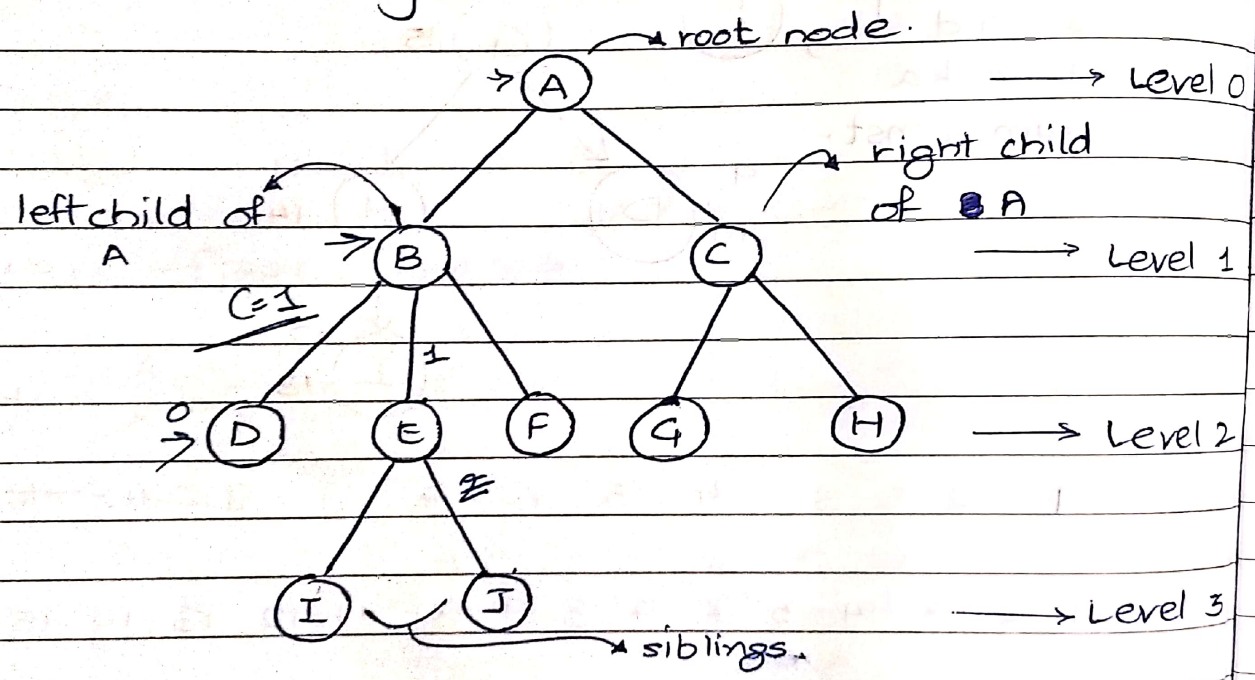
TREES

Tree is a non-linear data structure which ~~organizes~~ organizes data in hierarchial structure & this is a recursive definition.

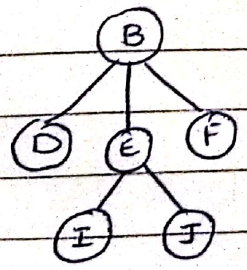
In a tree with 'N' nodes, there will be a maximum of 'N-1' edges.

In a tree, every element is called as 'NODE'.



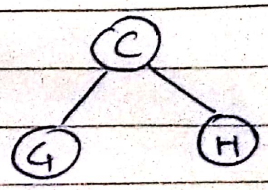
Internal nodes → nodes have children.
Leaf " → " not having children.

For A, →



→ this is left subtree of A.

&



→ this is right subtree of A.

→ Degree = 0 {leaf node OR terminal node}

Degree of B = 3 } ∴ Degree = no. of children.
Degree of A = 2 }

→ Height of Tree = No. of edges on the longest path to the deepest leaf node from the root node.

~~& Height of a Node :-~~

~~↳ No. of edges from that ~~deepest~~ particular ~~leaf~~ node to ~~that~~ the ~~particular~~ deepest node. is height of ~~that~~ node.~~

~~e.g. Height of H = 0~~
~~Height of E = 1~~
~~Height of B = 2~~

→ & Height of a Node:-

↳ No. of edges from that particular node to the deepest node is height of that node.

e.g. Height of H = 0
" " E = 1
" " B = 2

→ Depth:

↳ No. of edges from that particular node to the root node.

∴ Depth of a node = Level of that node.

e.g. Depth of H = 3.
" " E = 2

→ TYPES OF TREES:-

(i) General.
(ii) Binary Tree.
(iii) Binary Search Tree.

• (i) General Tree:

↳ No restriction of no. of children for a parent.

• (ii) Binary Tree:

↳ a parent can have a max. of 2 children
i.e. no. of children allowed = {0, 1, 2}.

Types of BT:

(a) Full Binary Tree:

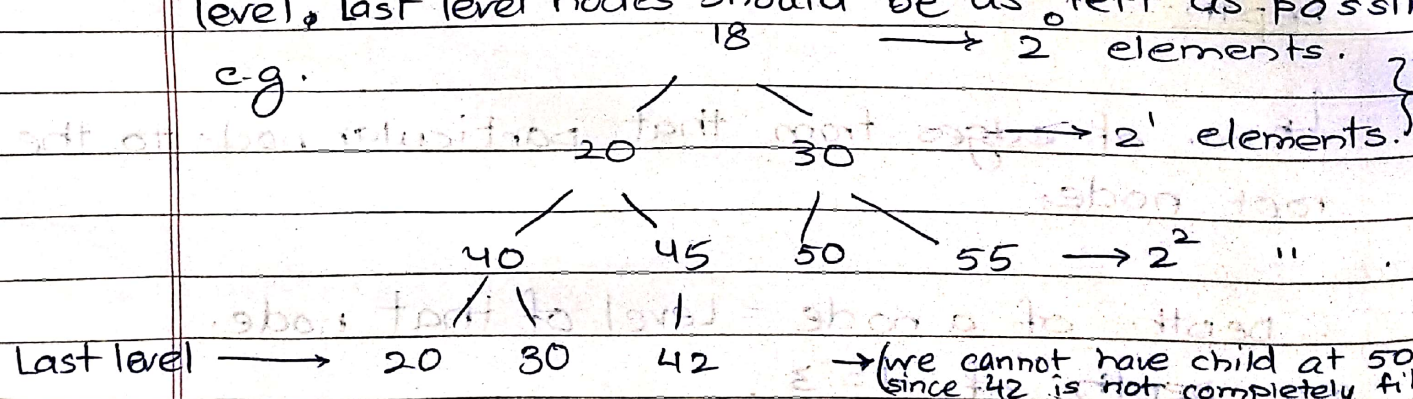
↳ has 0 or 2 children at ~~even~~ every parent node.

(b) Complete Binary Tree:

↳ all levels are completely filled except last level, last level nodes should be as (←) left as possible.

(means 2 children.)

e.g.



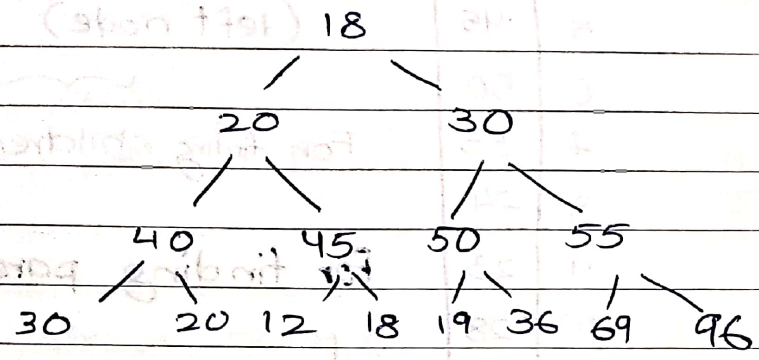
∴ In a B.T, max. possible elements in particular level = 2

Array Representation.

(c) Perfect Binary Tree:

All internal nodes should have 2 children & all leaves are at same level.

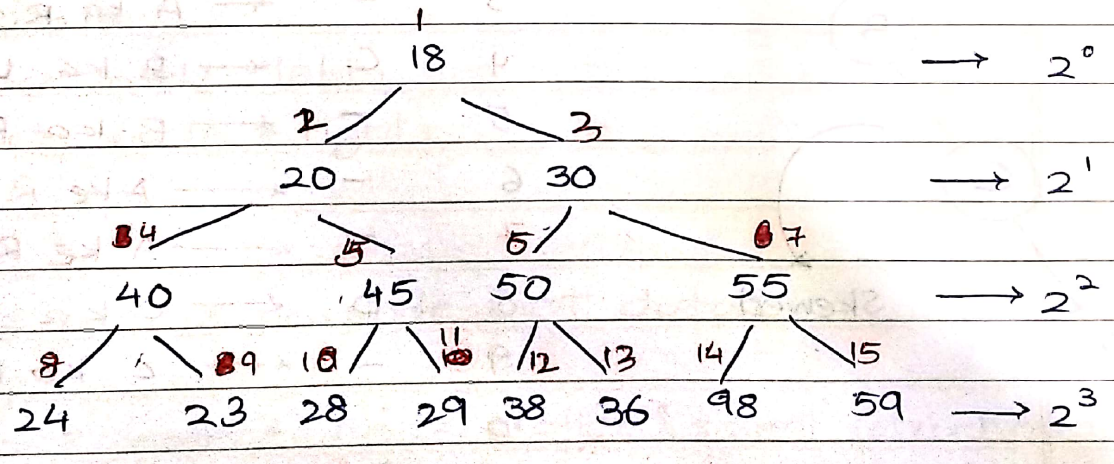
e.g.



Max. Total ~~of~~ no. of nodes in B.T. $= \sum_{l=0}^{h} 2^{l}$ h = height.

$h=3$, $\therefore \sum_{n=0}^{3} 2^{n} = 2^{0} + 2^{1} + 2^{2} + 2^{3} = 1+2+4+8$
$= 15.$

→ Array Representation of Perfect B.T:-



Array →

0	15
1	18
2	20
3	30
4	40
5	45
6	50
7	55
8	24
9	23
10	28
11	29
12	38
13	36
14	98
15	59

index no.

For i^{th} node

$2i$ (left node) $(2i+1)$ (right node)

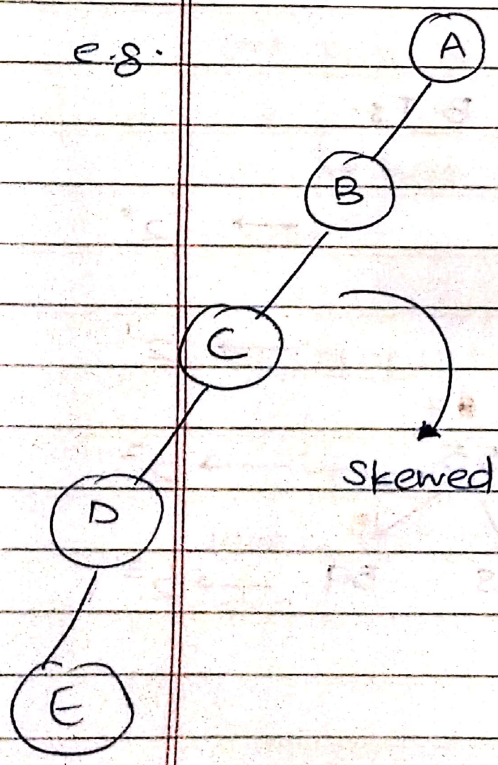
For finding children nodes.

