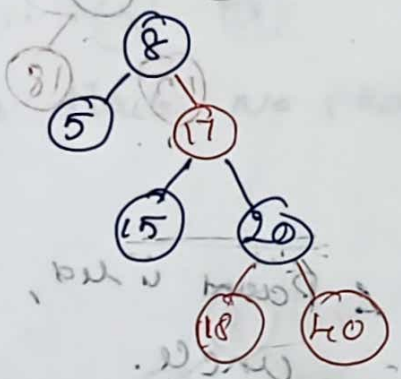


⇒ Parent is red, check for uncle, If no uncle make rotation. then recolor it.

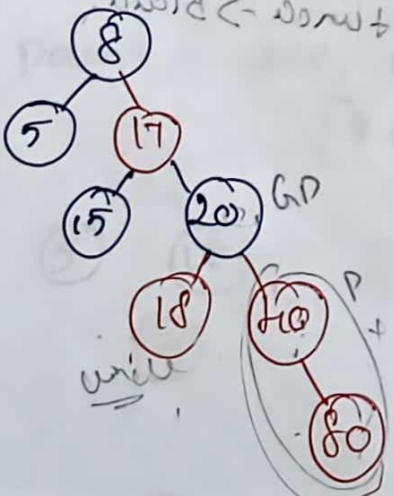
⇒ Already straight LR.

⇒ rotate left

recolors

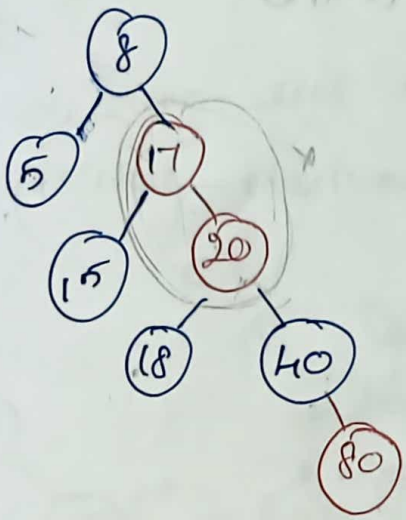


Insert 80:

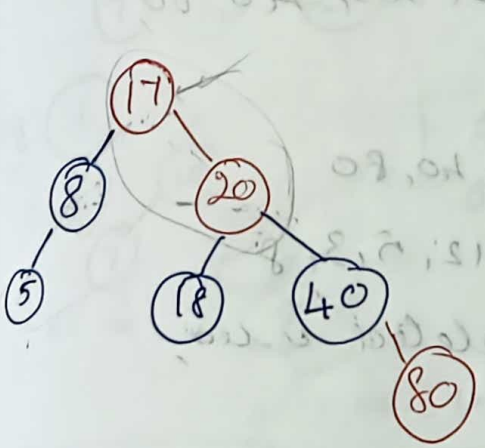


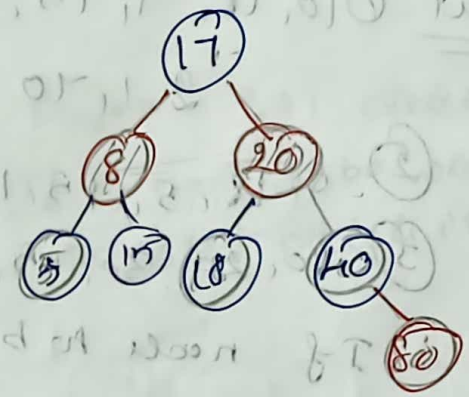
⇒ Parent is red, check for uncle.

⇒ uncle red. recoloring



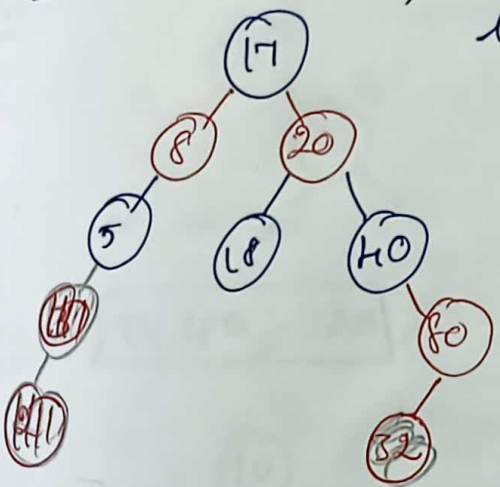
Make rotations:



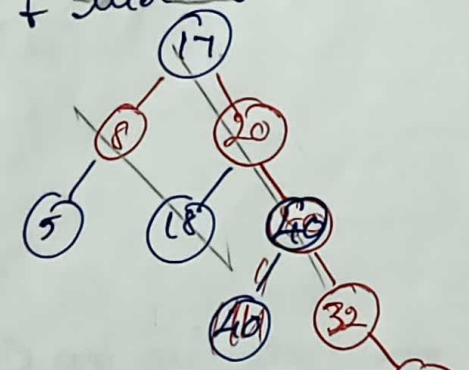


Insert 3, 2, 1

⇒ Parent is red no uncle make rotations & recolors



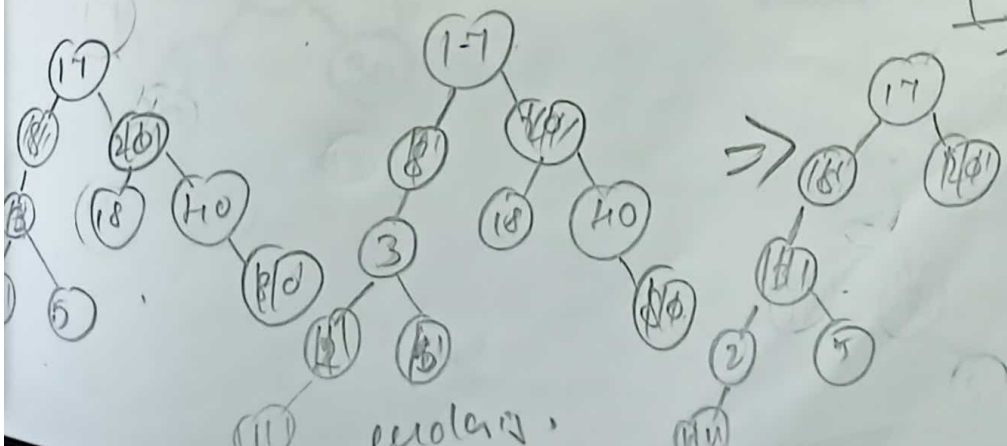
⇒



Insert 1

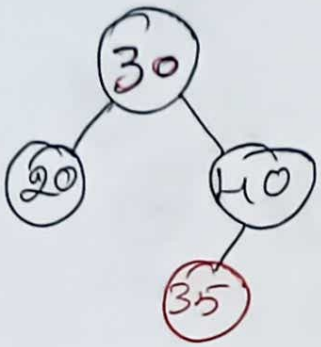
uncle red

recolors.



recolors.

Deletion in Red Black tree:



Delete: 20

Insert: 10, 18, 7, 15, 16, 30, 25, 40, 60,

2, 1, 70

(2) 8, 18, 5, 15, 17, 25, 40, 80

(3) 10, 20, 30, 15, 25, 12, 5, 3, 8

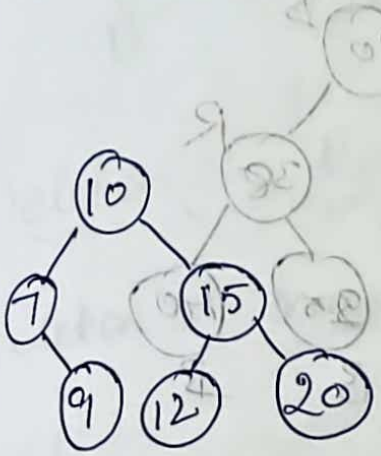
Case 1: If node to be deleted is red,

just delete it

Case 2:

Deletion:

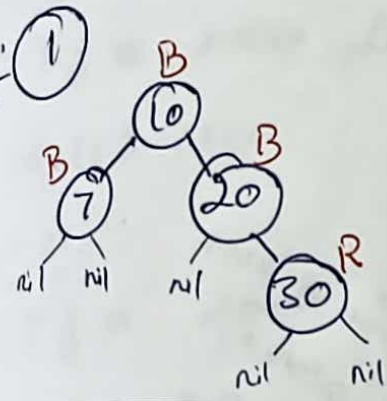
* If a deleting red node - no problem
* If deleting black node - problem [rotations recoloring]



BST

* leaf node no problem
* Node with one child -> Swap
* Node with two child -> inorder successor, inorder predecessor

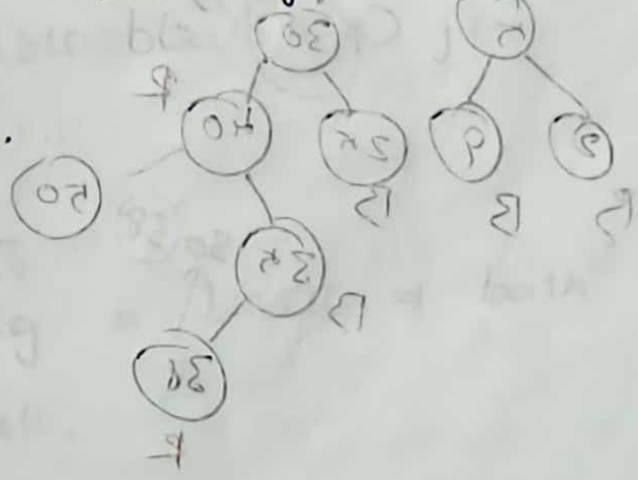
eg: 1



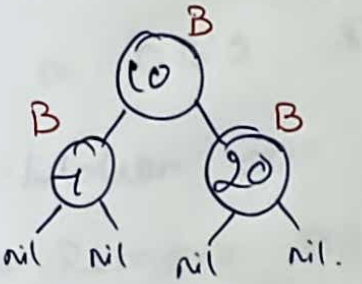
Step 1: Perform BST deletion

Step 2: If a node to be deleted is red just delete it.

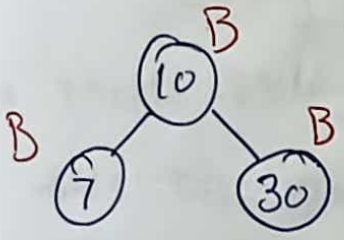
Delete 30



ase 1:

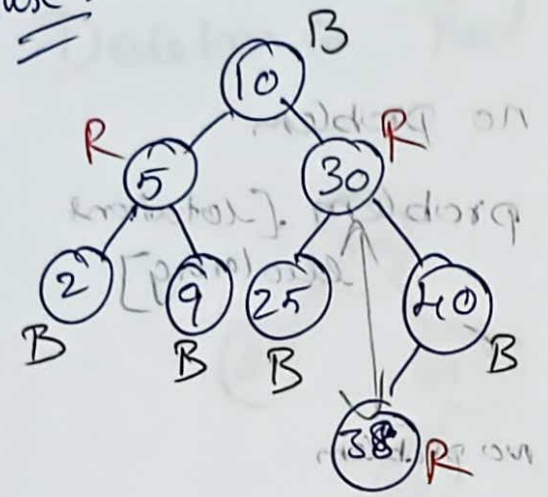


ase 2: Delete 20

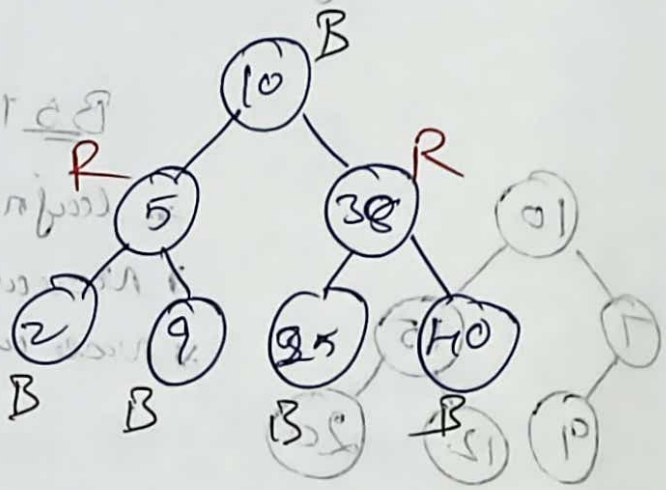


=> we are replacing the value only, not the color.

Case 3:

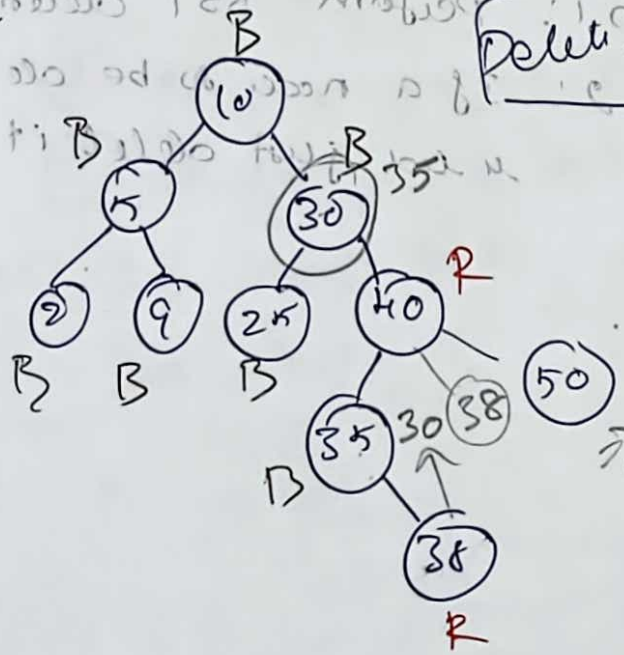


Delete 30



Case 4:

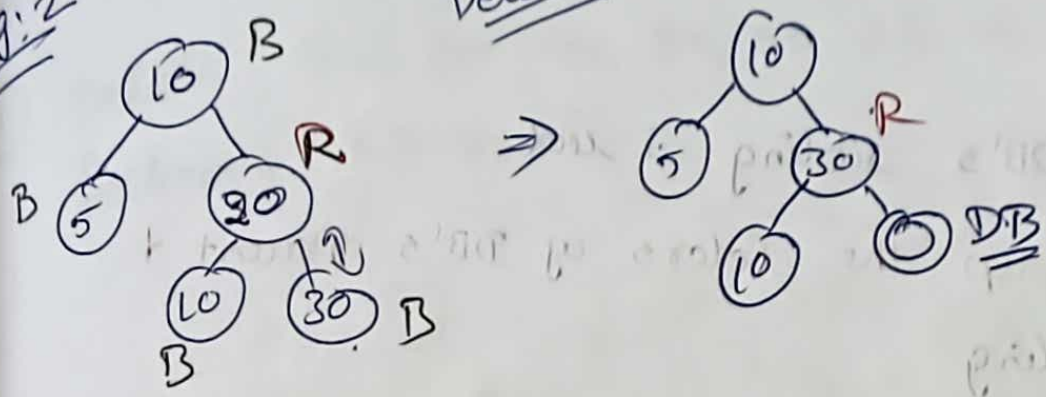
Delete 30



Smallest element

eg: 2

Delete 20



Deletion in Red Black tree:

Deletion is same as BST deletion.