For finding parent node:

↳ i = children node.

$\therefore \lfloor i/2 \rfloor$ = parent node.

e.g.



1	A	
2	B	← A ka Left (L)
3	–	← A ka Right (R)
4	C	← B ka L
5	–	← B ka R
6	–	← A ke R ka R
7	–	← A ke R ka L
8	D	← C ka L
9	–	← C ka R
10	⋮	
⋮	⋮	
16	E	

→ LINKED REPRESENTATION :-

typedef struct node * ~~tree_ptr~~ tree_pointer;

~~typedef struct node {~~

~~int data;~~
~~tree_pointer left_child, right_child;~~
~~}~~

typedef struct node {

int data;
tree_pointer left_child, right_child;
}

→ code :

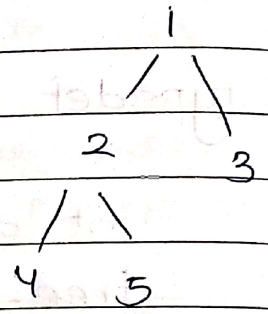
```
struct node {
int data;
struct node *left;
struct node *right;
}

struct node * newNode (int data)
{
  struct node * nn = malloc (sizeof (struct node));
  nn -> data = data;
  nn -> left = NULL;
  nn -> right = NULL;
  return nn;
}
```

```
int main( )
{
  struct node * root = newNode (1);

  root -> left = newNode (2);
  root -> right = newNode(3);
}
```

→ **TREE TRAVERSALS :-**



Depth First Traversals:

(a) Inorder (Left, Root, Right):

(4 - 2 - 5 - 1 - 3)

(b) Preorder (Root, Left, Right):

(1 - 2 - 4 - 5 - 3)

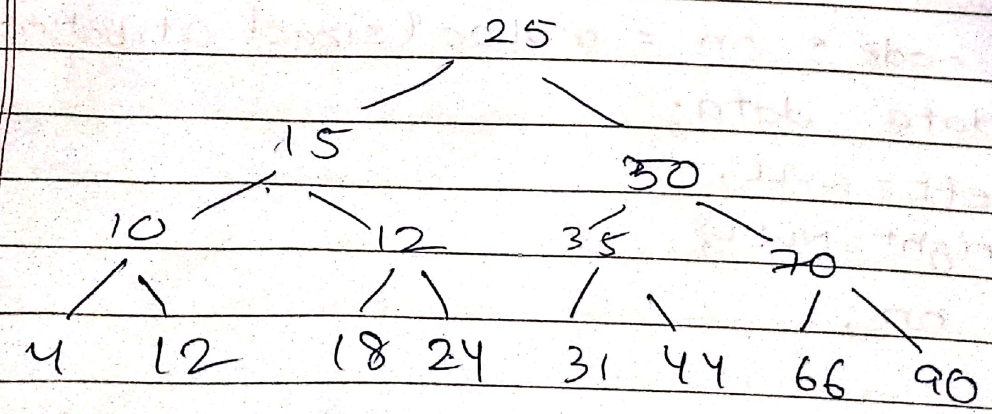
1st one is always root of tree

(c) Postorder (Left, Right, Root):

(4 - 5 - 2 - 3 - 1) . Last is always root of tree.

Breadth First or Level order Traversal : (1-2-3-4-5)

e.g



~~Inorder: 4, 10, 12, 15, 18, 12, 24, 31, 35, 44, 50, 66, 70,~~

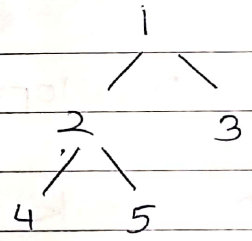
Inorder: (4, 10, 12, 15, 18, 12, 24, 25, 31, 35, 44, 50, 66, 70, 90)

Post: (4, 12, 10, 18, 24, 12, 15, 31, 44, 35, 66, 90, 70, 50, 25)

Pre : (25, 15, 10, 4, 12, 12, 18, 24, 50, 35, 31, 44, 70, 66, 90)

→ Inorder traversal code: → print nodes in inorder

```
void printInOrder (struct node * node)
{
    if (node == NULL)
        return;
    //first recur on left child
    printInOrder (node->left);
    // print node ka data
    cout << node -> data << " ";
    // recur on right child.
    printInOrder (node->right);
}
```



Internal Recursion Stack.

3
~~5~~
~~4~~
~~2~~
1

→ Preorder traversing code:

```
void printPreorder (struct node * node)
{
 if (node==NULL)
    return;

 cout << node->data<<" ";

 printPreorder (node->left);

 printPreorder (node->right);
}
```

Q→ Given: Inorder: E A C K F H D B G

Preorder: F A E K C D H G B.

Develop tree:

→ Look for first element in Preorder : F

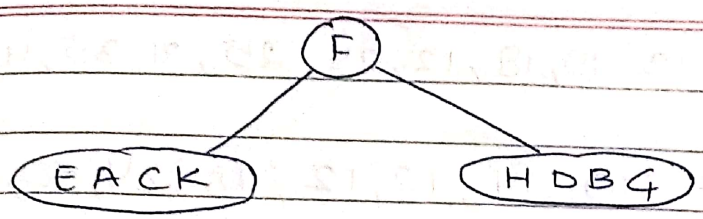
Now look for F in inorder:

E A C K F H D B G

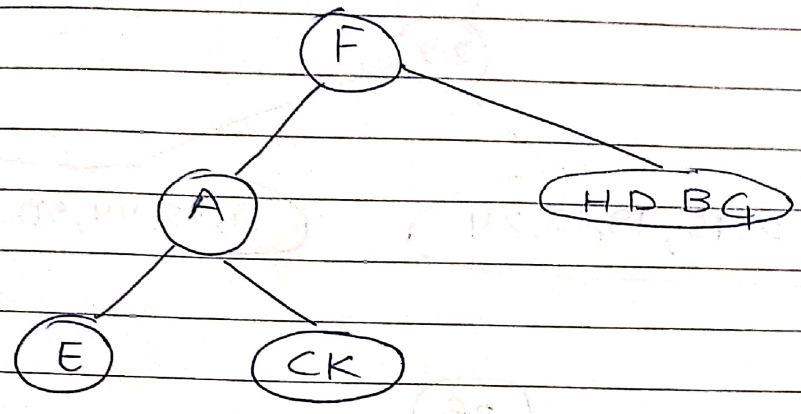
↓ Left subtree ↓ Root ↓ Right subtree

& go on. (Next look for A in E A C K)

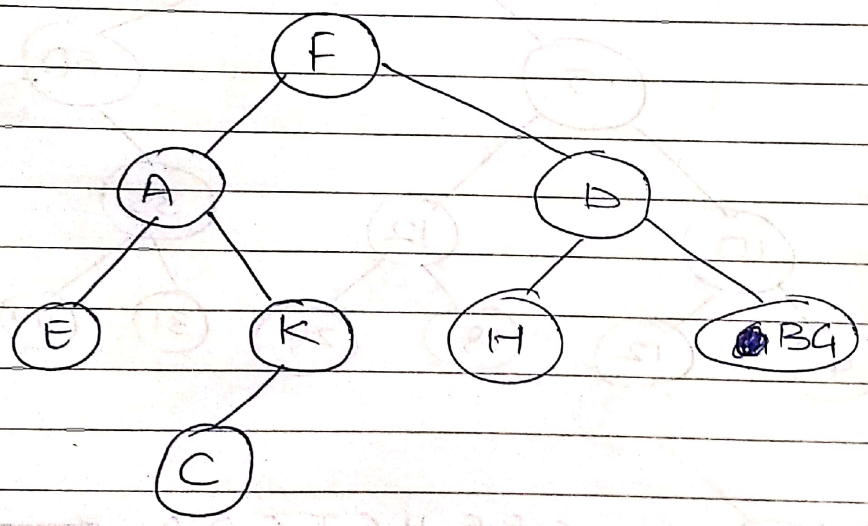
(i)



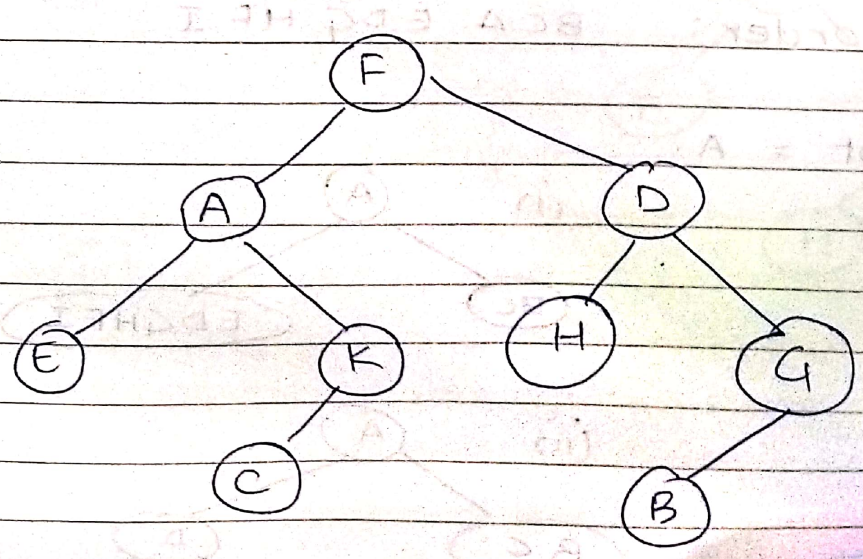
(ii)



(iii)



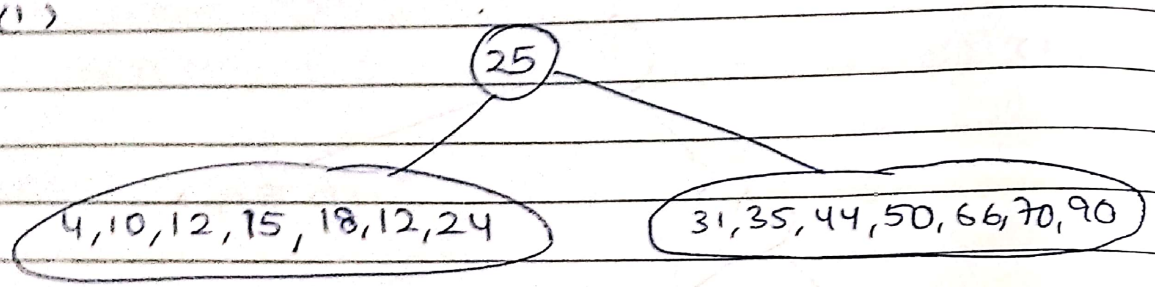
(iv)



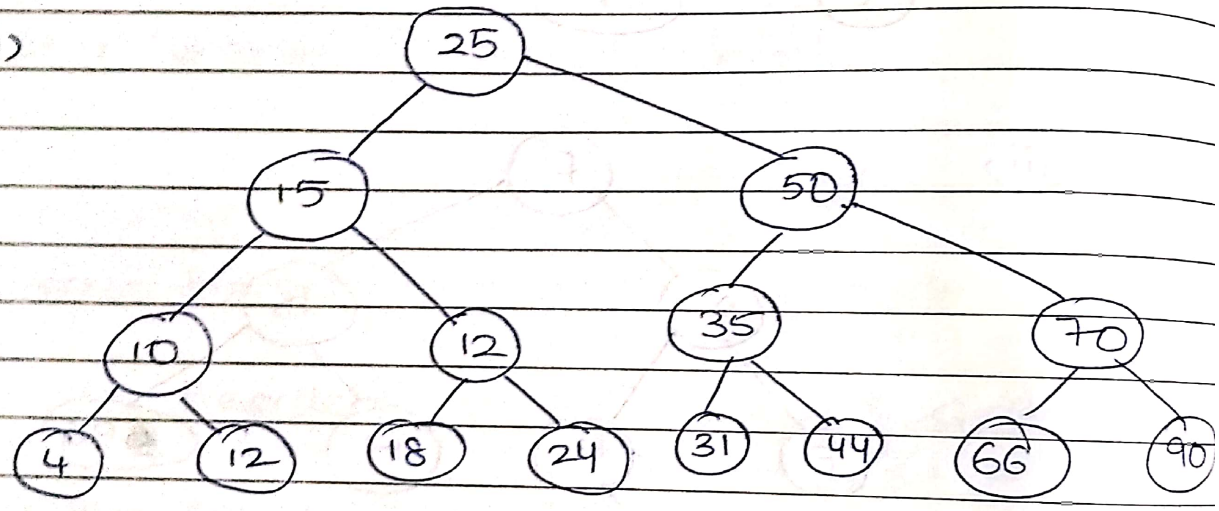
Q → In: (4, 10, 12, 15, 18, 12, 24, 25, 31, 35, 44, 50, 66, 70, 90)

Pre: (25, 15, 10, 4, 12, 12, 18, 24, 50, 35, 31, 44, 70, 66, 90)

→ (i)



(ii)

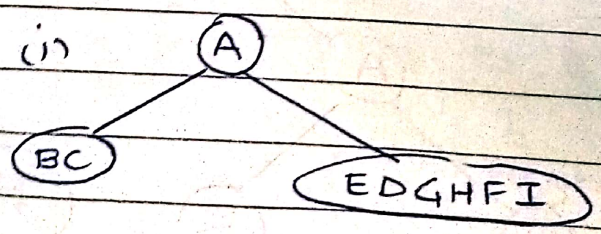


Q → Postorder: C B E H G I F D A ← start from back.

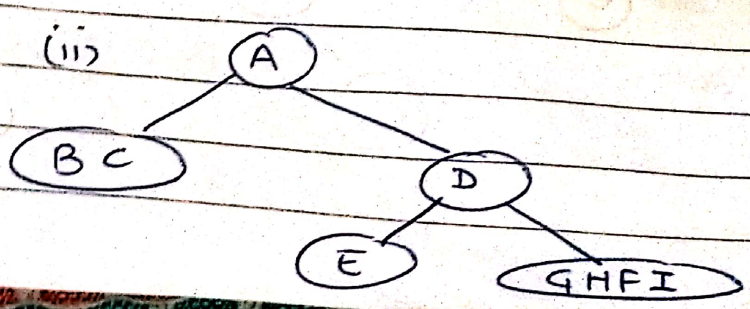
Inorder: B C A E D G H F I

→ Root = A.

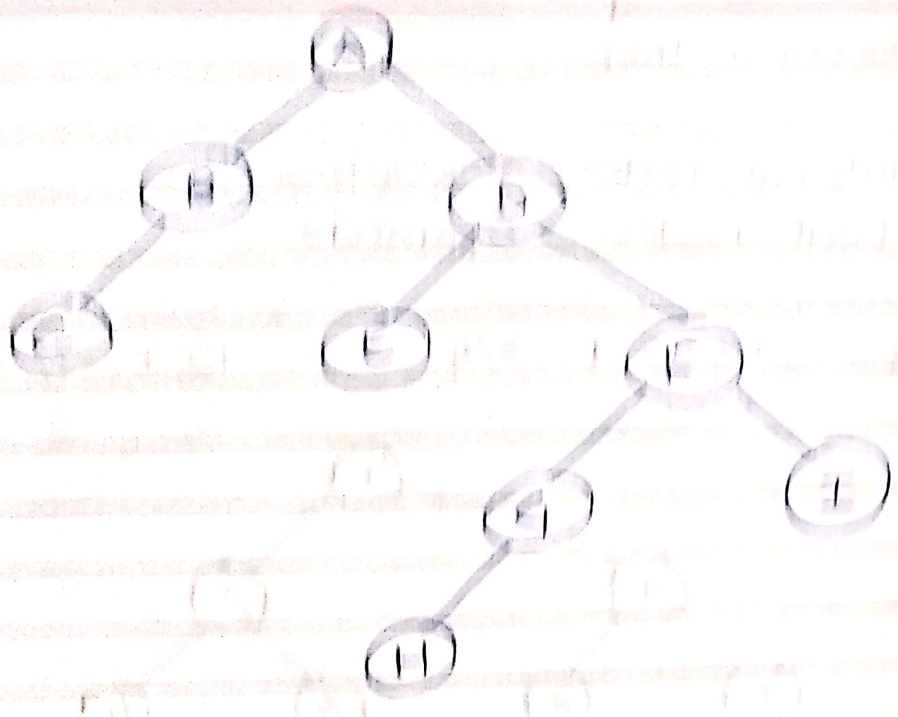
(i)



(ii)

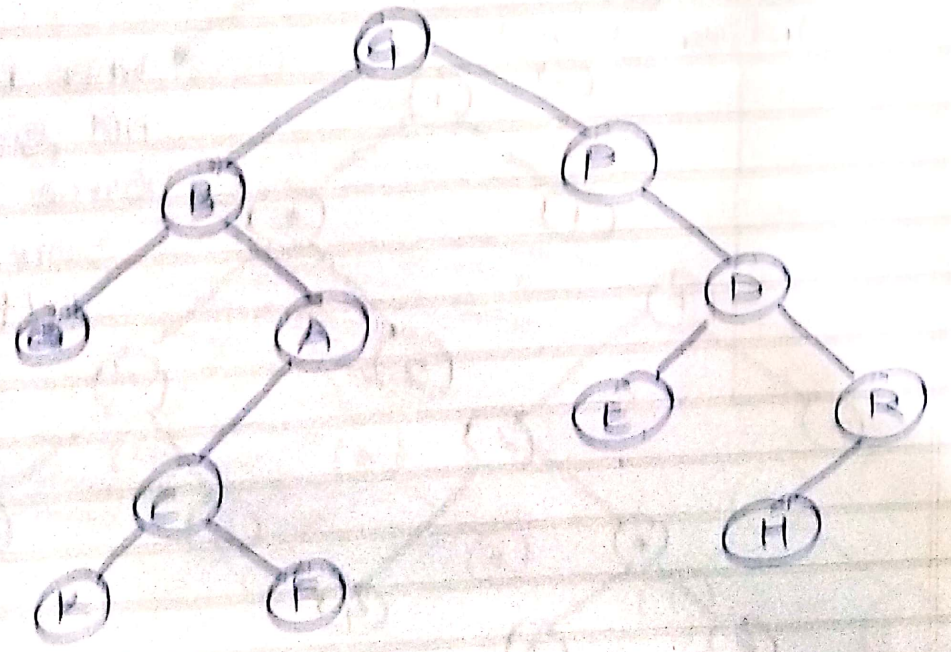


iii)



Q → Preorder : G , B, Q, A, C, K, F, P, D, E, R, H

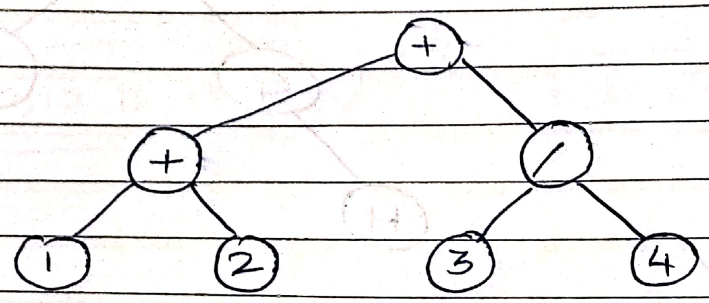
Inorder : Q, B, K, C, F, A, G, P, E, D, H, R



→ EXPRESSION TREE :-

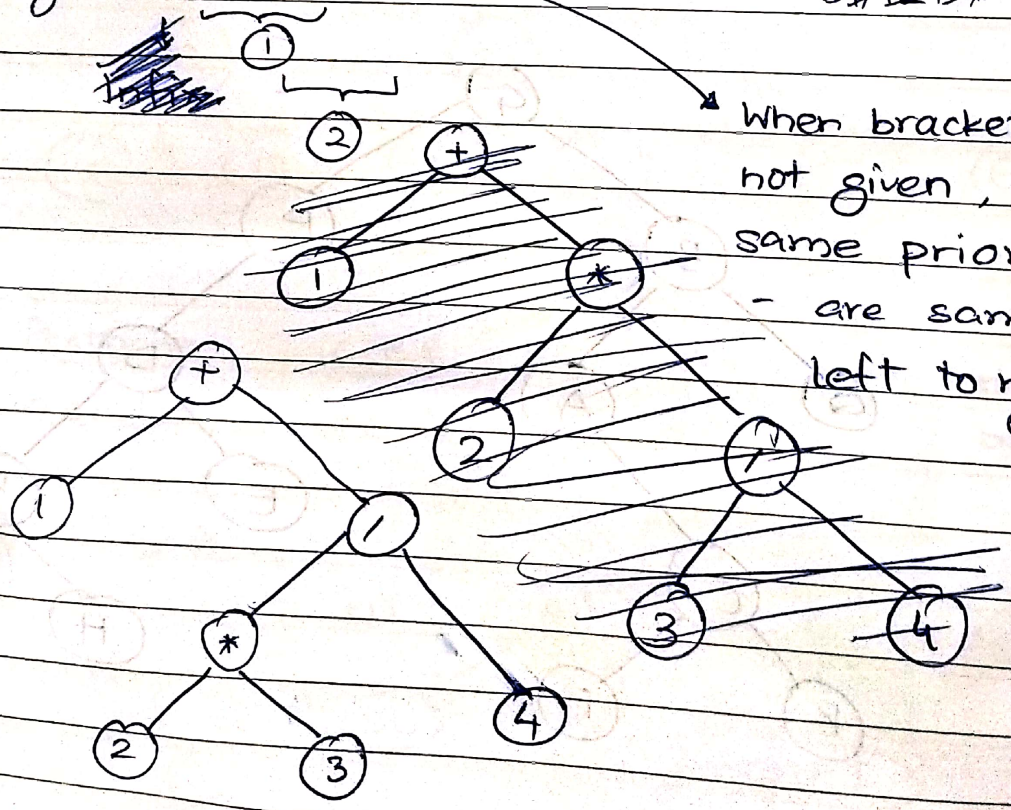
↳ Internal nodes → Operators.
Leaf nodes → Operands.