Case 1:
⇒ If a node to be deleted is red, just delete it

Case 2:
⇒ If a node is Double black (DB) just remove DB

Case 3:
⇒ If a DB's sibling is black & both of its children are black.

* Remove DB
* Add black to parent (P)
  - If P is red it becomes black
  - If P is black, it becomes DB
* Make sibling red.

⇒ If still DB exits, apply other cases.

Case 4:

⇒ If DB's sibling is red

* Swap the colors of DB's parent & its sibling
* Rotate parent of DB in DB's direction

⇒ Reapply cases.

Case 5:

⇒ If DB's sibling is black, sibling's child who is far from DB is black but near child to DB is red.

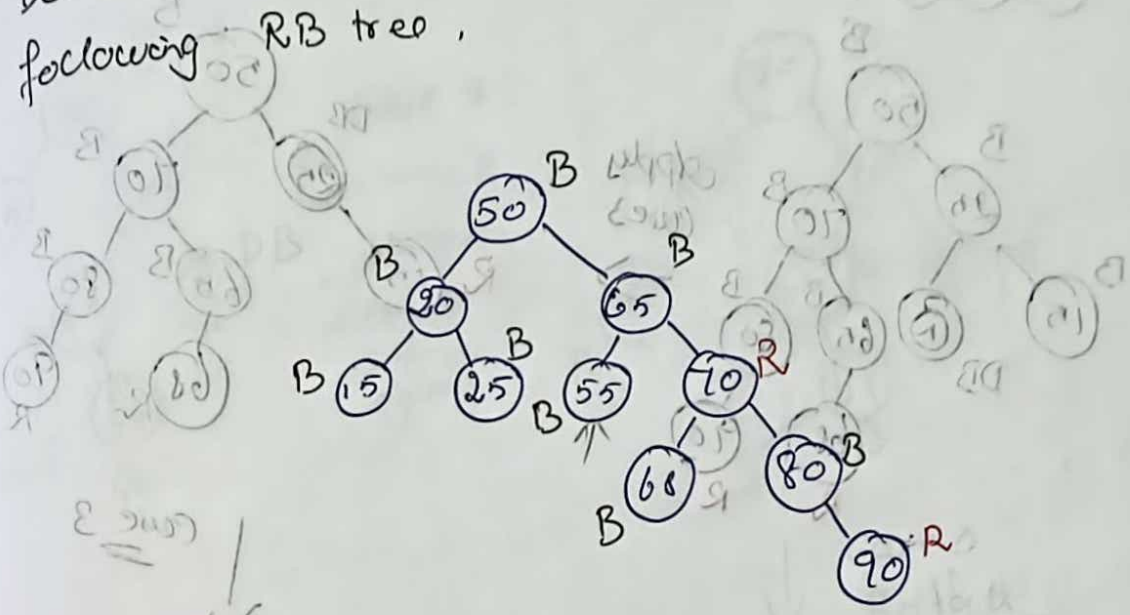
* Swap the color of DB's sibling & Siblings child who is near to DB
* Rotate sibling in opposite direction to DB
* Apply case 6.

Case 6:

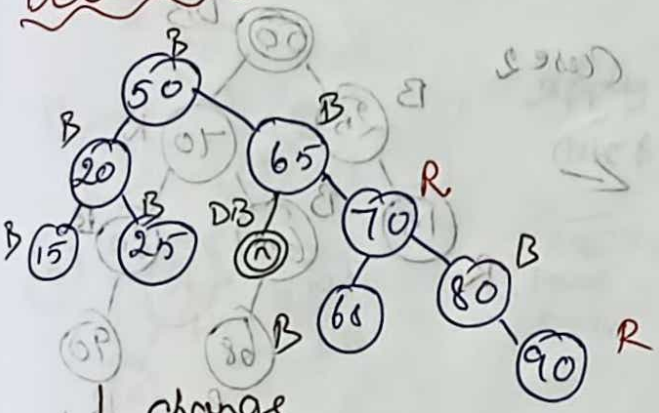
⇒ If DB's sibling is black and far child is red

* Swap the color of parent & sibling
* Rotate parent in DB's direction
* Remove DB
* Change color of red child to black.

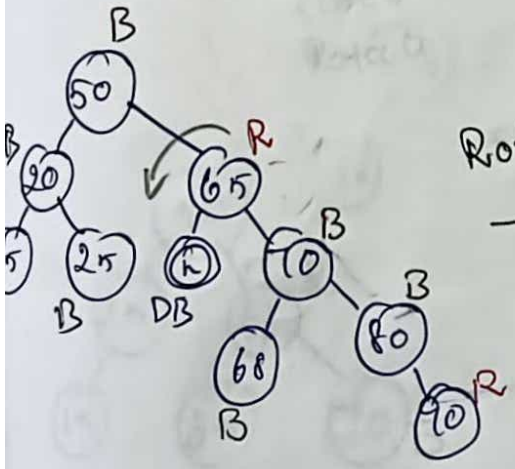
1) Delete 55, 20, 90, 80, 50, 25, 15 from the following RB tree.



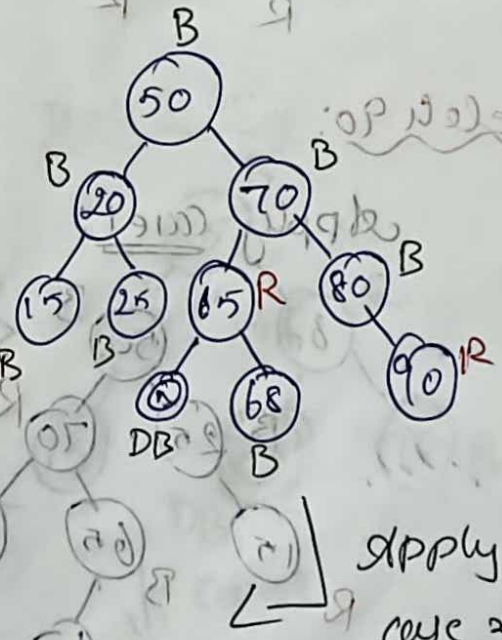
Delete 55:



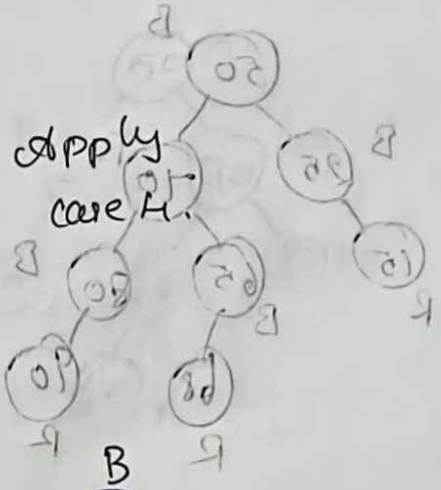
change color.



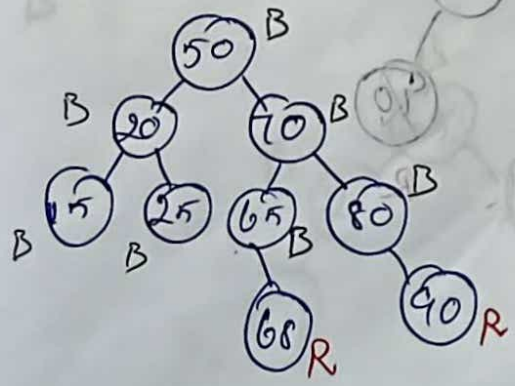
Rotate



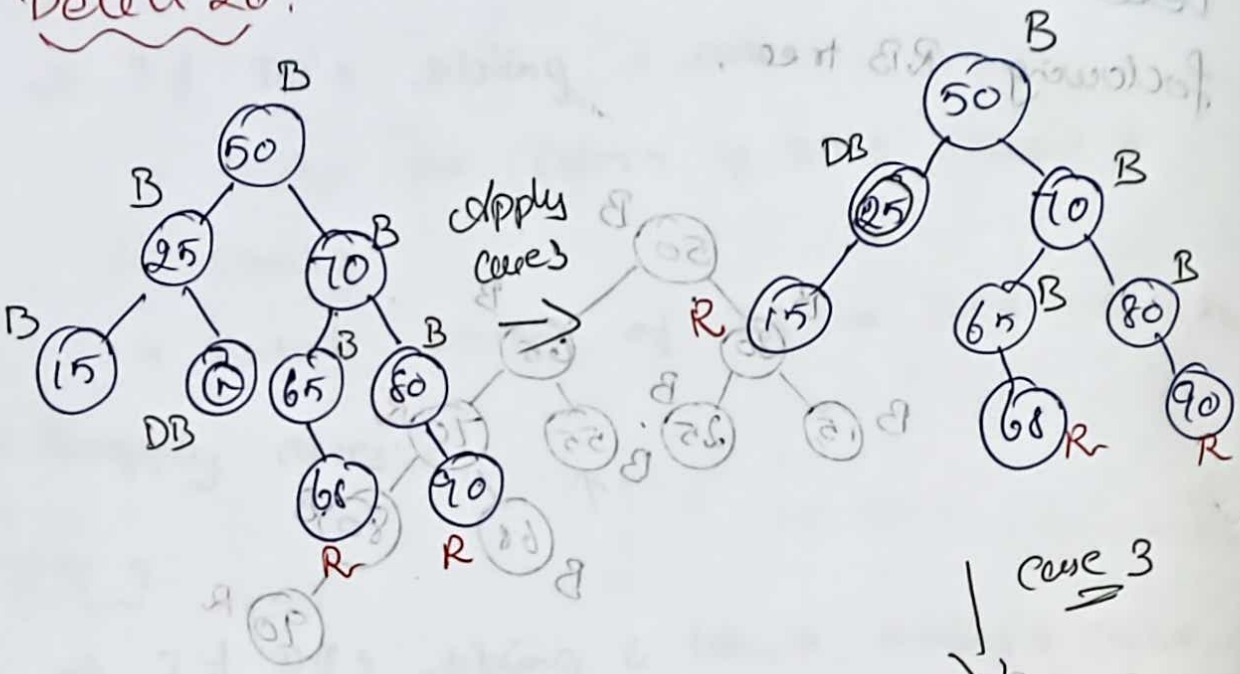
Apply case 4.

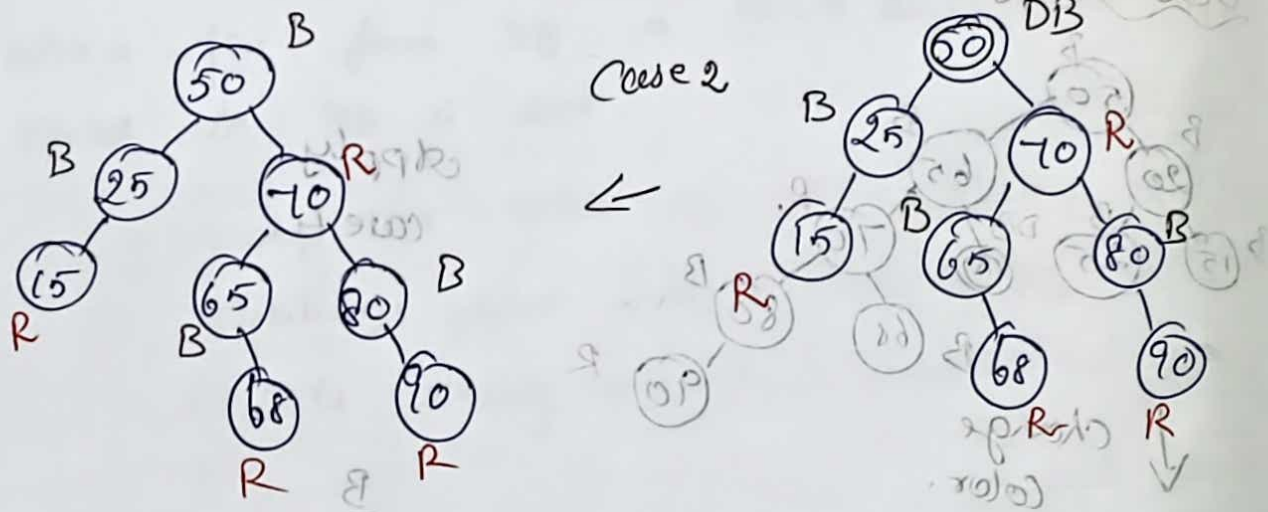


Apply case 3

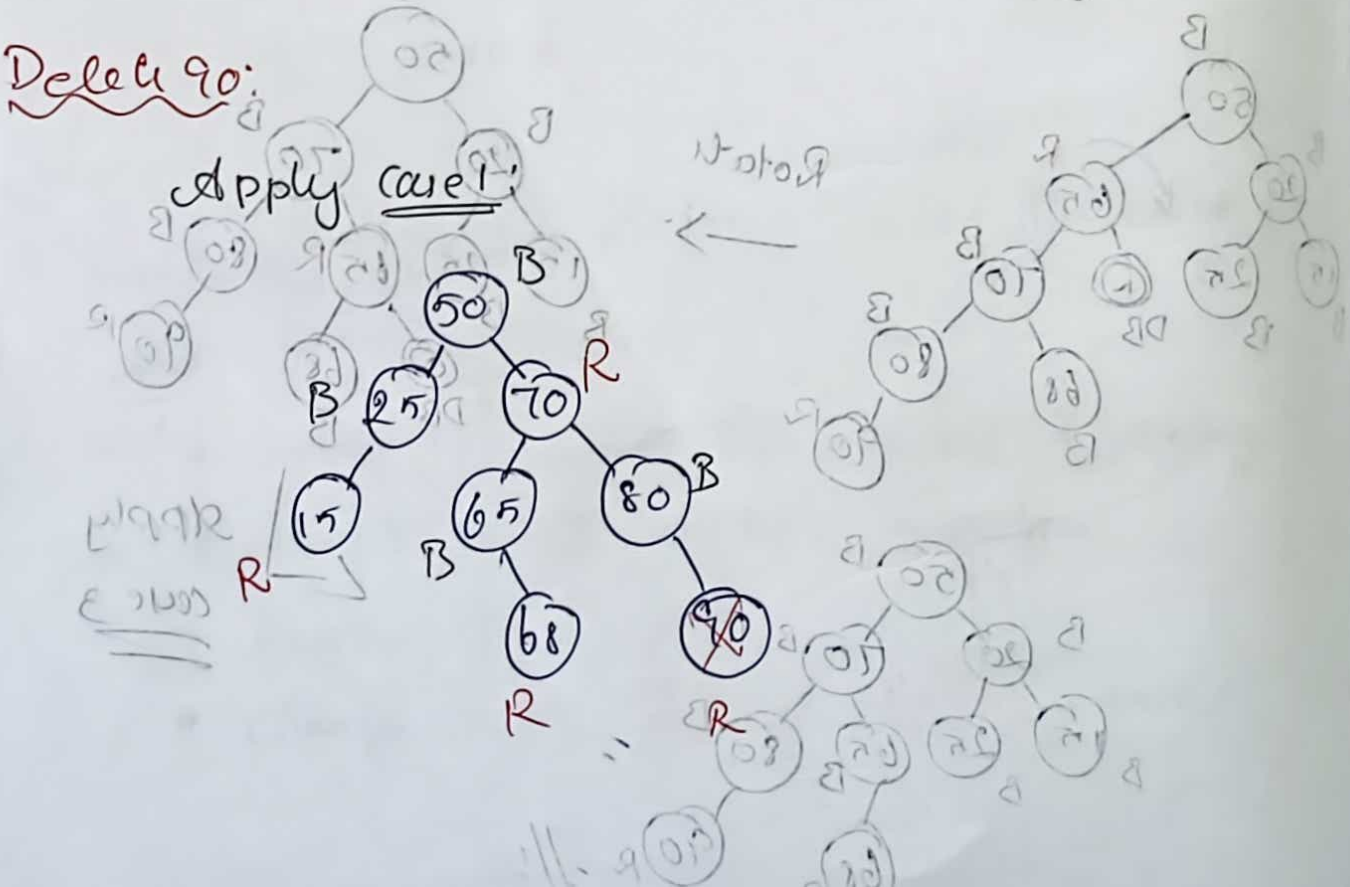


Delete 20:

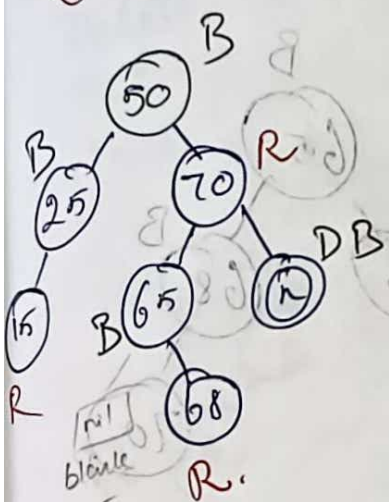




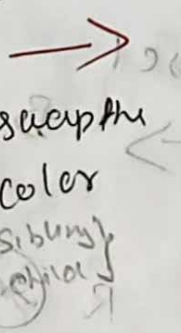
Delete 90:

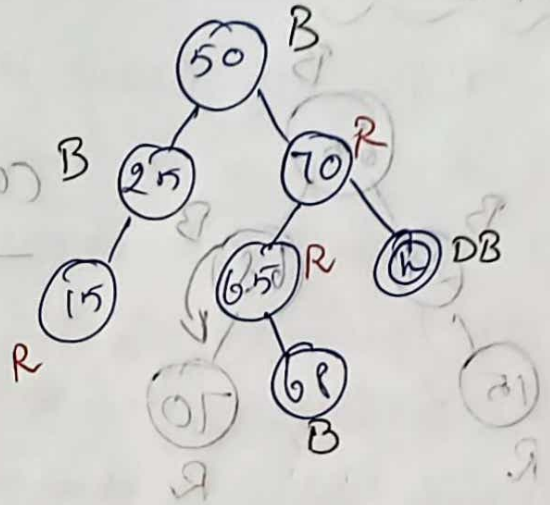


Delete 80:

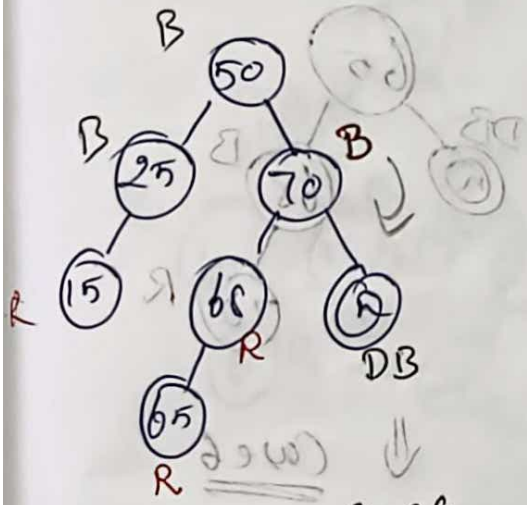


Case 5:

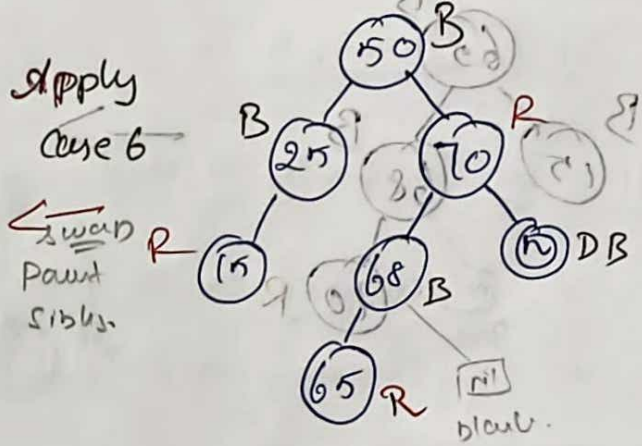




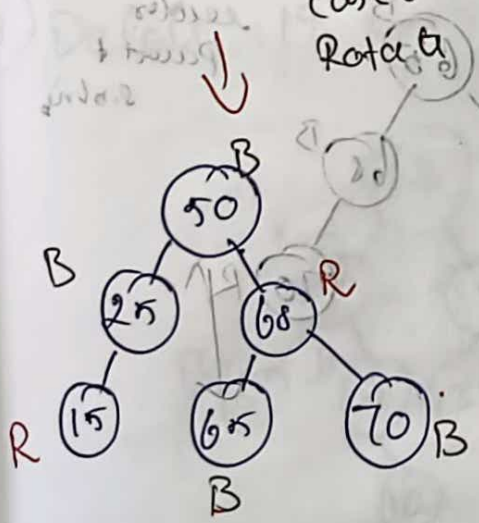
Case 5:
Rotate



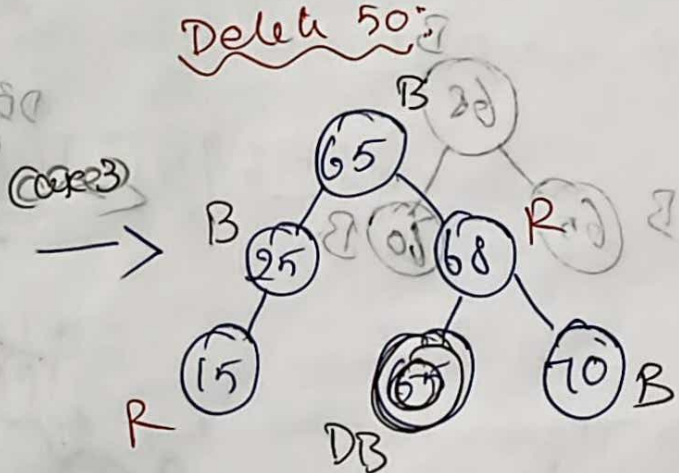
Apply Case 6



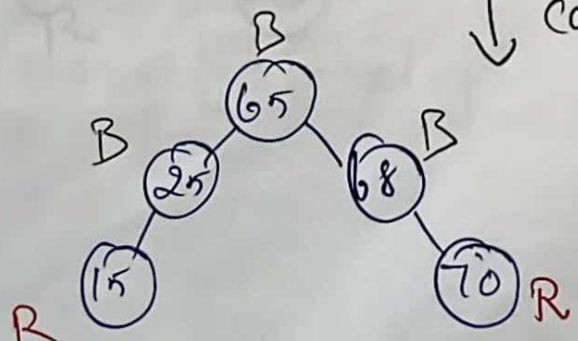
Case 6
Rotate



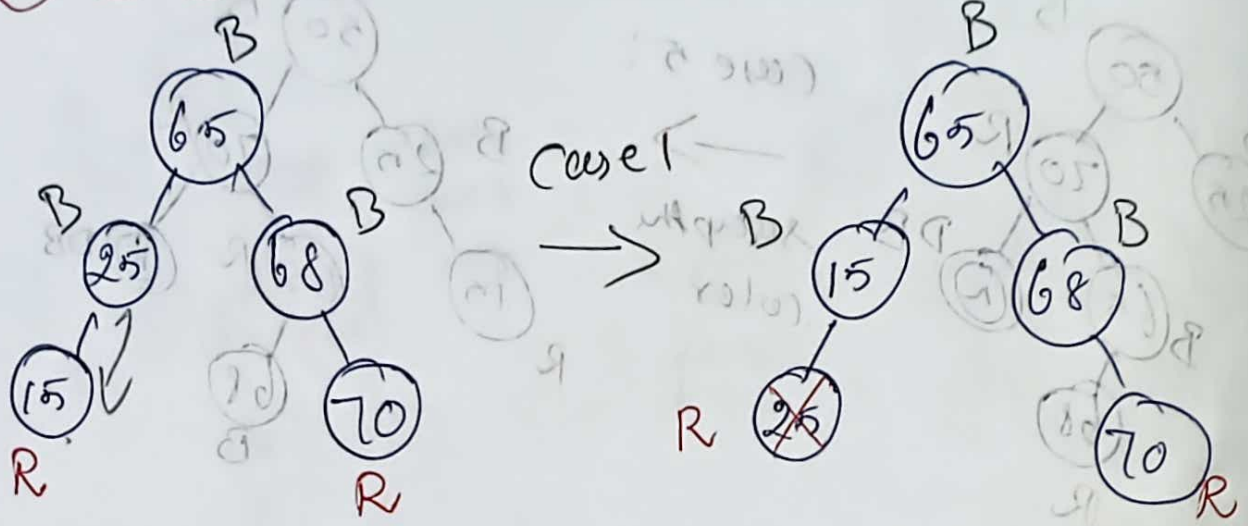
Delete 50:



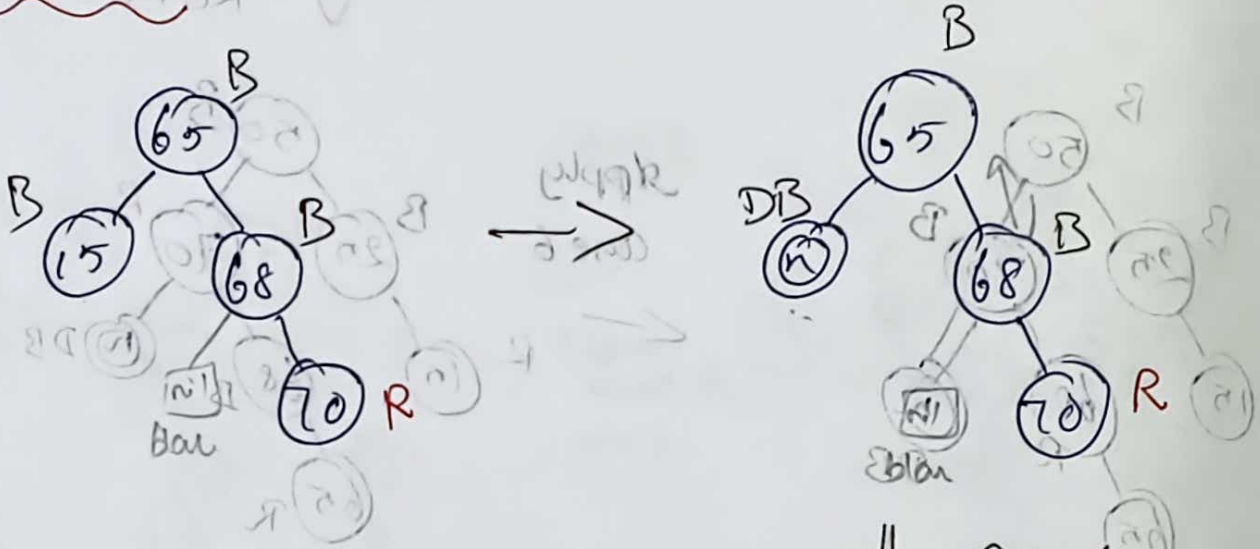
Case 3

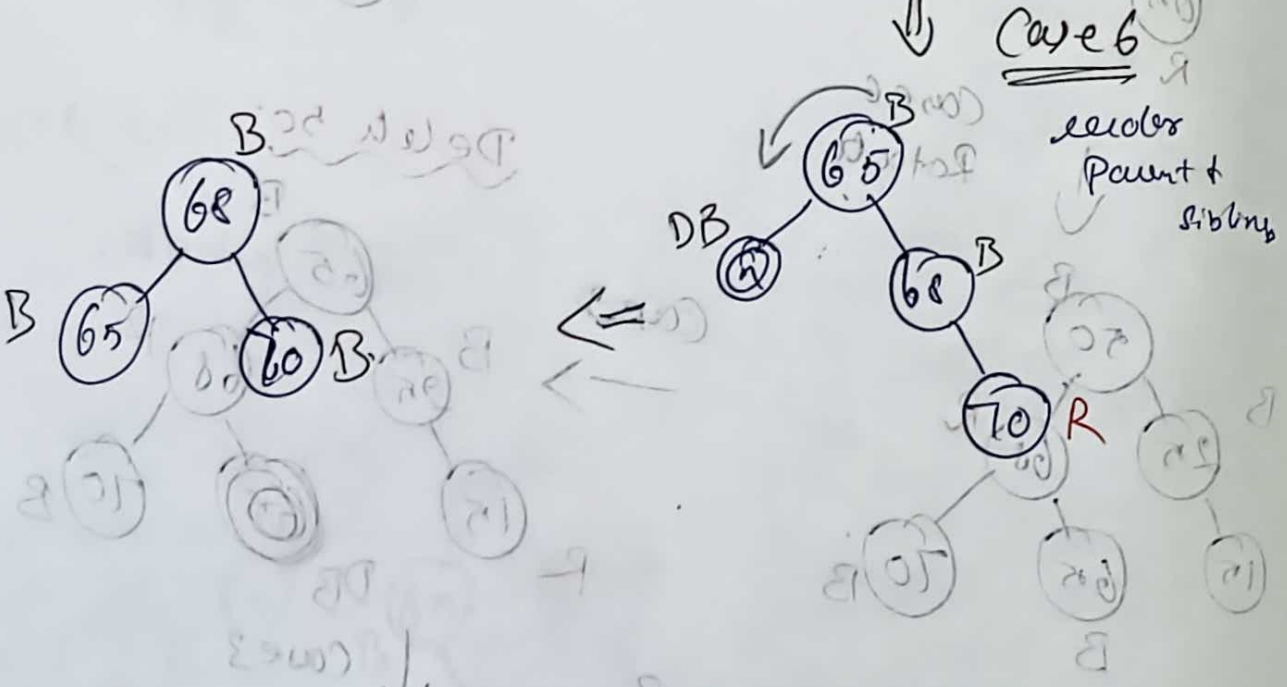


Delete 25:

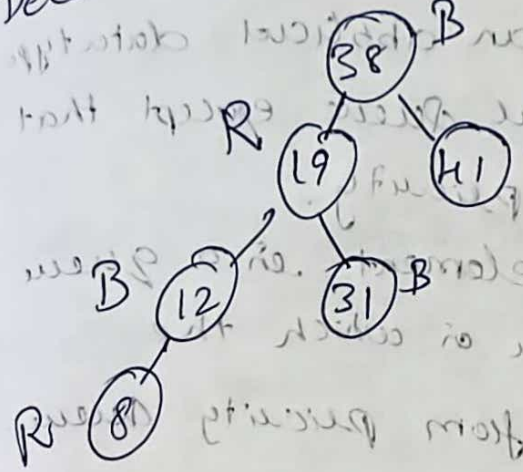


Delete 15:

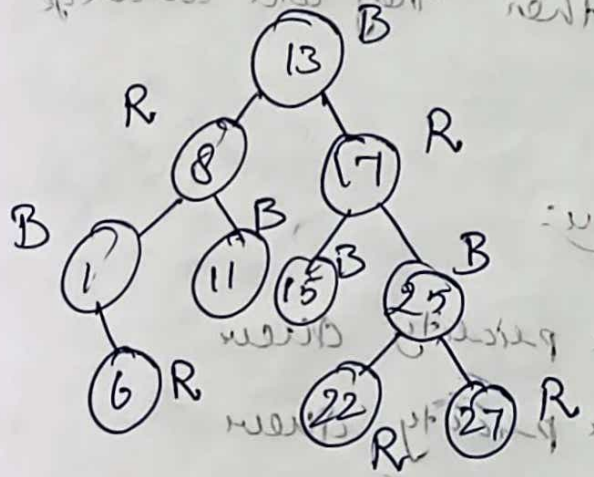




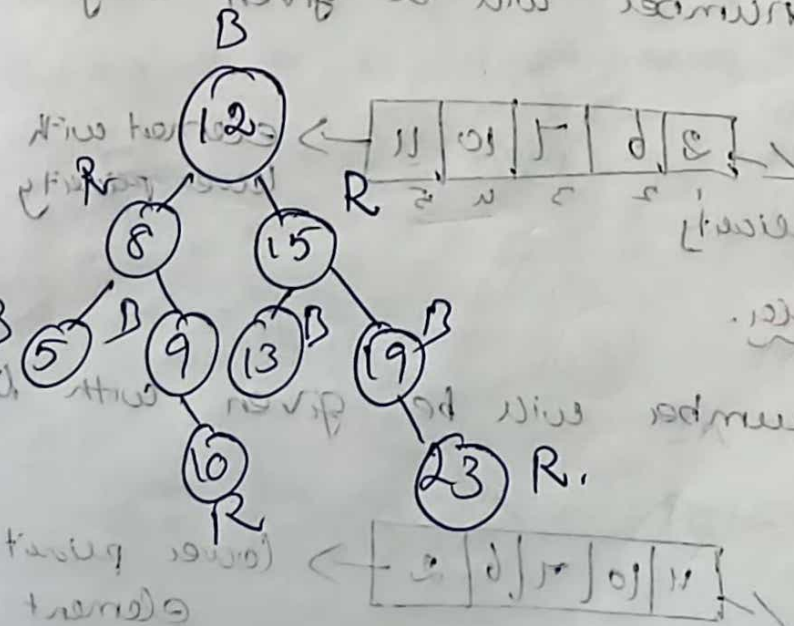
1) Delete 8, 12, 19, 31, 38, 41



2) Delete 6, 1, 11, 25, 8



3) Delete 19, 8, 15, 12,



Priority Queue.

=> A priority queue is an abstract data type that is similar to normal queue except that each element has some priority.

=> The priority of the element in a queue will determine the order in which the elements are removed from priority queue.

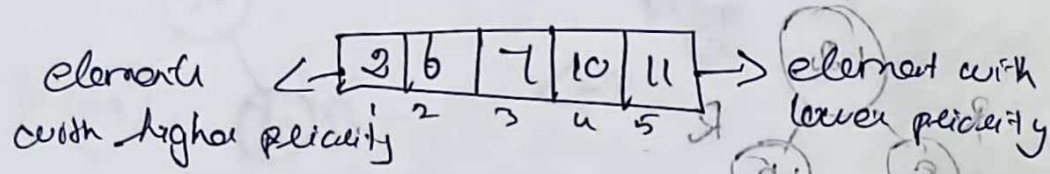
=> If two elements in a priority queue have the same priority, then they will arrange in FIFO principle.

Types of priority queue:

- Ascending order priority queue
- Descending order priority queue

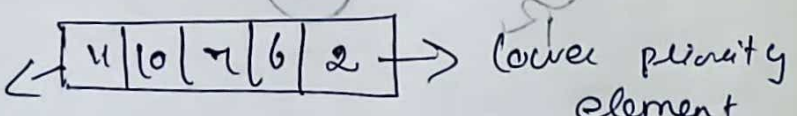
Ascending order.

=> Lower number will be given higher priority



Descending order.

=> Higher number will be given with higher priority



Representation of Priority Queue:

The priority queue can be represented with array, linked list, heap & binary search tree.

Heap:

Heap is a special case of balanced binary tree data structure where the root node key is compared with its children & arrange accordingly.

Binary Heap:

A binary heap is a binary tree with 2 properties * It is a complete tree (all levels are completely filled except the last level & in last level all keys are as left as possible).

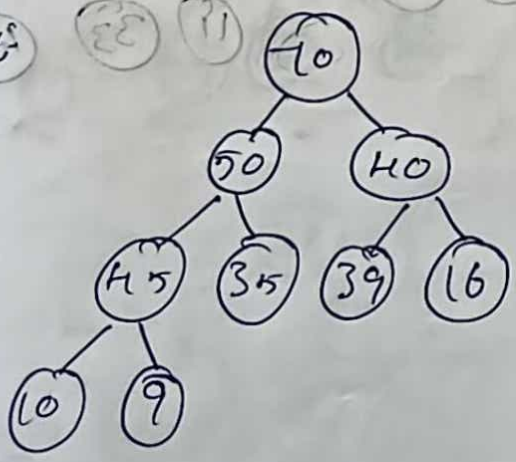
* Binary heap may be either min heap (or) max heap

Max heap:

For every node i, the value of node is less than (or) equal to its parent node.

A[Parent(i)] ≥ A[i]

Examples:



In all nodes Parent is ≥ child

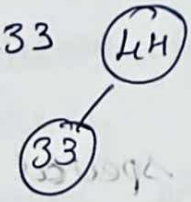
Application: heapsort, OS like priority scheduling, load balancing, interrupt handling.

Insertion in max heap: (top down approach)

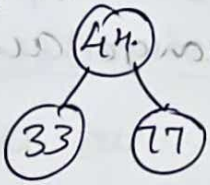
44, 33, 77, 11, 55, 88, 66

step 1: 44

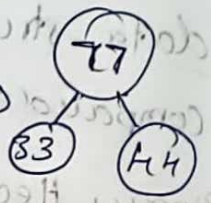
step 2: 33



step 2: 77

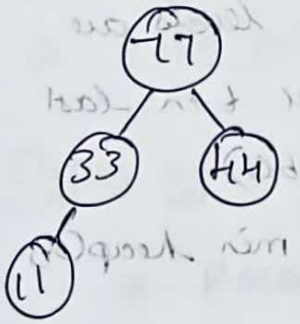


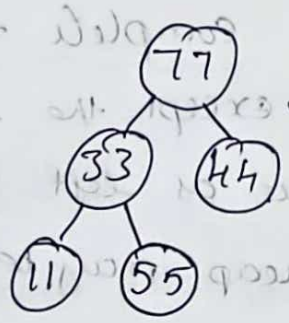
heapify. Because: parent node should be greater



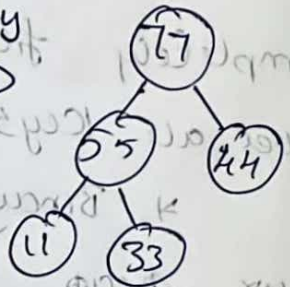
step 4: 11

step 5: 55

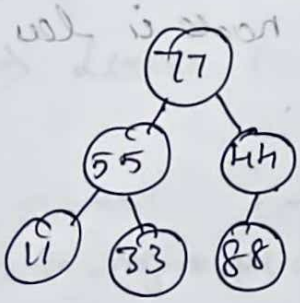




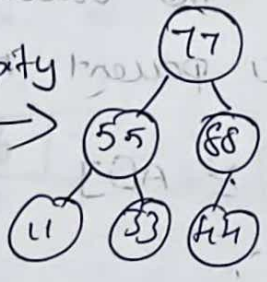
heapify



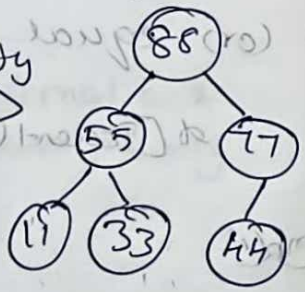
step 6: 88



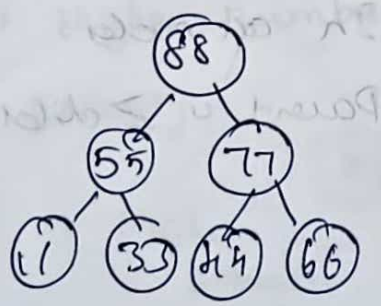
heapify



heapify



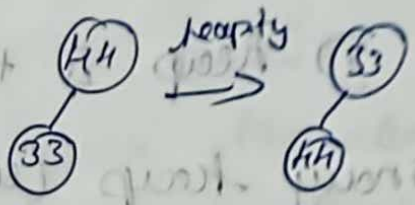
step 7: 66



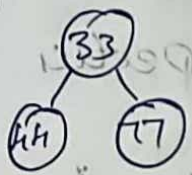
Insert in min heap:

Step 1: 44

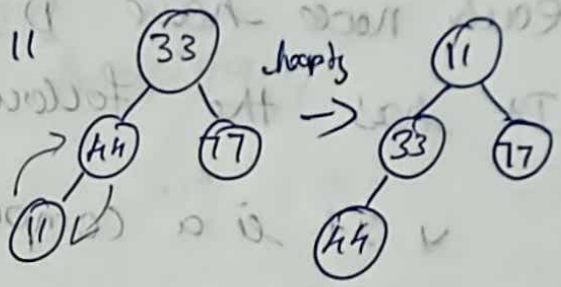
Step 2: 33



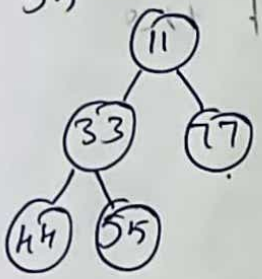
Step 3: 77



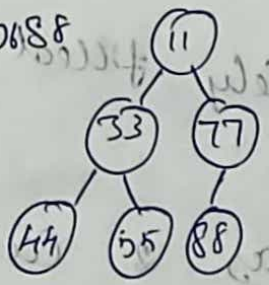
Step 4: 11



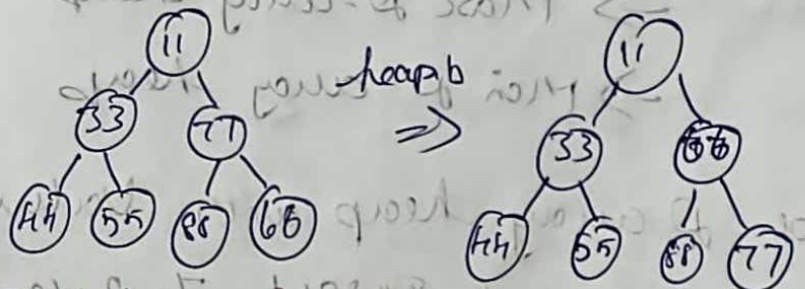
Step 5: 55



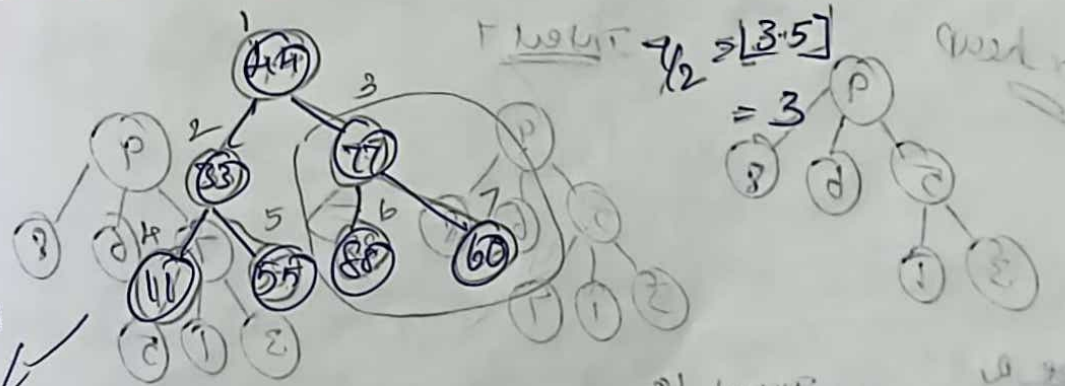
Step 6: 88



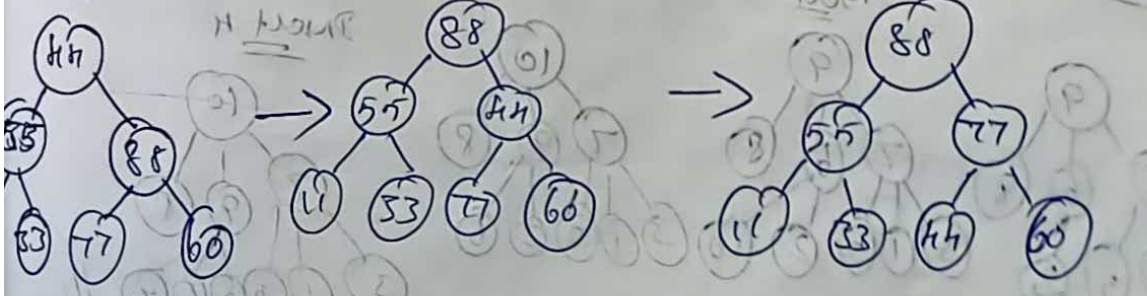
Step 7: 66



Bottom up approach:



heapify



D-heap

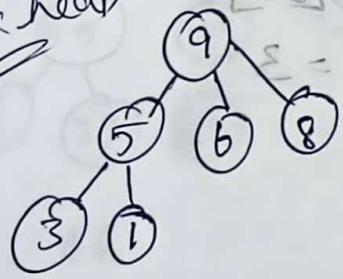
D-heap is the generalization of binary heap (where D = 2) in which each node have D children instead of 2.