e.g. 1 + 2 + 3/4 12 + 34/ +



Inorder: (1 + 2) + (3/4)

e.g. 1 + 2 * 3 / 4 ~~34/12+*~~

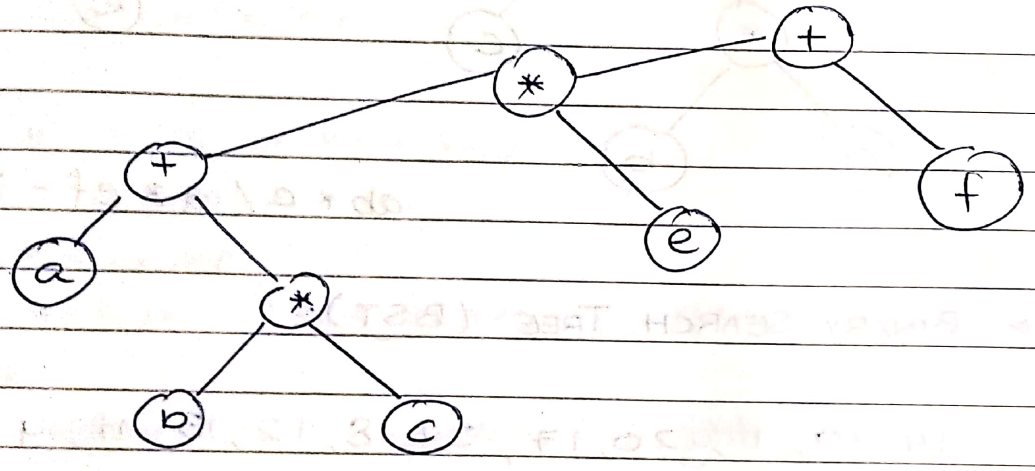


When brackets are not given, * & / are same priority & + & - are same priority left to right precedence

POSTFIX :-

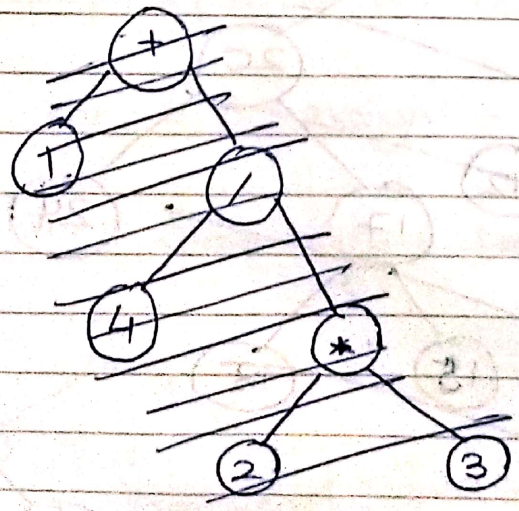
Expression : abc * + e * f +

when operand is encountered push into stack.
When operator " " , pop 2 elements from stack & push answer in stack.

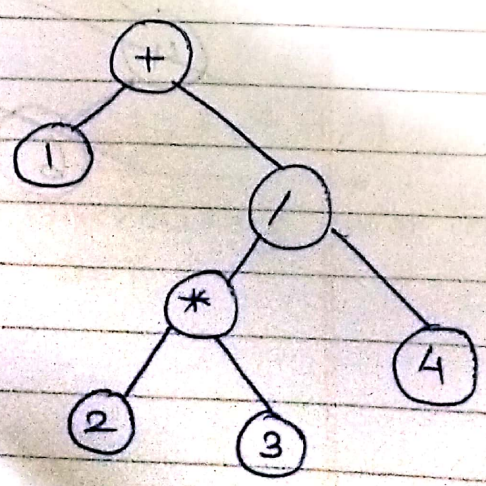


e.g. 1 23 * 4 / + → 1 + 2 * 3 / 4
(Postfix form)

4
~~3~~
~~2~~
1



~~4/(2*3)~~
~~4~~
~~2*3~~
~~3~~
~~2~~
~~1~~

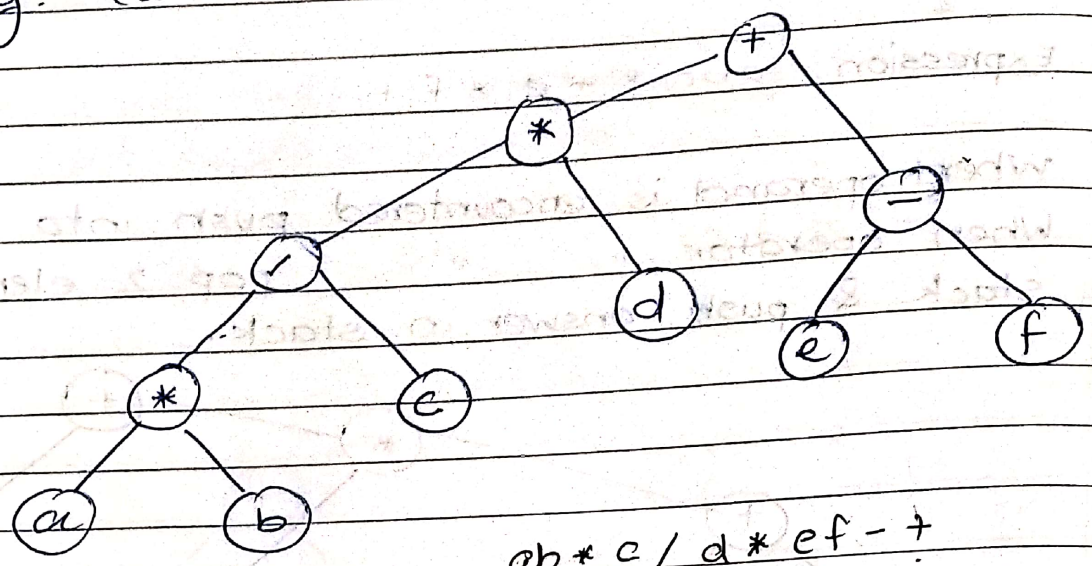


$1 + \frac{2}{3} + 4$ $1 + (2/3) + 4$ $1 + \frac{2}{4}$

ab * c/d * ef - +

L to R.

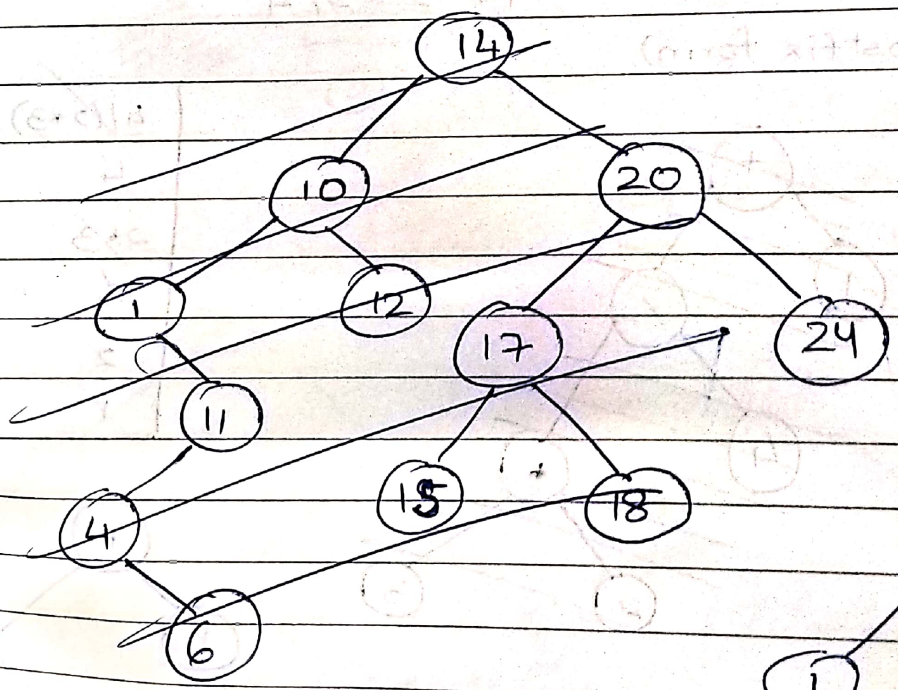
eg: (a*b) / c * d + (e-f)

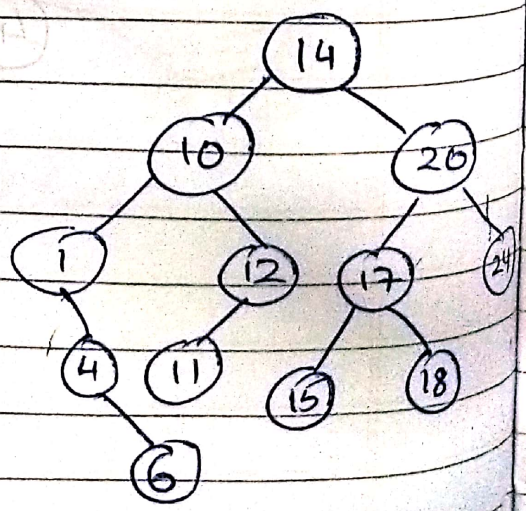


ab * c / d * ef - +

→ BINARY SEARCH TREE (BST):-

14, 10, 1, 20, 17, 24, 18, 12, 15, 11, 4, 6.





Left subtree ka leftmost element → smallest no.

Right subtree ka rightmost element → largest no.

→ ~~BFS~~ BST Algorithm:

TREE-SEARCH (x, k)

```
1. if x == NIL or k = x.key
2.         return x
3. if k < x.key
4.         return TREE-SEARCH (x.left, k)
5. else
6.         return TREE-SEARCH (x.right, k)
```

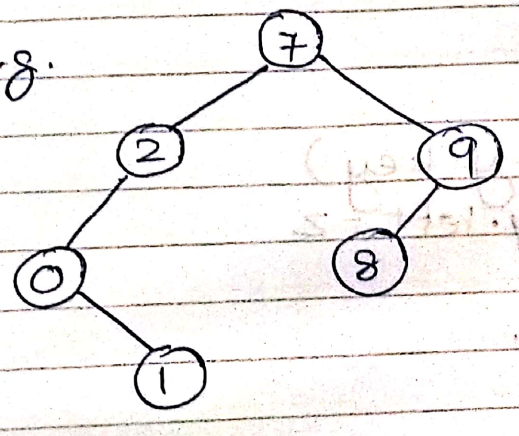
↳ Time-complexity:

Usually depends on height of tree.

& For C.B.T. height = logn.

∴ Time complexity = ~~lena~~ logn.

e.g.



Inorder: 0, 1, 2, 7, 8, 9 → increasing order in BST.