It has the following properties

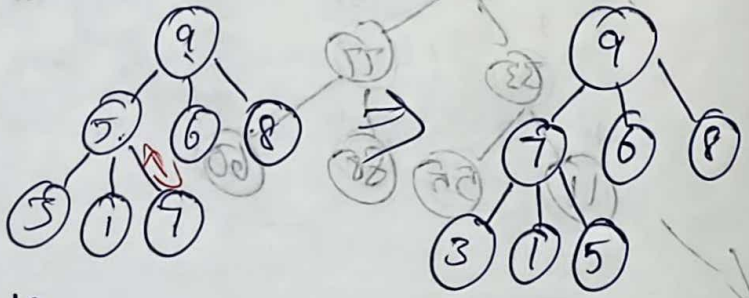
- It is a complete binary tree with all levels completely filled except the last level
- It has 2 categories
  - ⇒ Max D-ary heap
  - ⇒ Min D-ary heap

Build a D ary heap with values.
9, 5, 6, 8, 3, 1. & Insert 7, 2, 10, 4 for D = 3

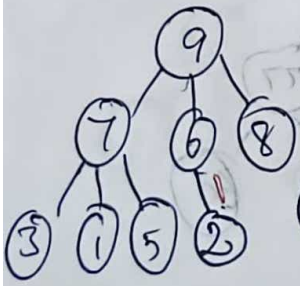
Max heap



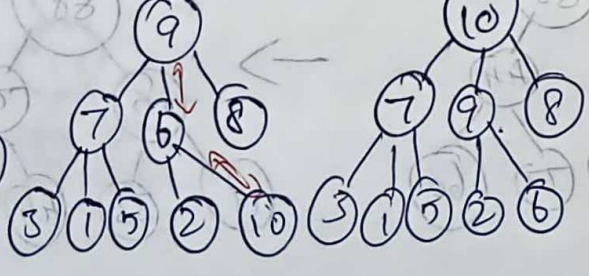
Insert 7



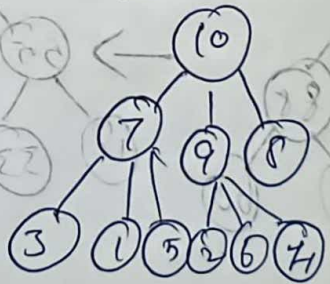
Insert 2



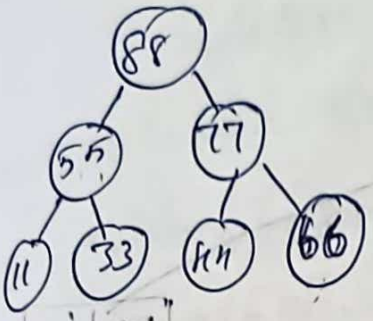
Insert 10



Insert 4

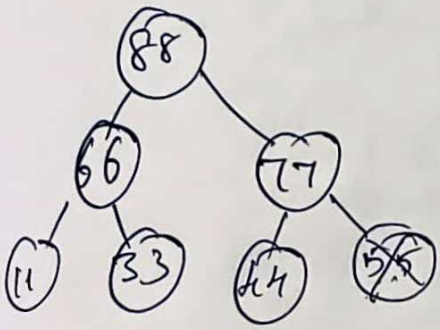


Delete 55, 33 from max heap tree.



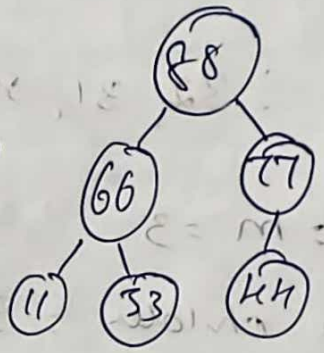
* Replace the deleted element with last element
* Delete the last element
* heapify.

Delete 55

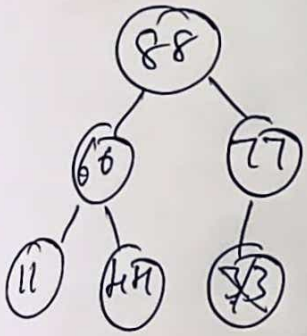


Delete

heapify

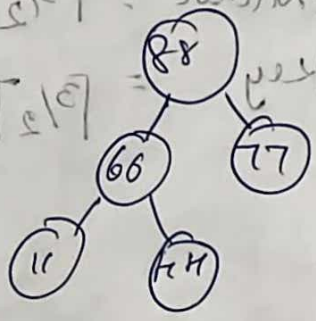


Delete 33



Delete

heapify

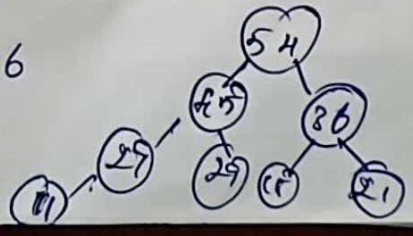


HW

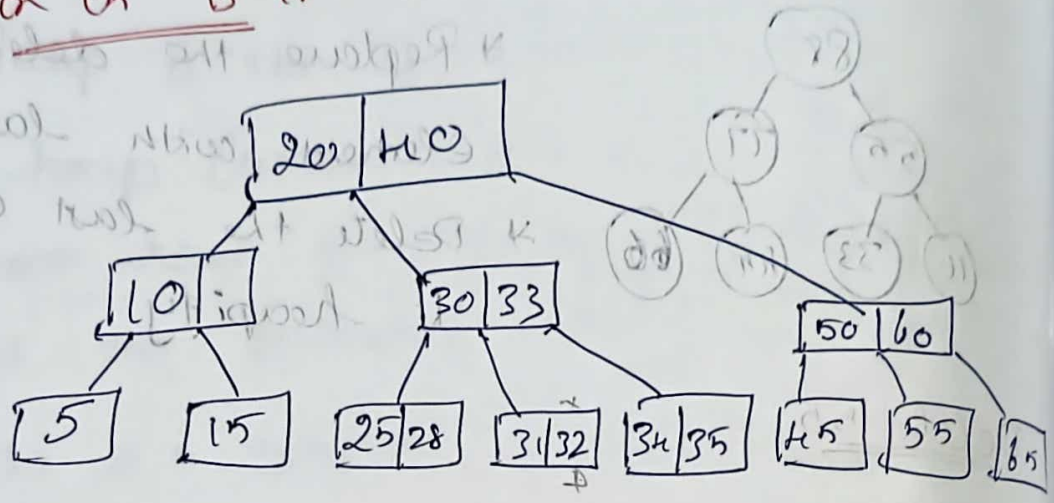
1. Build max heap from the following set of numbers 45, 36, 54, 27, 63, 72, 61, 18

2. Consider the following max heap

Delete node 54 & 36



Deletion in B Tree. order = 3



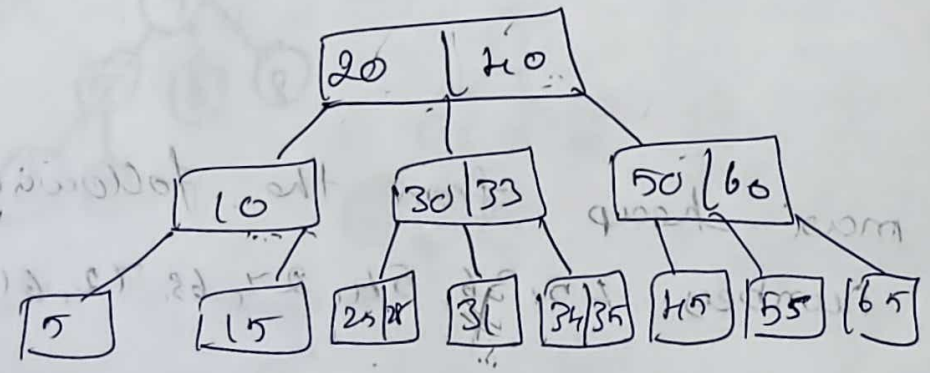
Delete : 32, 31, 50, 33

order = m = 3

1. max children = 3
   max key = 3-1 = 2

2. min children = $\lceil 3/2 \rceil = 2$ ✓
   min key = $\lceil 3/2 \rceil - 1 = 2 - 1$

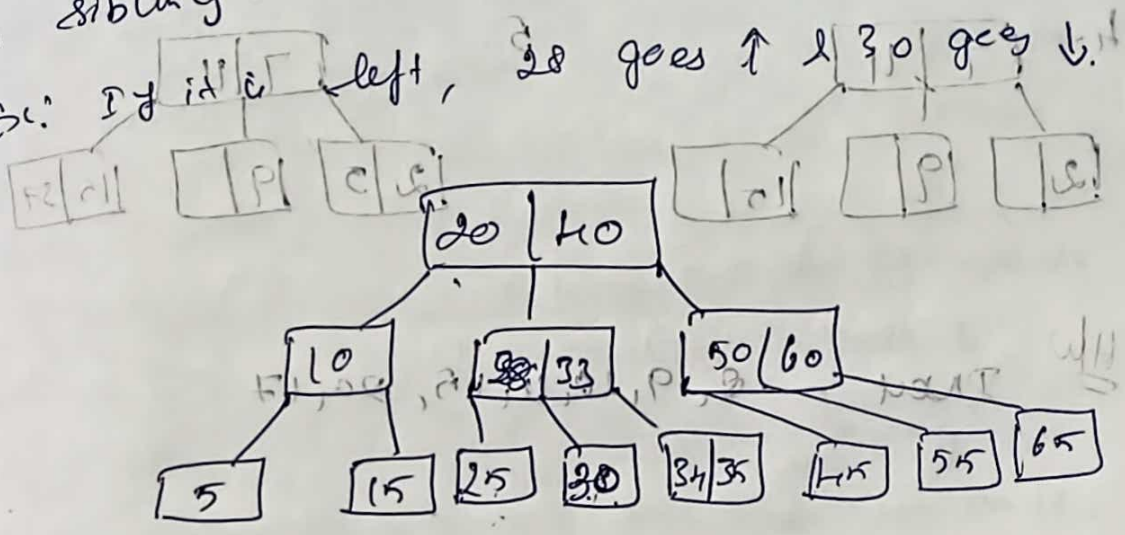
Delete 32:



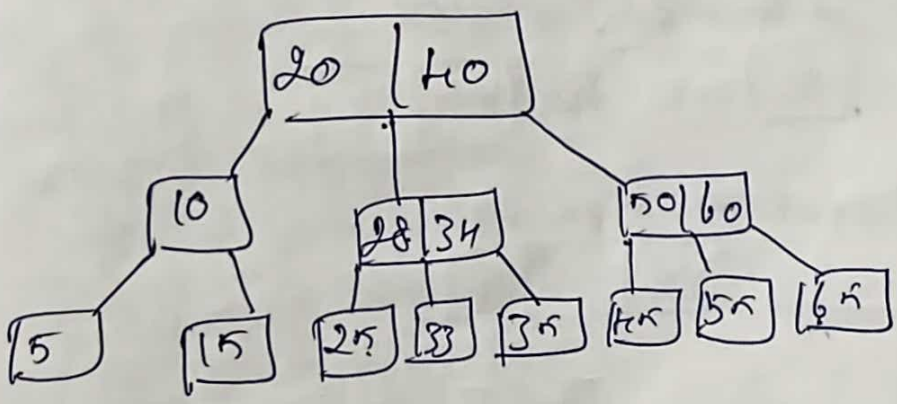
Delete 31:

To delete 31, the node should have atleast one key element. So it has to borrow from either left sibling (or) right sibling.

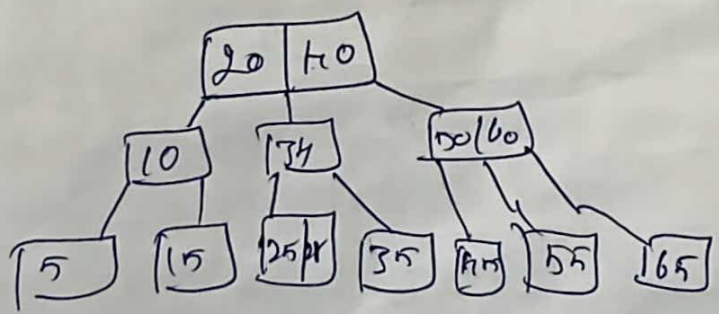
Ex: If it is left, 28 goes ↑ & 30 goes ↓.



Delete 30:



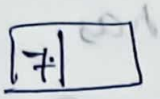
Delete 33:



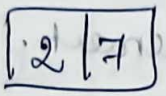
B tree order = 3

Creation: 7, 2, 9, 11, 15, 5, & 37.

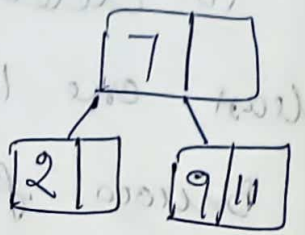
1.



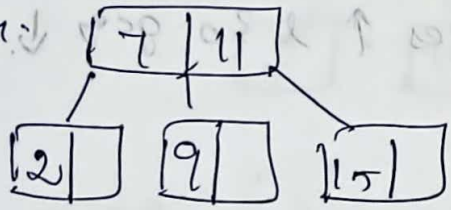
2.



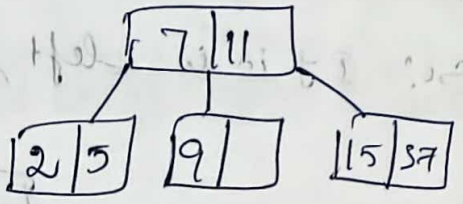
3.



4.



5.



HW Insert - 8, 9, 10, 11, 15, 20, 17

Unit - V

Sorting & Searching

⇒ Sorting & Searching are the two of the most common operation performed by computer all around the world.

⇒ The sorting operation arranges the numerical & alphabetical data present in a list in a specific order or sequence.

⇒ Searching, on the other hand, locates specific element across a given list of elements.

⇒ At times, a list may require sorting before the search operation can be performed on it.

eg: A telephone directory (sorting & searching)

⇒ There are number of sorting techniques. & selection of specific technique depends on

* Size of the data structure
* algorithm efficiency
* programmer's knowledge of the technique.

Different techniques:

* Bubble sort
* Shell Sort
* Bucket Sort
* Selection Sort
* Quick sort
* Merge sort
* Insertion sort.

Internal sorting: All sorting techniques which require the data set to be present in the main memory.

External Sorting: the data set cannot be adjusted in main memory at once, it needs to be stored in a hard disk, floppy disk (or) any other storage device.

Bubble Sort: (n-1) pass.

It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element & replace them accordingly

Algorithm:

Begin BubbleSort(arr)
for all array element
if (arr[i] > arr[i+1])
swap(arr[i], arr[i+1])
end if
end for.
return arr
end BubbleSort

Working:

Elements of array arr.

| 13 | 32 | 26 | 35 | 10 |

First Pass:

=> | 13 | 32 | 26 | 35 | 10 | * 32 > 13, sorted

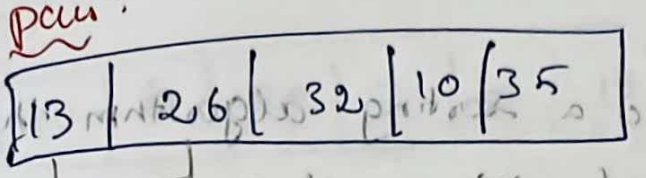
=> | 13 | 32 | 26 | 35 | 10 | → 26 is small, swapping

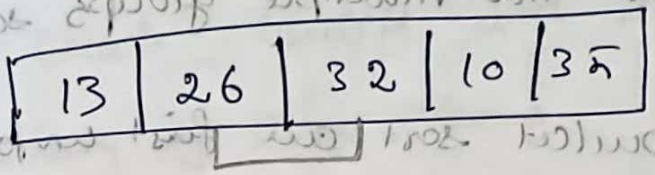
=> | 13 | 26 | 32 | 35 | 10 | → 35 > 32, already sorted

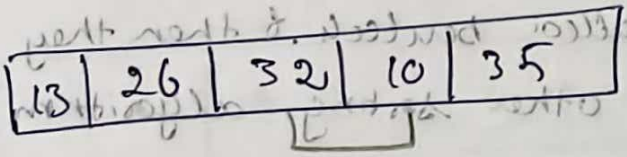
=> | 13 | 26 | 32 | 35 | 10 | → 10 is small, swapping

=> | 13 | 26 | 32 | 10 | 35 |

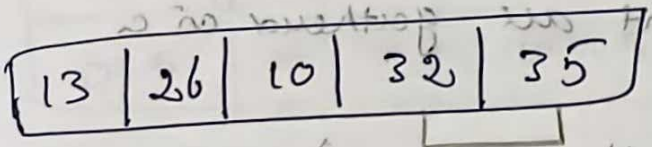
Second Pass:

=> 

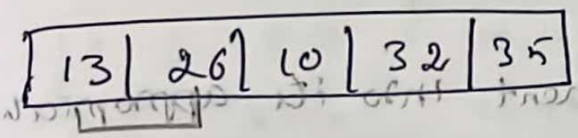
=> 

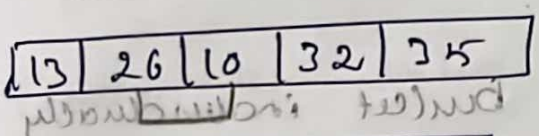
=> 

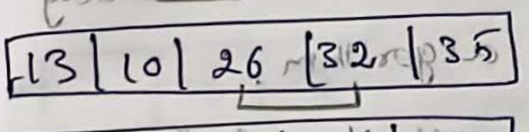
* 10 is smaller, swapping.

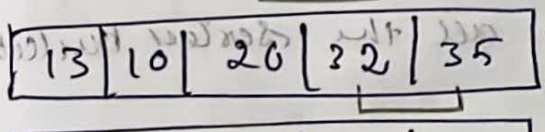
=> 

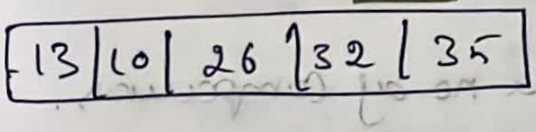
Third Pass:

=> 

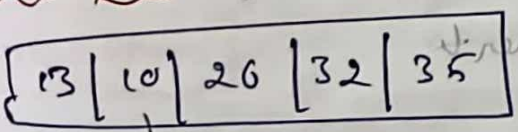
=> 

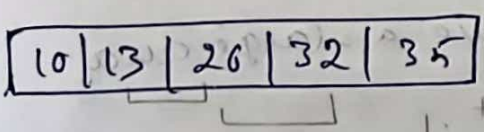
=> 

=> 

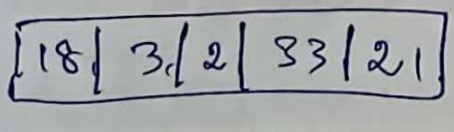
=> 

Fourth Pass:

=> 

=> 

Finally array is completely sorted.

HW => 

Bucket Sort:

⇒ Bucket sort is a sorting algorithm that separate the elements into multiple groups said to be buckets.

⇒ Elements in bucket sort are first uniformly distributed into groups called buckets & then they are sorted by any other sorting algorithm.

⇒ After that, element are gathered in a sorted manner.

Procedure: (Scatter-gather approach)

1. First, partition the range into a fixed number of buckets
2. Then, toss every element into its appropriate bucket
3. After that, sort each bucket individually by applying a sorting algorithm
4. And at last, concatenate all the sorted buckets.

Advantages:

* Bucket sort reduces the no. of comparison.
* It is asymptotically fast because of the uniform distribution of elements.

Disadvantages:

* It is not useful if we have a large array. Because it increases the cost
* Needs extra space to sort the buckets.

Example:

10	8	20	7	16	18	12	1	23	11

⇒ Create bucket with range 0 to 25

① Scattering of array elements.

1	8, 7	10, 12, 11	20, 16, 18	23
0-5	5-10	10-15	15-20	20-25

② Sorting using any sorting algorithm.

1	7, 8	10, 11, 12	16, 18, 20	23
0-5	5-10	10-15	15-20	20-25

③ Gather the sorted elements from each bucket in order

1	7	8	10	11	12	16	18	20	23

∴ Now the array is completely sorted.

HW 2, 27 13 18 21 43 42 39 31 4.

Algorithm

```
Bucket Sort (A[])
Let B[0... n-1] be a new array
n = length[A]
for i=0 to n-1
make B[i] an empty list
for i=0 to n
do insert A[i] into list B[n A[i]]
for i=0 to n-1
do sort list B[i] with insertion sort
concatenate list B[0] ... B[n-1]
```

Selection sort:

(18) 3 2 33 21 smallest

2 (3) (18) 33 21

2 3 18 21 33

Insertion sort:

18 3 2 33 21

3 18 2 33 21

2 3 18 33 21

2 3 18 33 21

2 3 18 21 33 //

Shell Sort

* Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

* Insertion sort, at a time elements can be moved ahead by one position only.

* To move an element to a far-away position, many movements are required. Shell sort overcomes this drawback. It allows movement & swapping of far away element as well.

Interval
$h = h*3+1$.
where h is interval with initial value 1

Algorithm:

→ given array
→ size of array

```
shellsort (a,n)
for (interval = n/2 ; interval > 0 ; interval /=2)
  for (i = interval ; i<n ; i++ = 1)
  temp = a[i];
  for (j=i ; j>=interval && a[j-interval] > temp ; j-=interval)
  a[j] = a[j-interval];
  a[j] = temp;
End shellsort
```

Working:

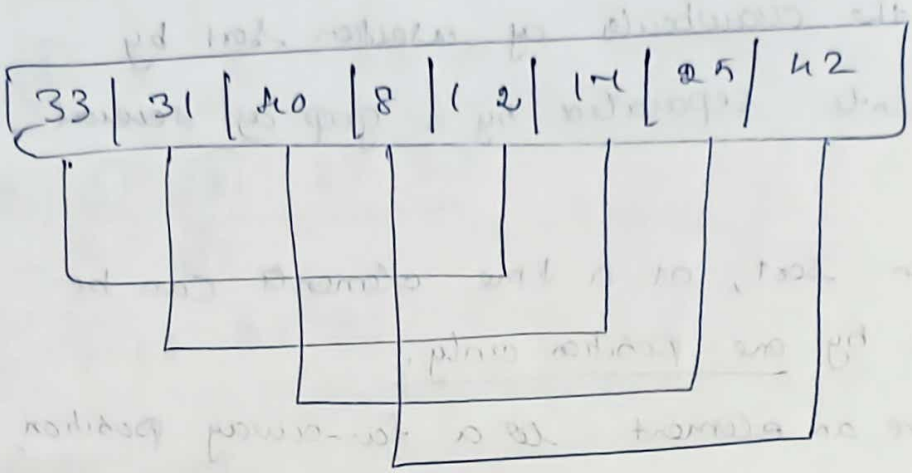
33	31	40	8	12	17	25	42

Intervals = $n/2, n/4, n/1$

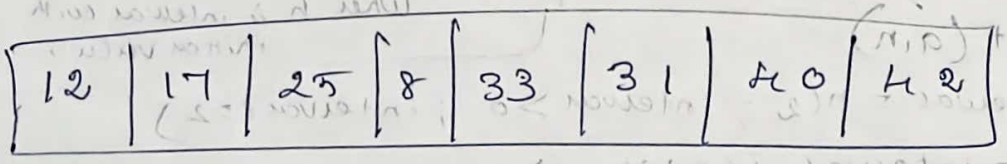
First loop: $n=8$ $(n/2 = 8/2 = 4)$

⇒ The elements lying at the interval of 4 ($n/2 = 4$)
⇒ Elements will be compared & swapped if they are not in order.

⇒ At the interval 4, the sublists are.

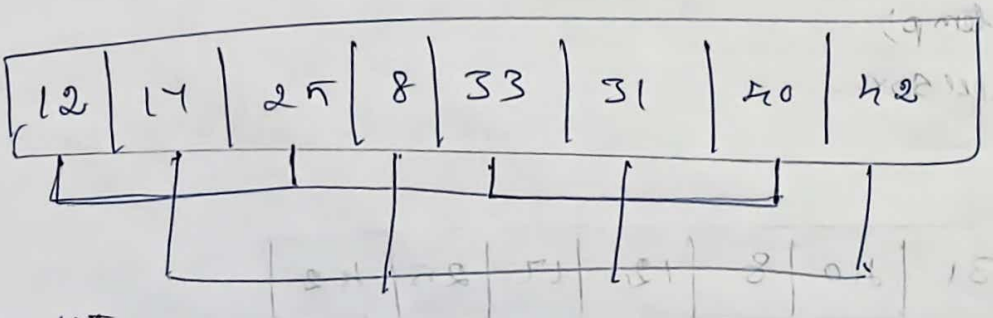


⇒ After comparing & swapping, updated arr.

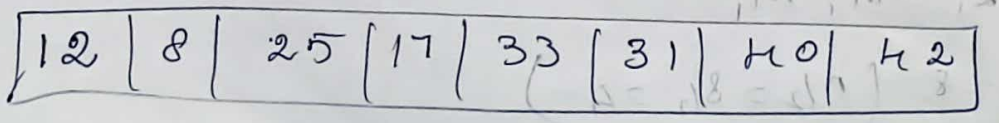


Second loop:-

⇒ Elements are lying at the interval h = 2.



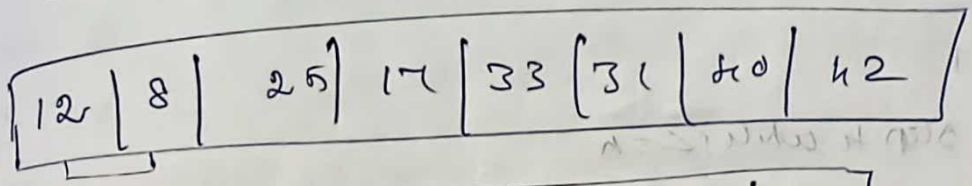
⇒ After comparing & swapping

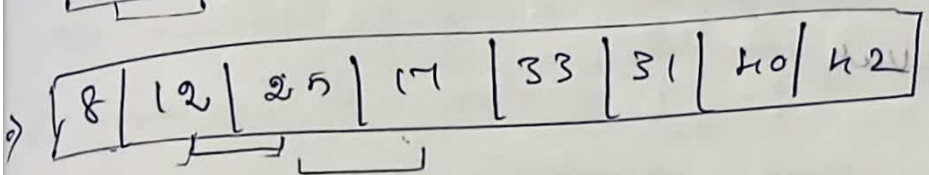


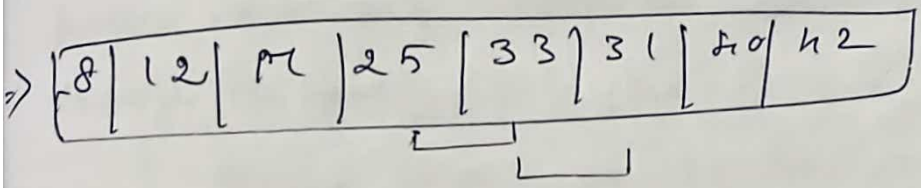
third loop:-

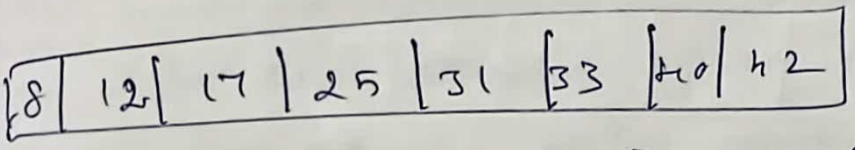
=> Elements lying in interval 1 to 1 => 1.

=> uses insertion sort to sort the array element.









∴ the array is sorted in ascending order.

Searching
- Linear search
- Binary search.

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful & the process returns the location of that element otherwise, the search is called unsuccessful.

Linear Search algorithm:

⇒ Also called as sequential search algorithm.

Algorithm:

1: Set pos = -1.
2: Set i = 1
3: Repeat step 4 while i <= n
4: if a[i] == val
   Set pos = i
   print pos
   go to step 6
   [end of if]
   Set ii = i+1
   [end of loop]
5: if pos = -1
   print "value is not present in the array"
   [end of if]
6: exit

Working:

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Let the element to be searched is (i <= 41)

⇒ Now start from first element & compare with each element of the array.

⇒ When the element is found, it returns the index.