- TREE - MAXIMUM (x)

```
while x.right ≠ NIL
    x = x.right
return x.
```

- TREE - MINIMUM (x)

```
while x.left ≠ NIL
    x = x.left
return x.
```

→ BST insertion Algorithm:

```
TREE-INSERT(T, z)
y = NIL
x = T.root.
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
z.p = y
if y == NIL
    T.root = z
else if (z.key < y.key)
    ~~z.key = y.key~~ y.left = z
else
    y.right = z.
```

Q→ Suppose we have nos. b/w 1 to 1000 in a BST. & we wish to find no. 363. Now which of the following sequences could not be the sequence of ~~note~~ nodes examined.

(a) 2, 252, 401, 398, 330, 344, 397, 363.

(b) ~~920~~ 924, 220, 911, 244, 898, 258, 362, 363.

(c) 925, 202, 911, 240, 912, ~~914~~ 245, 363.

(d) 2, 399, 387, 219, 266, 382, 381, 278, 363.

(e) 935, 278, 347, 621, 299, 392, 358, 363.

AVL Trees, Red Black Trees, Spray Trees. all are egs of height maintaining tree.

Do read Red-Black Trees.

→ AVL Trees :-

Adelson - Velskii - Landson.

- self balancing tree. (by maintaining height of tree)
- All operations → Time Complexity : $O(\log n)$ in avg & worst case

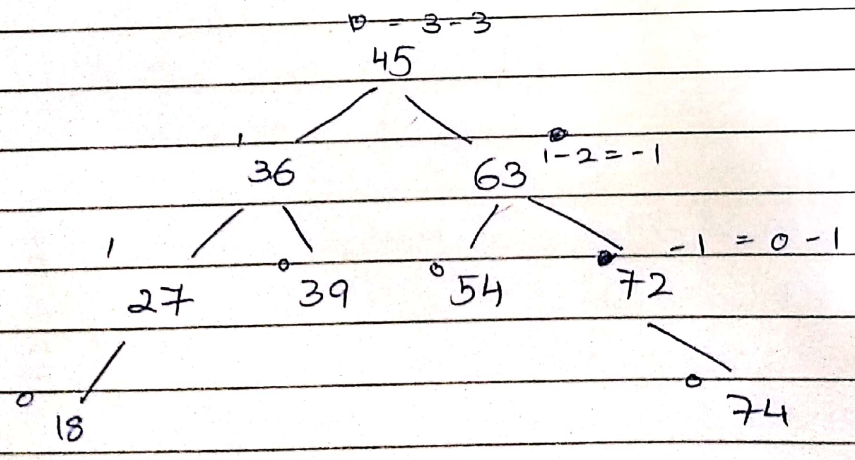
- Balance factor (B.F.) = height of left subtree − height of right subtree.

- Balanced tree if every node's B.F. is either :

Acceptable B.F values = {1, 0, −1}
1 → left heavy
0 → L = R
−1 → right heavy.

AVL, Red - Black, Spray Trees are all Binary search Trees but in a balanced form.

e.g.



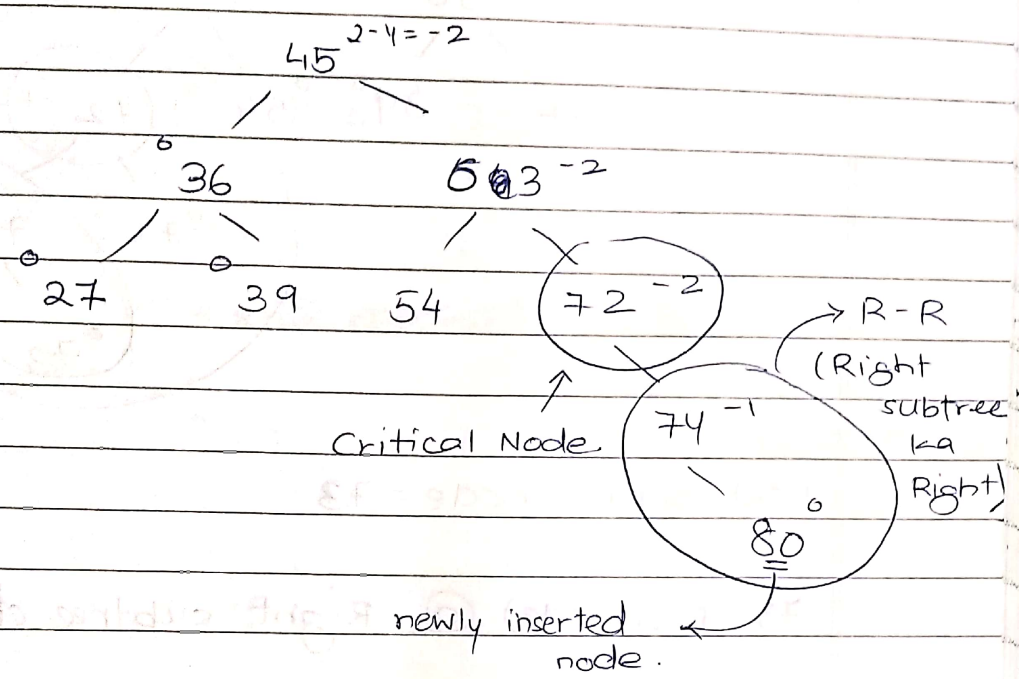
IF B.F of a node goes out of range i.e. (−1, 0, 1) then we try to bring B.F in range by doing rotations

Critical node : nearest ancestor of newly inserted node having B.F = 2 or −2.

Always check for critical node from newly inserted node.

→ For B.F = -2 :-

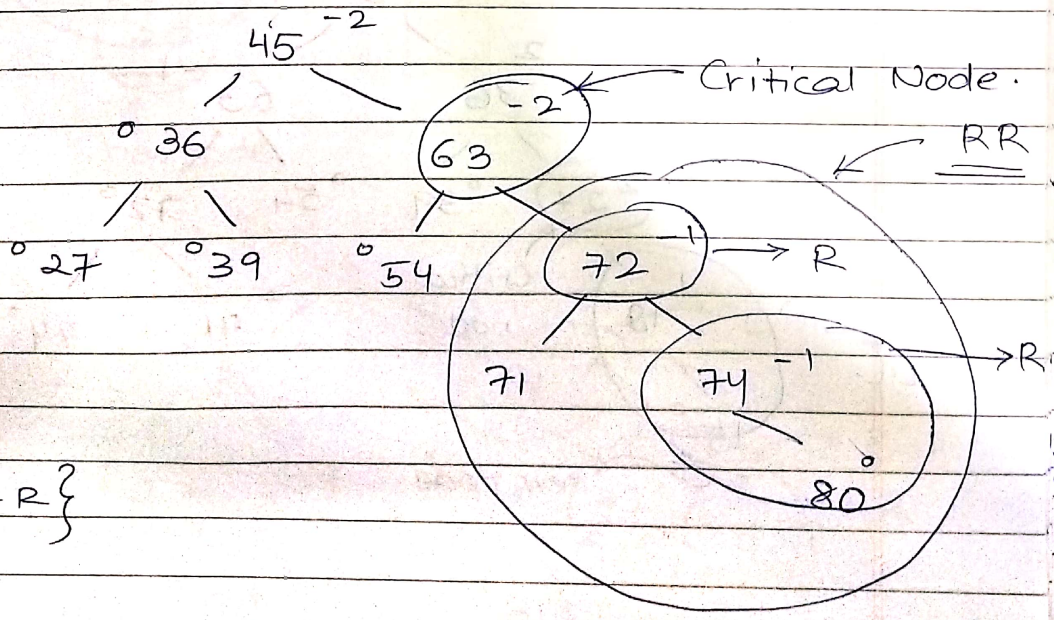
e.g.



New node is R-R (Right subtree of Right subtree)

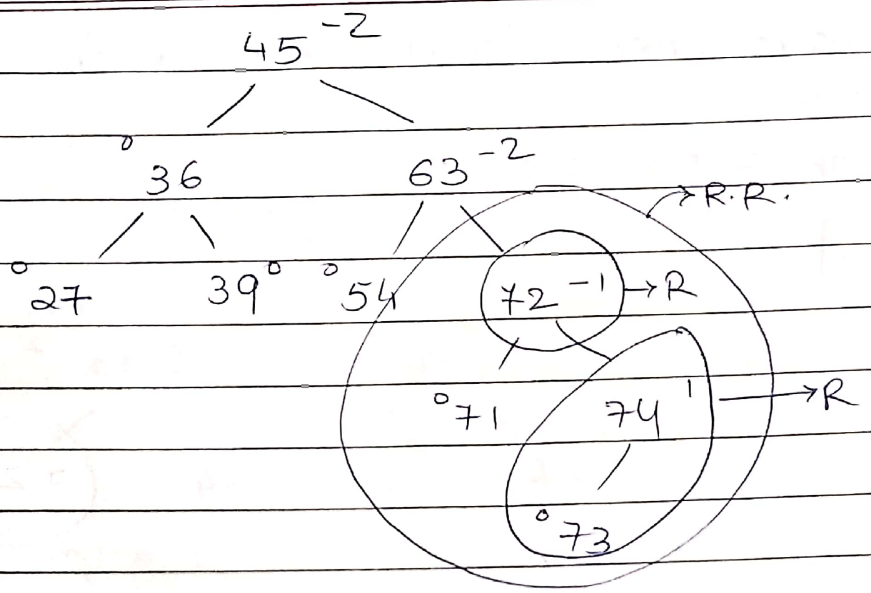
↳ for B.F = -2.

Now, if ~~newly~~ last inserted node = 80.



{80 = R.R}

e.g.



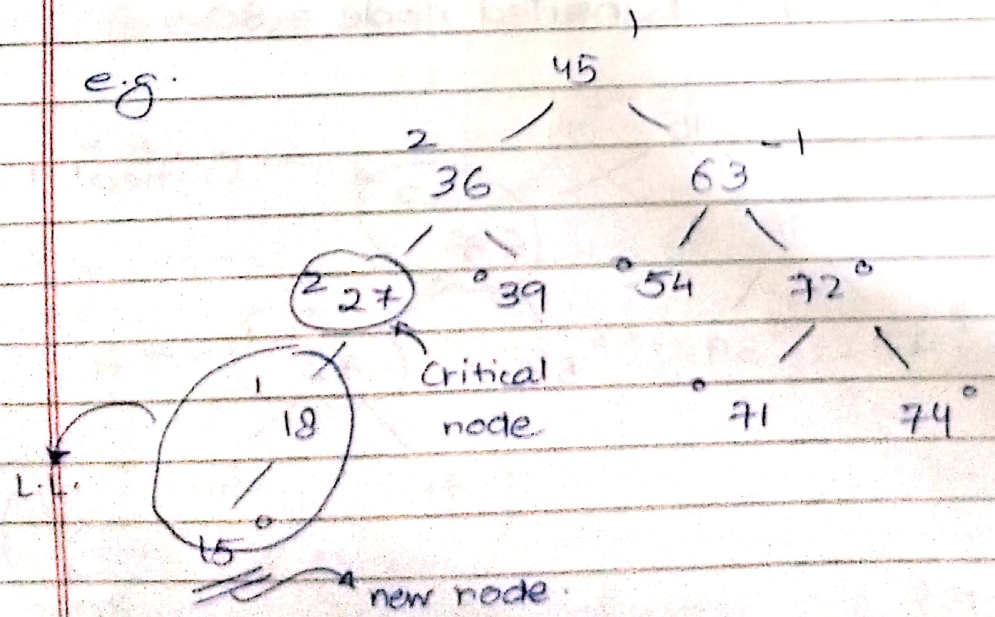
Last added node = 73.