Binary search algorithm:

⇒ It follows divide & conquer approach.

⇒ the list is divided into two halves & the item is compared with the middle element of the list.

⇒ If the match is found then, the location of the middle element is returned.

⇒ otherwise we search into either of the halves depending upon the result produced through the match.

Note: Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

2 methods:

* Iterative method
* Recursive method (divide & conquer approach).

Elements of array:

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

⇒

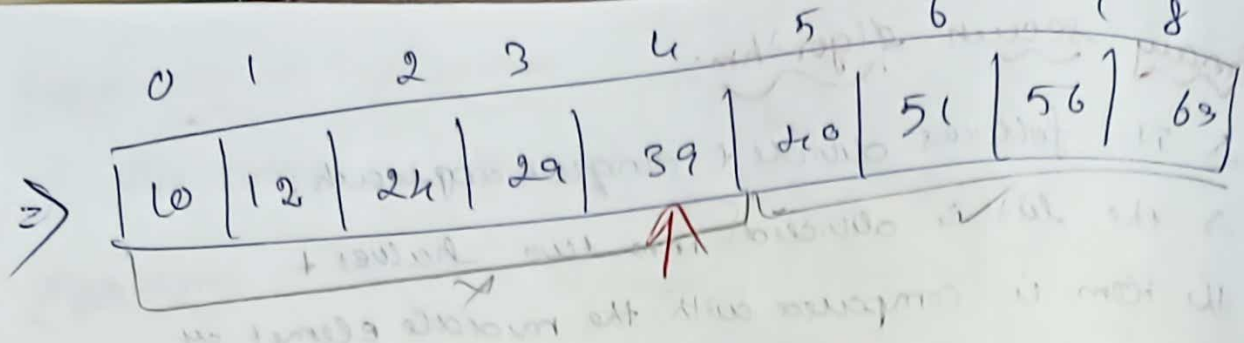
K = 56

Mid of array = (beg + end)/2
= (0 + 8)/2
= 4

↳ Mid = (beg + end)/2

↳ key < mid element
end = mid - 1

↳ key > Mid element
beg = mid + 1



(1) ⇒ A[mid] = 39

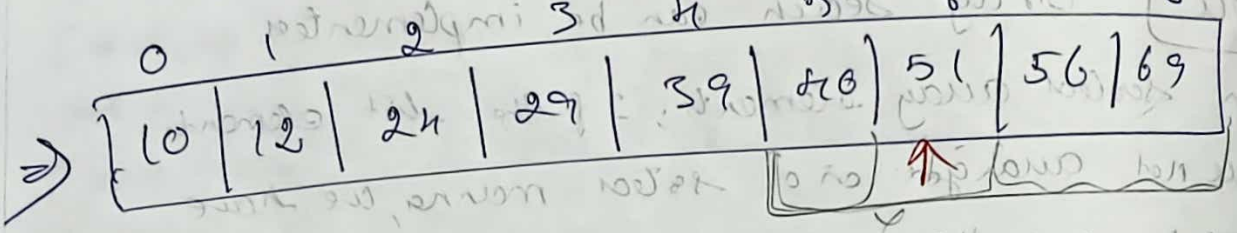
A[mid] < K

39 < 56 ✓

⇒ So beg = mid + 1

= 5, end = 8

Now mid = (5 + 8)/2 = 13/2 = 6



A[mid] = 51 ✓

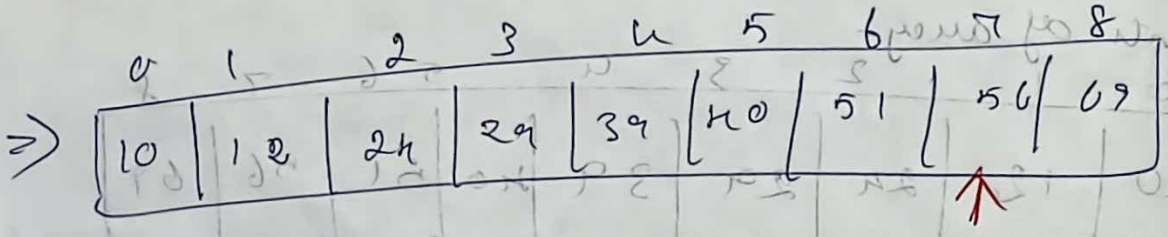
A[mid] < K

51 < 56

⇒ So beg = mid + 1

= 6 + 1

= (7 + 8)/2 = 15/2 = 7



A[mid] = 56

A[mid] = K (56 = 56)

So, location = mid

∴ Element found at 7th location of the array.

H/W

10	14	19	26	27	31	33	35	42	44

K = 31

1) mid = (0+9)/2 ⇒ 4

mid location = 27

27 < 31

2) ⇒ so, beg = mid+1 = 5

mid = (5+9)/2 ⇒ 14/2 ⇒ 7 ⇒ 5 + (9-5)/2 ⇒ 5 + 4/2 ⇒ 7

mid location = 35

35 > 31

3) beg = mid-1 = 6

mid = (6+9)/2 ⇒ 15/2 ⇒ 7

Linear search algorithm:

0	1	2	3	4	5	6
90	87	53	10	15	23	67

1st index

key value = 15

Binary search algorithm:

3	10	15	20	35	40	60
a[0]	1	2	3	4	5	6

key = 15.

⇒ mid = (beg + end)/2 = (0+6)/2 = 3 [20]

key < mid element: 15 < 20, end = mid-1 = 3-1 = 2.

key > mid element: beg = mid+1 = 3+1 = 4

⇒ mid = (4+6)/2 = 10/2 = 5

① mid = beg + end
= 2
= $\frac{0+6}{2}$ = 3

a[3] = 20 ||.

15 < 20 ⇒ end = mid − 1
= 3 − 1
end = 2

② mid = $\frac{0+2}{2}$ = 1

a[1] = 10

15 > 10 ⇒ beg = mid + 1
= 1 + 1
= 2

③ mid = $\frac{2+2}{2}$ = 2.

a[2] = 15

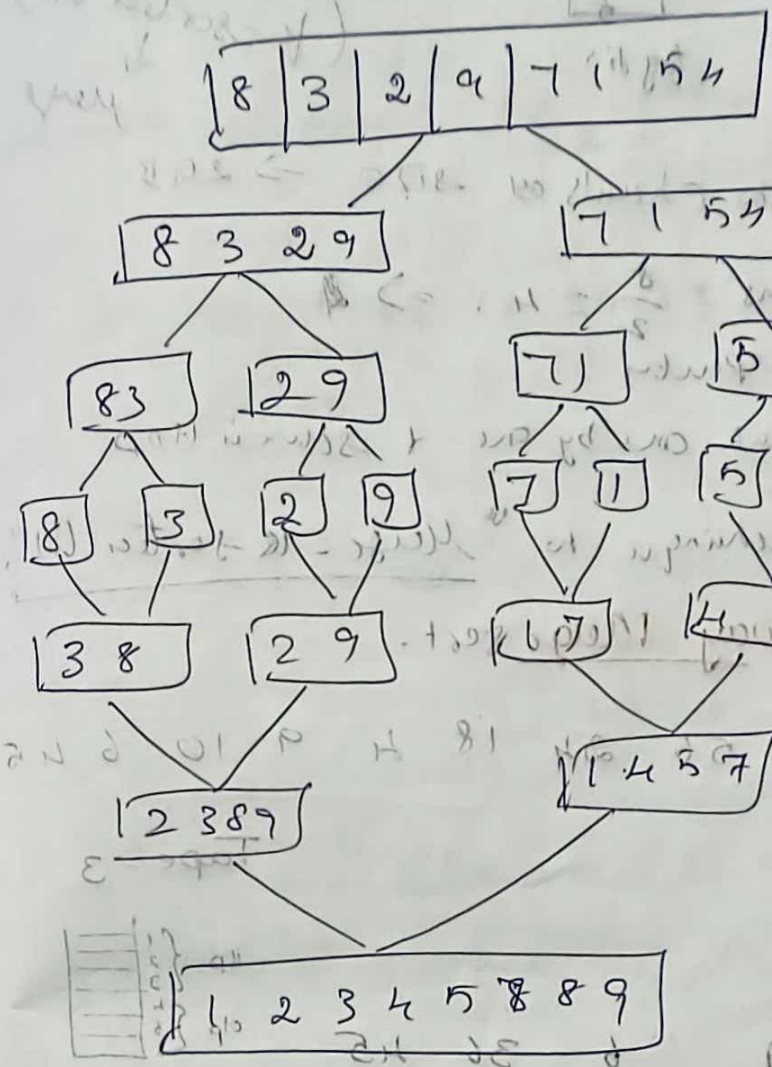
15 = 15

∴ Index value is 2 ||.

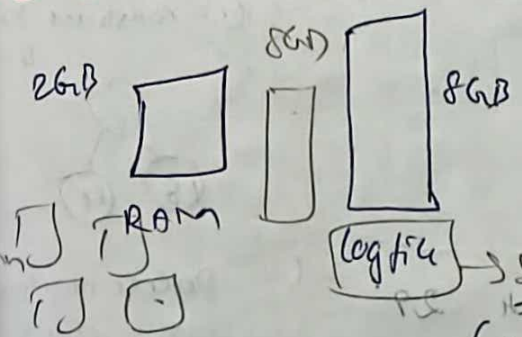
Merge Sort:

8, 3, 2, 9, 7, 1, 5, 4.

* It follows divide and conquer approach
* It divides the given array into 2 halves recursively & merge the sorted halves.



External sorting: How to sort very large data?



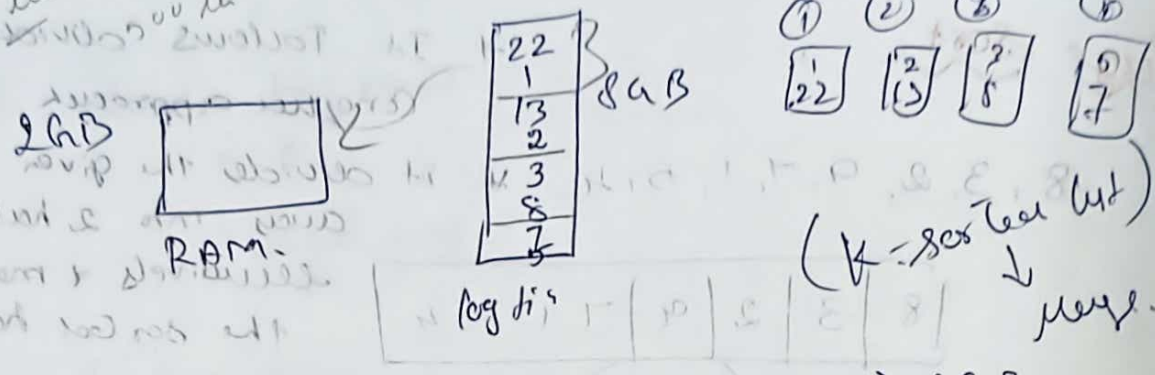
Sorting algorithm.

Logrotate

Ideal: sort logfile based on Timestamp.

Const: Sorting on RAM is not possible.

* logfile → RAM (sort)
* RAM of 8GB simple - buy new RAM (Vertical Scaling) (or) increase capacity
* 4 different RAMs utilize each slot of 2GB
  - 8GB of logfile - part size is 2GB
  - (Horizontal scaling)



1) Divide data into chunks of size → 2GB

No of chunks = $\frac{8}{2} = 4$ → 4

2) Read each chunk separately

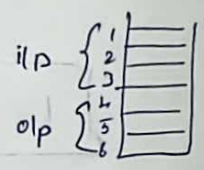
3) Sort each chunk one by one & store in HDD

4) Apply the technique to "Merge - K sorted list".

Example: Multiway Merge sort.

Ta, 17 3 29 56 24 18 4 9 10 6 45
36 11 43

Tape = 3

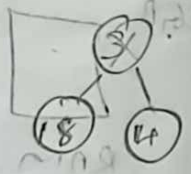


Tb1 3 17 29 6 36 45

Tb2 18 24 56 11 43

Tb3 4 9 10

(i) construct heap tree.

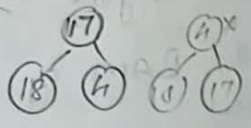


(ii) Delete min heap

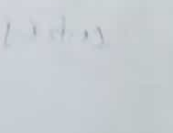
(iii) Next store in

Ta1 3 4 9 10 17 18 24 29

Ta2 6 11 36 43 45



(iv) construct heap



(v) Delete min heap

Tbl 3 4 6 9 10 11 17 18

21 29 36 43 45 56

Number of pass :- $\log_k [n(k+1)+1]$

Hashing

⇒ Hashing is a technique (or) process of mapping keys into the hash table by using hash function.

⇒ It is done for faster access to elements

⇒ The efficiency of mapping depends on the efficiency of hash function used.

Key → unique Integer that is used for indexing

Value → Data that are associated with keys.

Hash(key) = index

h(x%10) = index

Example: List [11, 12, 13, 14, 15]

h(x) = x%10 → hash function.

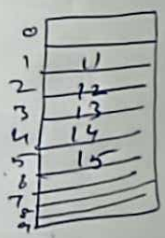
↳ tablesize.

↳ h(x) = key % m → where m is size of hash table

h(11) = 11%10 ⇒ 1 12%10 ⇒ 2 13%10 = 3

14%10 = 4

15%10 = 5



Hash collision:

When the hash function generates the same index for multiple keys, then there will be a conflict. This is called as a hash collision.

Example: 6, 26, Here both leads to Index 6

∴ result is collision.

Collision resolving Techniques:

1. Open Hashing / Closed Addressing → Separate Chaining
2. Closed Hashing / Open Addressing.
   - → linear probing
   - → Quadratic probing
   - → open addo Double Hashing.

(1) Closed Addressing (Separate chaining)

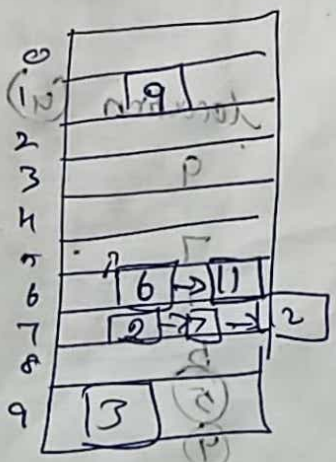
In chaining method if hash function produces the same index for multiple elements, then these elements are stored in same index by using Linked list.