73 (New node) @ Right subtree of right.

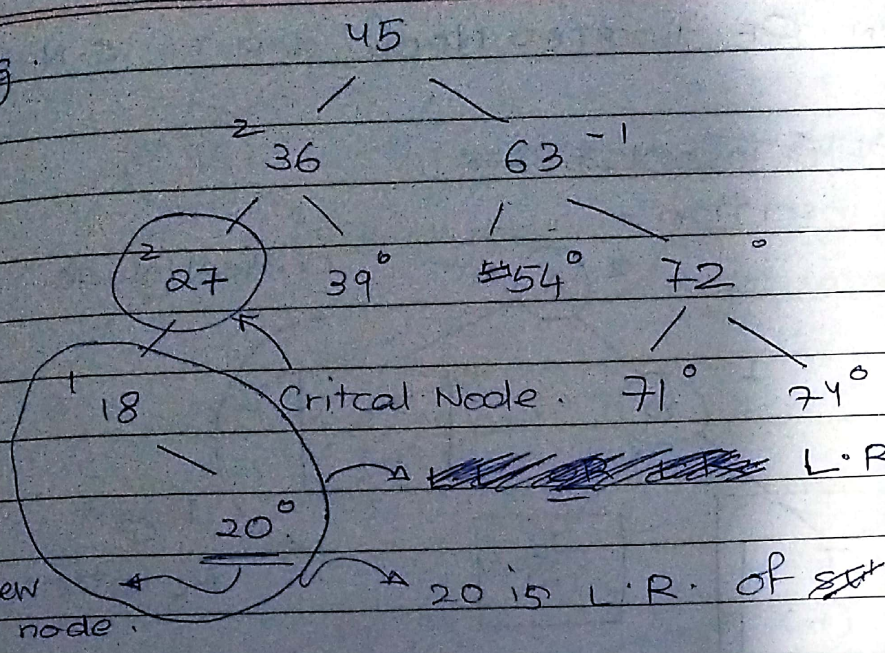
Always there are 2 L's or 2 R's.
OR LR or RL.

→ For B.F = 2.

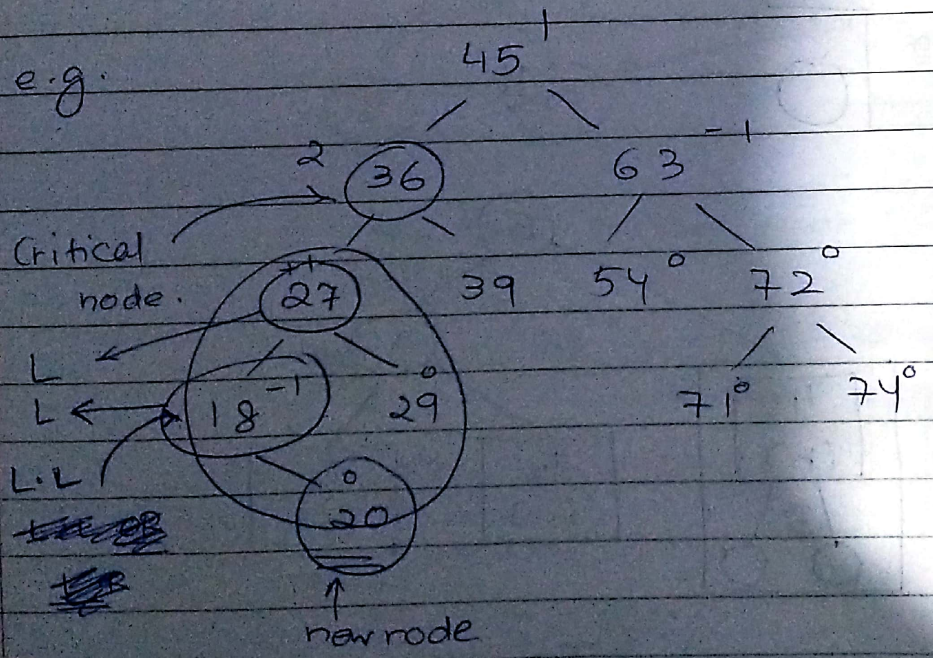
e.g.



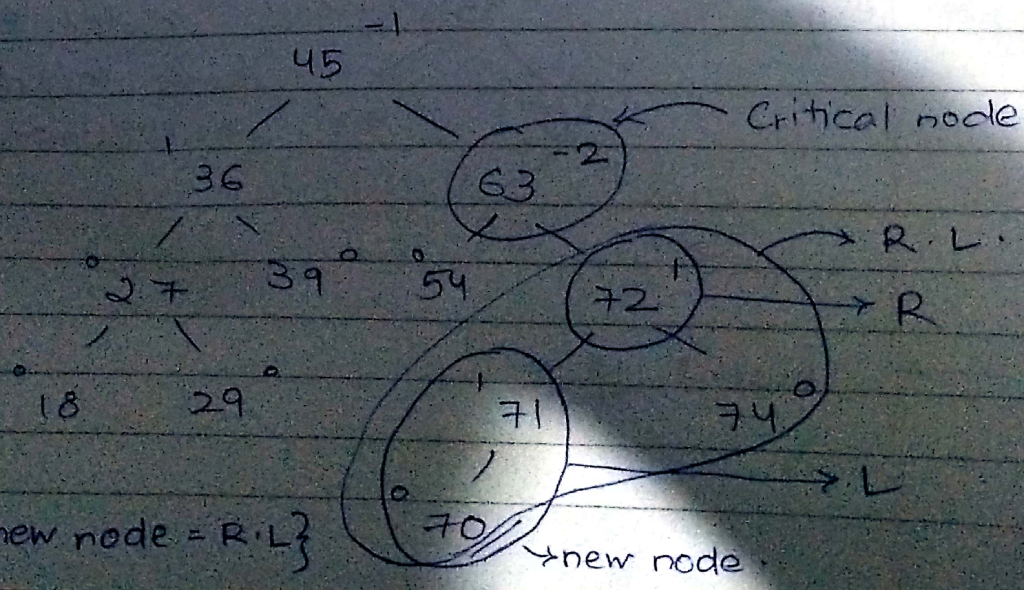
e.g.



e.g.



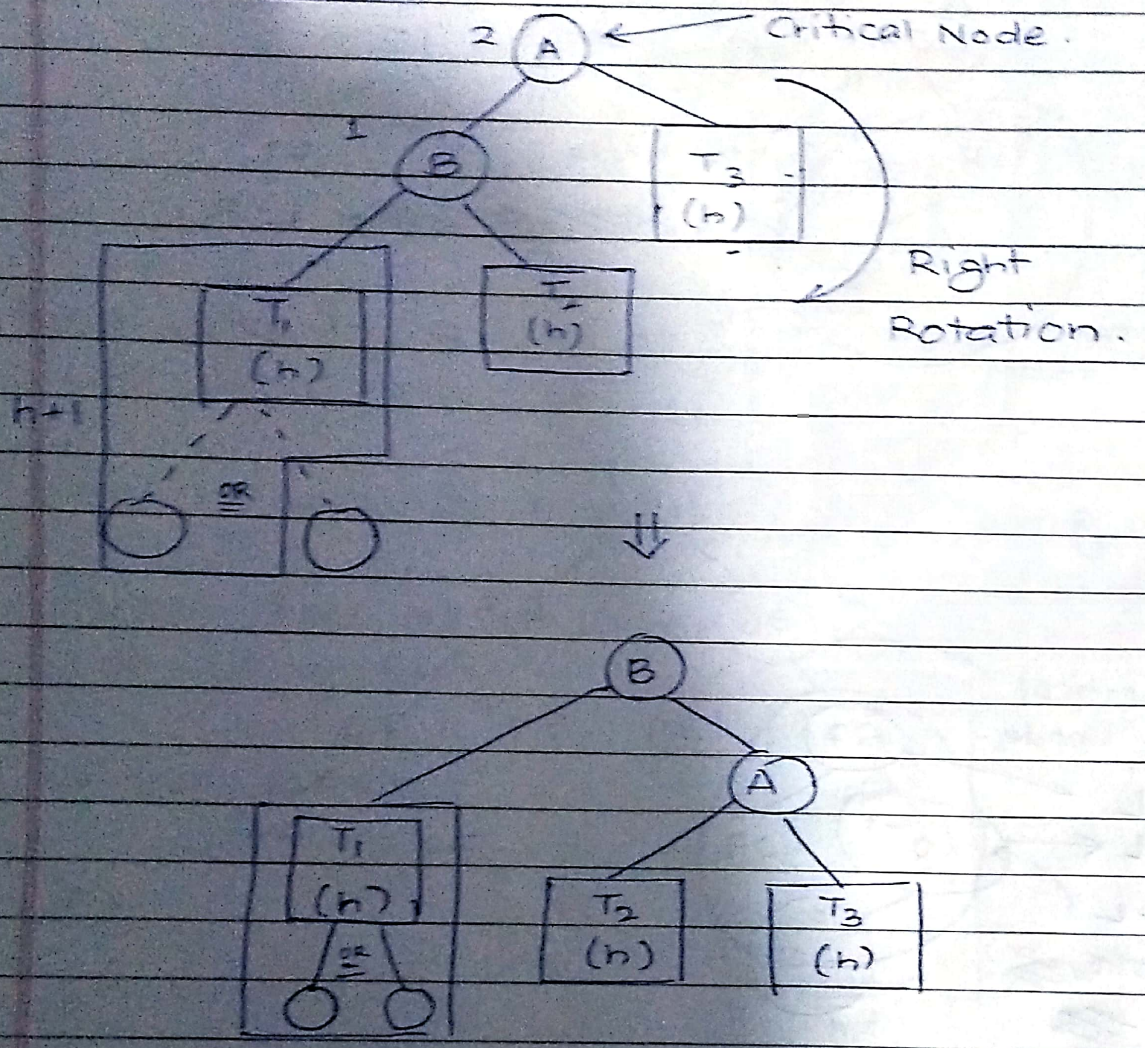
e.g.