1. Construct hash table for the following key values 3, 2, 9, 6, 11, 13, 7, 12 with hash function $h(K) = 2K+3$ with table size 10. use division method to store these values. Closed addressing

Solution: 3, 2, 9, 6, 11, 13, 7, 12 division method $K \% 10$

$h(K) = 2K+3$ & $m=10$ $K = 2K+3$

Key	Index	Hash table
3	$2(3)+3 \% 10 = 9$	
2	$2(2)+3 \% 10 = 7$	
9	$2(9)+3 \% 10 = 1$	
6	$2(6)+3 \% 10 = 5$	
11	$2(11)+3 \% 10 = 5$	
13	$2(13)+3 \% 10 = 9$	
7	$2(7)+3 \% 10 = 7$	
12	$2(12)+3 \% 10 = 7$	



2) Open Addressing:

- Linear probing
- Quadratic probing
- Double Hashing

(1) Linear probing:

In linear probing, if hash function produces the same index for multiple elements, then search the next empty location in the array until a final an empty cell.

Example:

Do hash table construction using open addressing

$A = 3, 2, 9, 6, 11, 13, 7, 12$

$h(k) = 2k + 3$

$m = 10$

Note: Insert 'Ki' at first free location from $(h(k)+i) \% m$ where $i = 0$ to $m-1$

key	location (h)	probes
3	9	1
2	7	1
9	1	1
6	5	1
11	(5)	2
13	(9)	2
7	7	2
12	7	6

Hash table

0	13
1	9
2	12
3	
4	
5	6
6	11
7	2
8	7
9	3

(5) $(u+i) \% m$

$(5+0) \% 10 = 5$

$(5+1) \% 10 = 6$

(7) $(u+i) \% m$

$(7+0) \% 10 = 7$

$(7+1) \% 10 = 8$

$(7+2) \% 10 = 9$

$(7+3) \% 10 = 0$

$(7+4) \% 10 = 1$

$(7+5) \% 10 = 2$

(9) $(u+i) \% m$

$(9+0) \% 10 = 9$

$(9+1) \% 10 = 0$

$(9+2) \% 10 =$

(iv) Quadratic probing :-

In Quadratic probing, if hash function produces same index for multiple elements, then search for i^2th slot in the i^{th} iteration.

	0
	1
	2
	3
	4
	5
	6
	7
	8
	9

Example:-

3, 2, 9, 6, 11, 13, 7, 12, $h(k) = 2k+3$

$m = 10$

Note: Next ci at first free location from 1

$= (u+i^2) \% m$, where $i = 0$ to $m-1$

Key	Location (ci)	probe
3	$(2\times3+3) \% 10 = 9$	1
2	7	1
9	1	1
6	5	1
11	(5)	2
13	(9)	2
7	7	2
12	7	5

Hash table

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	
9	3

$(u+i^2) \% m$

$(5+0) \% 10 = 5$

$(5+1) \% 10 = 6$

$(9+0) \% 10 = 9$

$(9+1) \% 10 = 0$

(7) $(7+0) \% 10 = 7$

$(7+1) \% 10 = 8$

$(7+4) \% 10 = 1$

$(7+9) \% 10 = 6$

$(7+10) \% 10 = 3$

Double Hashing:

1: We use 2 hash function.

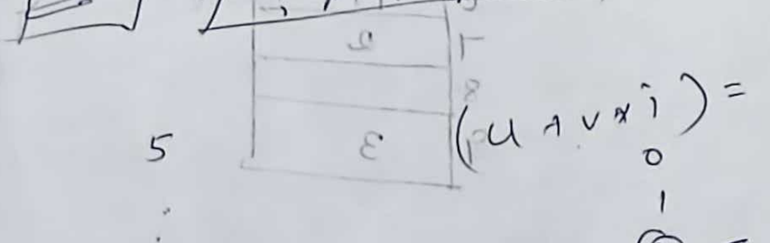
1. $hash_1(key) = key \% tablesize$;
   If collision occurs go to second function.
2. $hash_2(key) = R - (key \bmod R)$

* R is prime number
* If tablesize is 10 then 7 is R

Q: 89 18 49 58 69.

0	69
1	
2	
3	58
4	
5	49
6	
7	
8	18
9	89

* $89 \% 10 = 9$
* $18 \% 10 = 8$
* $49 \% 10 = 9$ (collision)
  → $R - (key \% size)$
  → $7 - (49 \% R)$
  → $7 - 0 = 7$
* $58 \% 10 = 8$
  → $7 - (58 \% 7)$
  → $7 - 2$
  → 5
* $69 \% 10 = 9$
  → $7 - 69 \% 7$
  → $7 - 6 = 1$



$(u + v \times i) =$

Rehashing

When the hash table becomes too full in open addressing hashing, the successive insertion operation will take more time to complete. To overcome this situation, rehashing technique is used.

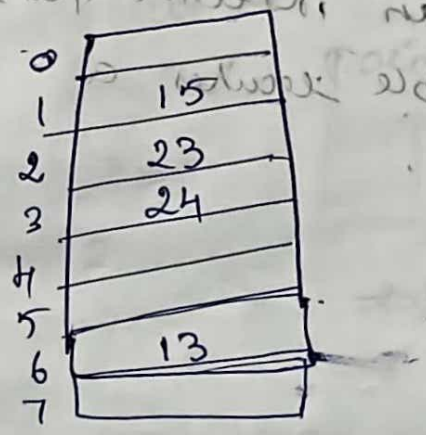
In rehashing technique another hash table is build that is about twice as big + scan down the entire original hash table, computing the new hash value for each element & inserting it in the new table.

Example: Insert [13, 15, 24, 6, 23]

hash table of size 7.

The hash function is $h(x) = x \bmod 7$

Key	Index
13	13 % 7 = 6
15	15 % 7 = 1
24	24 % 7 = 3
6	6 % 7 = 6
23	23 % 7 = 2



Next prime 17

New table is 70% full. So, new table is created. The size of this table is 17, because this is the first prime which is twice as large as the old table size

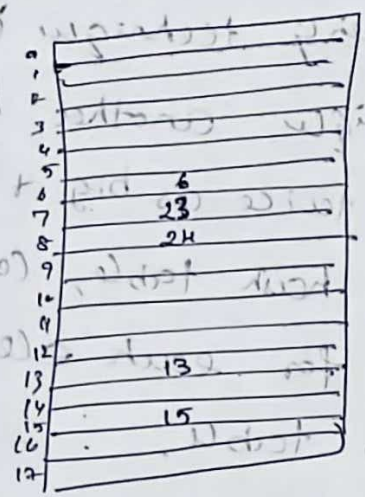
The new hash function is then,

$$h(x) = x \bmod 17$$

Rehashing

Now the old table is scanned & the elements are inserted into the new table. The resulting table is

Key	Index
13	13 % 17 => 13
15	
24	
6	
23	



This entire operation is called rehashing.

∴ Rehashing can be implemented in several ways:

* Rehash as soon as the table is half full
* Rehash only when an insertion fails
* Rehash when the table reaches a certain load factor.

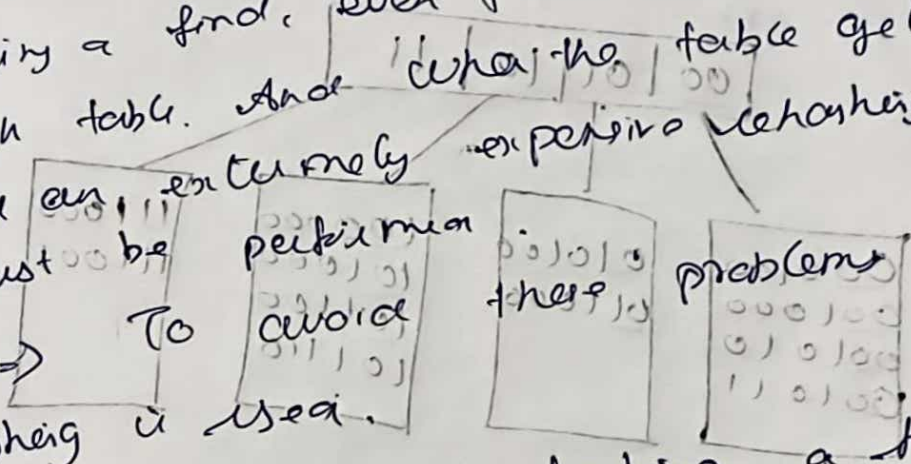
Disadvantages:

* Expensive method.

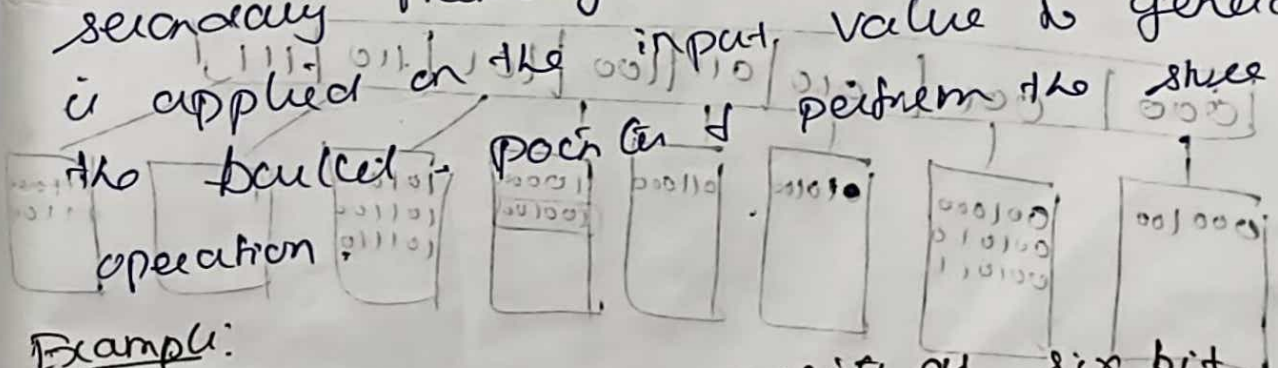
Extendible Hashing

⇒ The major problems in using open hashing or closed hashing is that collisions could cause several blocks to be examined during a find, even for a very clustered hash table. And when the table gets too full, an extremely expensive rehashing step must be performed.

⇒ To avoid these problems extendible hashing is used.



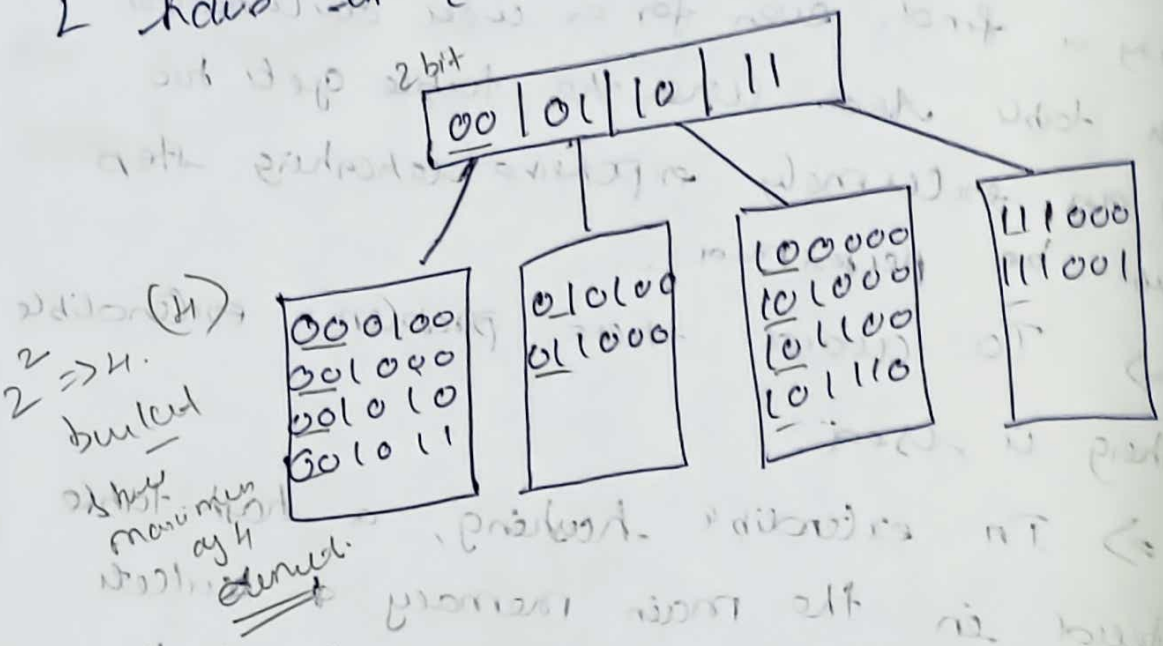
⇒ In extendible hashing, a hash table is stored in the main memory & buckets are stored in the disk. Each value in the hash table is a pointer to a bucket in the secondary memory. The hash function is applied on the input value to generate the bucket pointer & perform the store operation.



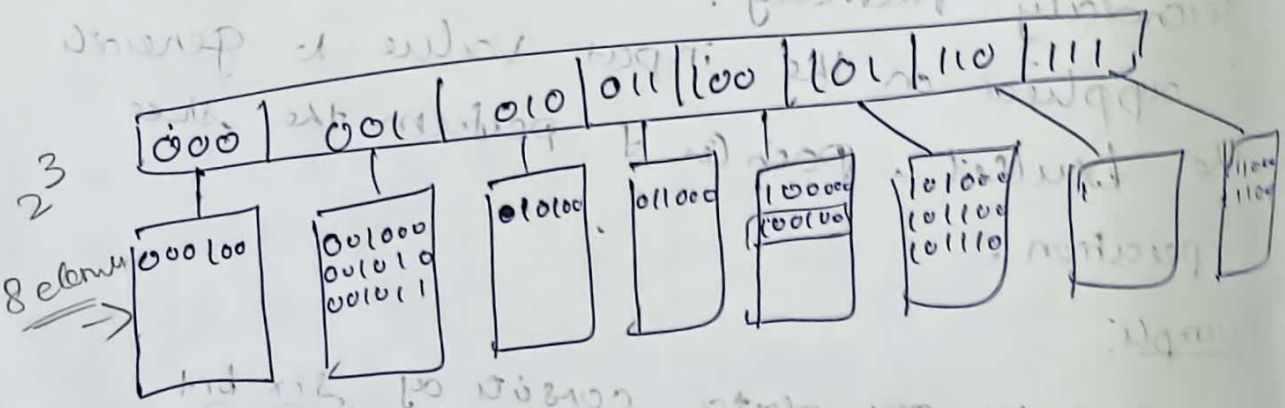
Example:

Consider our data consists of six bit integer value. The root of the tree contains four pointers determined by the leading two bits of the data. Each leaf has up to m=4 elements

⇒ The number of entries in directory is 2^D where D is the number of leading bits that all the elements of some leaf have in common $d_L <= D$.



⇒ suppose 100100



Advantages:

- very simple
- Quick access time for insert & find operation on large database.

Disadvantages:

- This algorithm does not work, even if there are m duplicates.