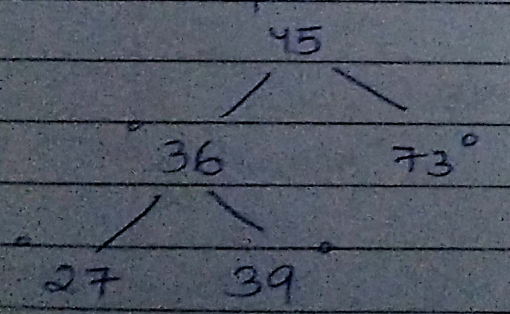
{70 = new node = R.L}

→ POSITION OF INSERTED NODE W.R.T. C.N :-

(i) L.L insertion :-

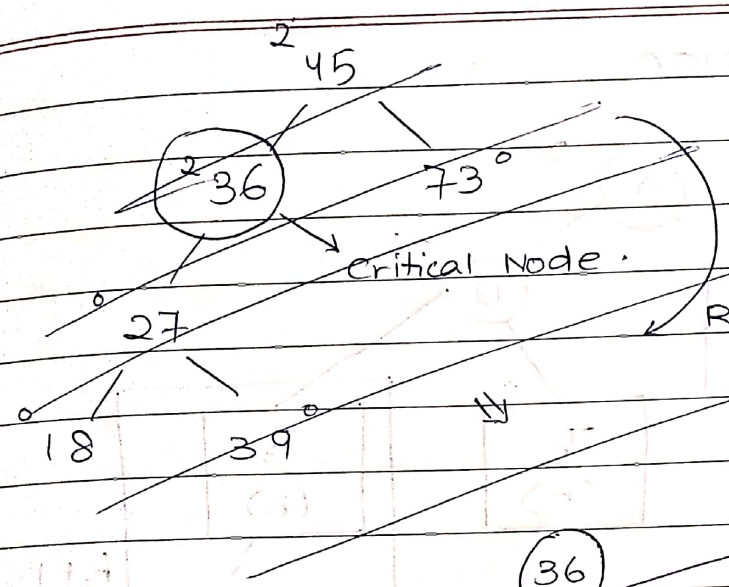


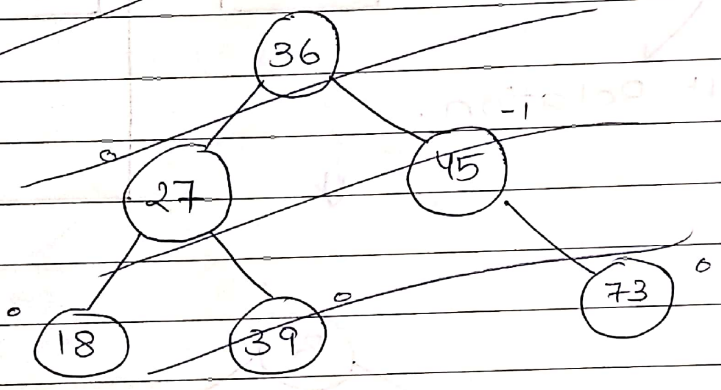
e.g.

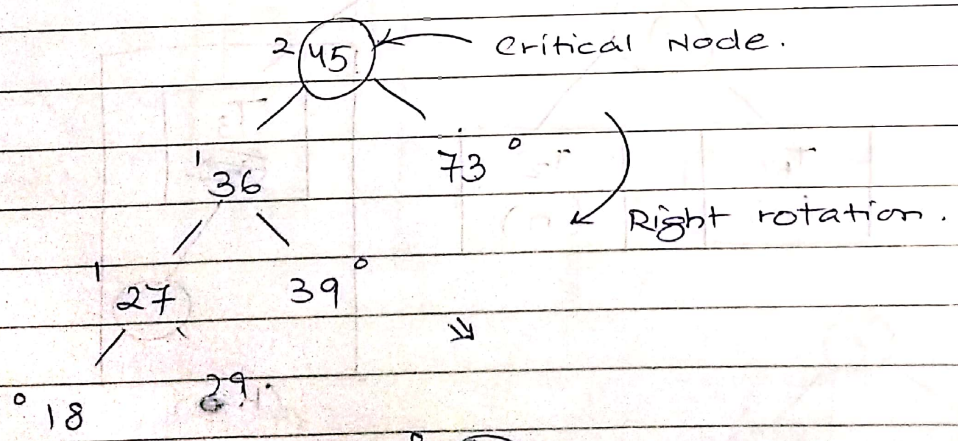


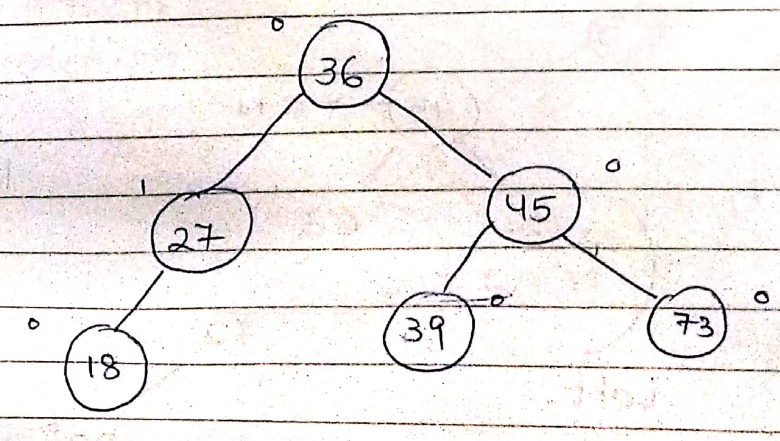
Calc. B.F of every node & insert 18. Show rotation.

→ Insert 18 :-

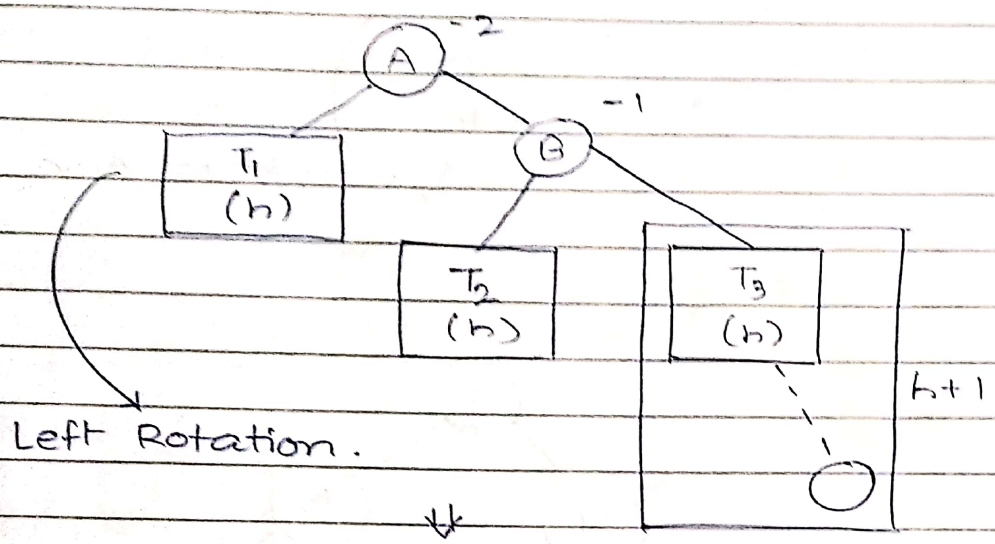

2
45
2
36
73
0
critical Node.
0
27
Right rotation.
0
18
39
0

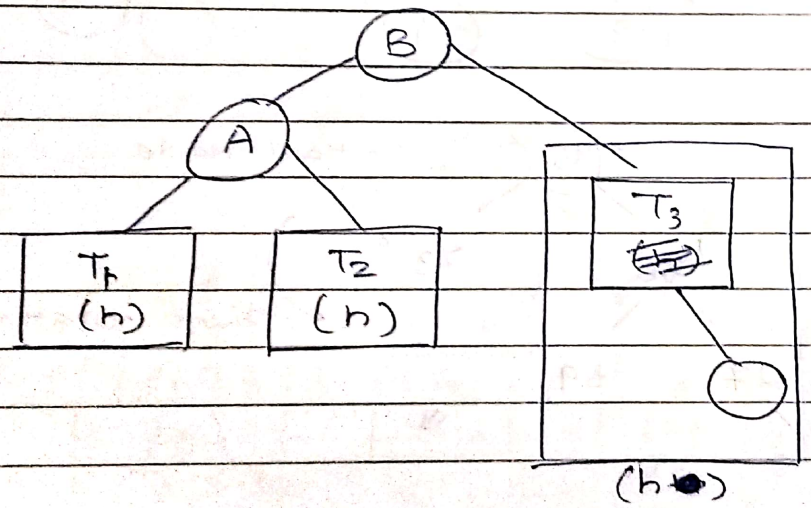

36
-1
0
27
45
0
73
0
18
39
0


2
45
critical Node.
1
36
73
0
Right rotation.
1
27
39
0
0
18
29

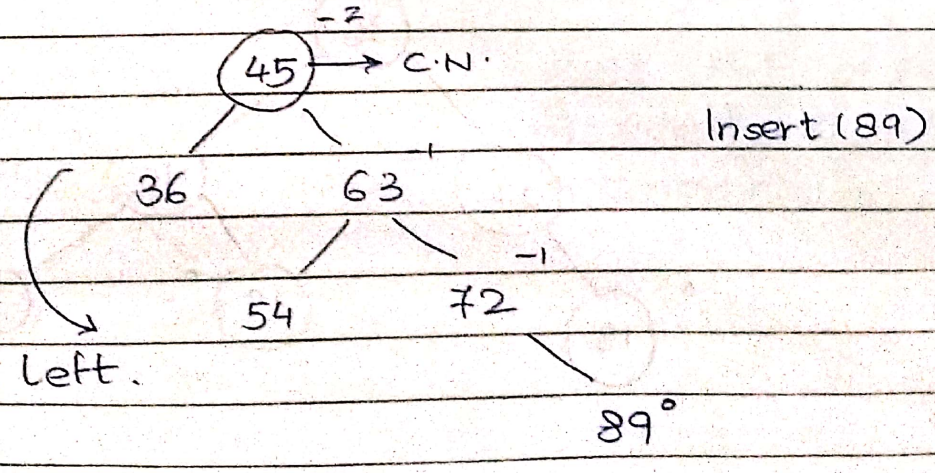

0
36
1
27
45
0
0
18
39
0
73
0

(ii) R.R insertion:

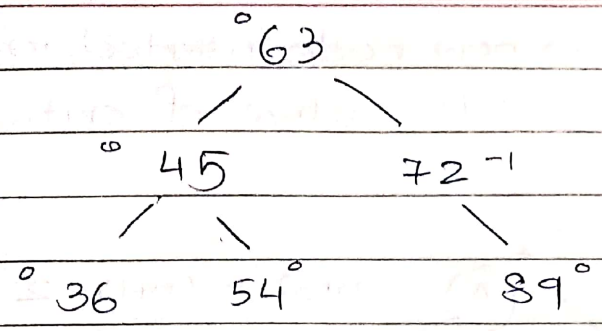




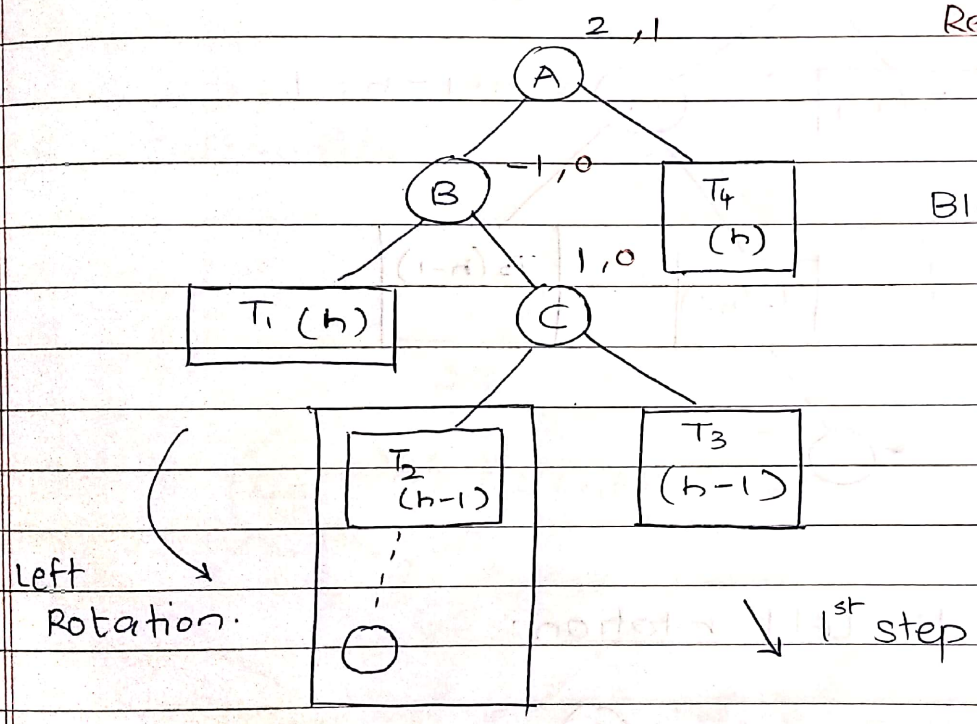
e.g.



Insert (89)



(iii) L·R insertion:

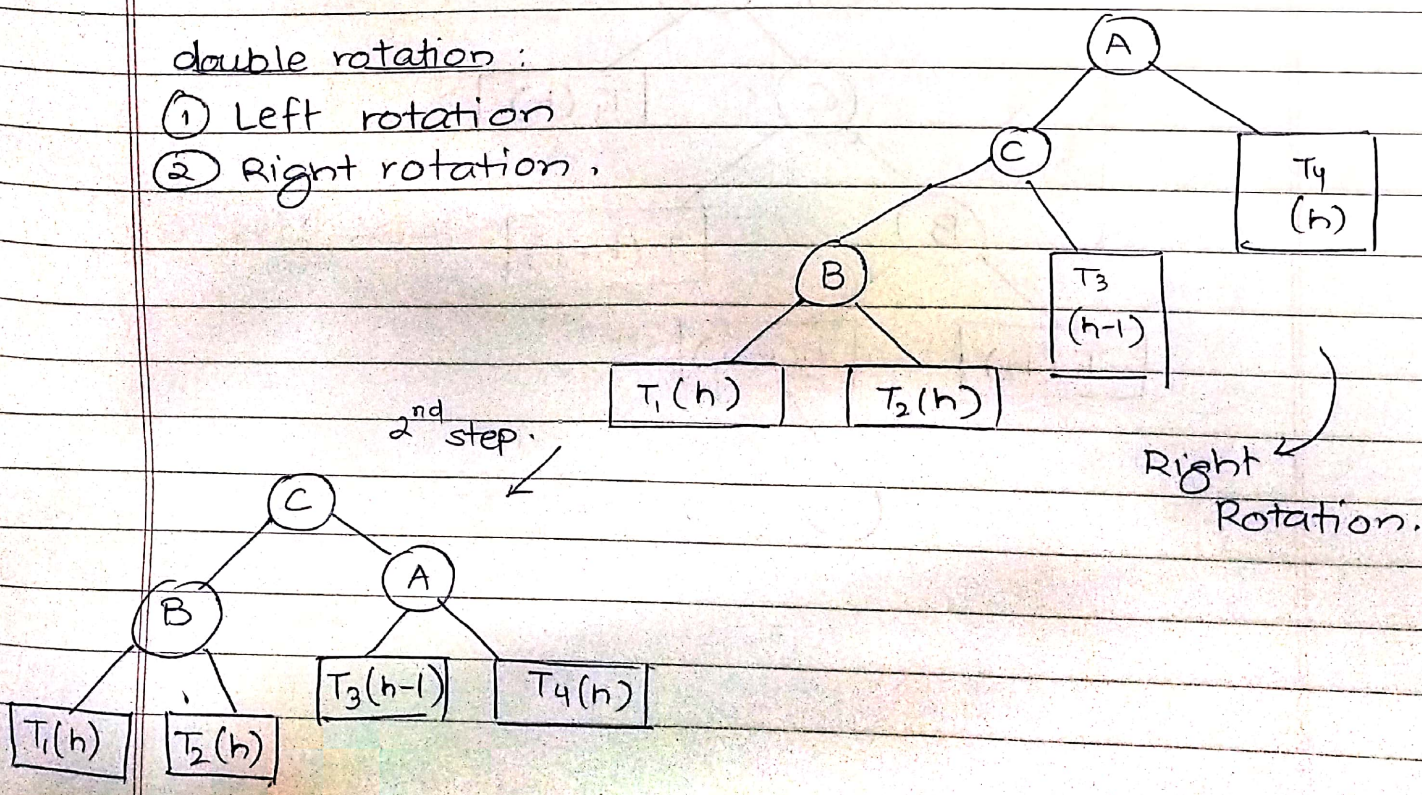


Red → Before insertion B.F.

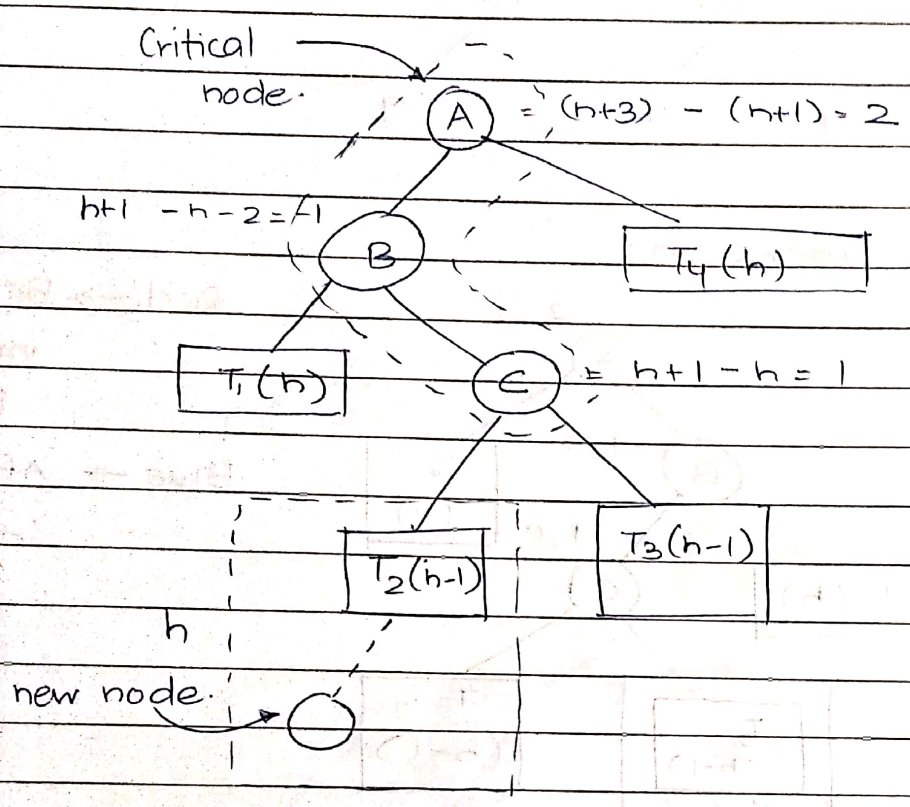
Blue → After insertion B.F.

double rotation:

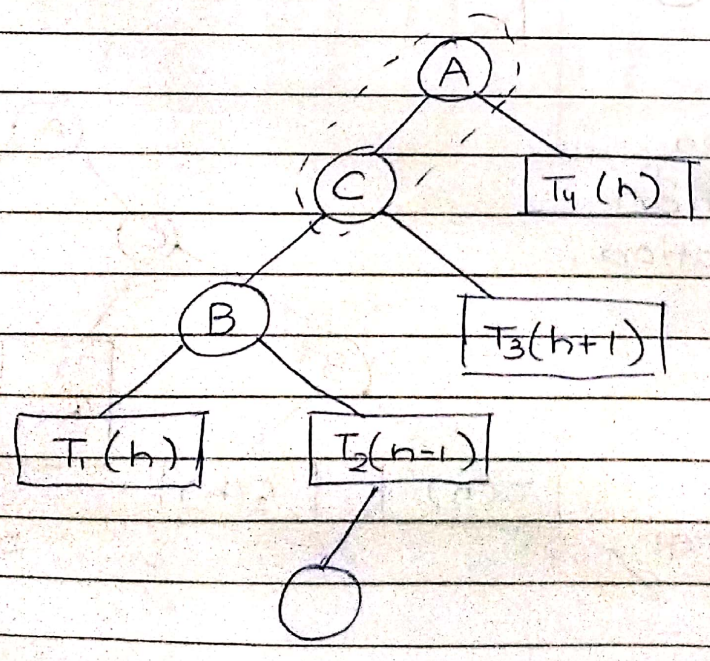
1. Left rotation
2. Right rotation.



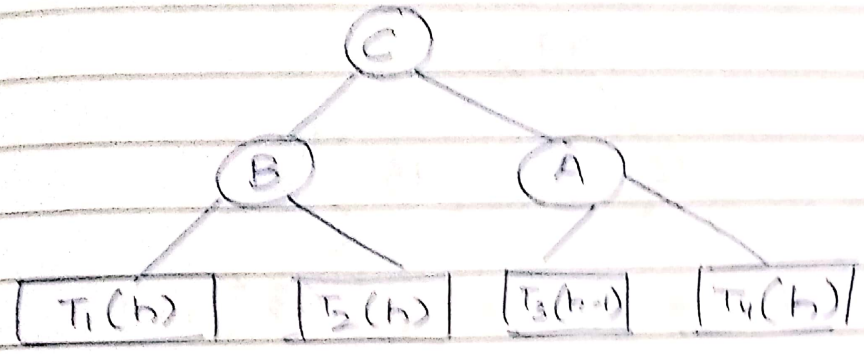
L·R insertion → new node inserted in Right subtree of left subtree of critical node.



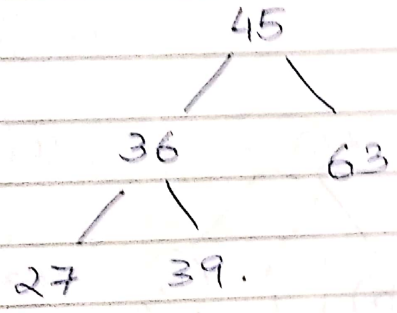
(i) <u>Step 1</u>:- <u>Left rotation</u>:-

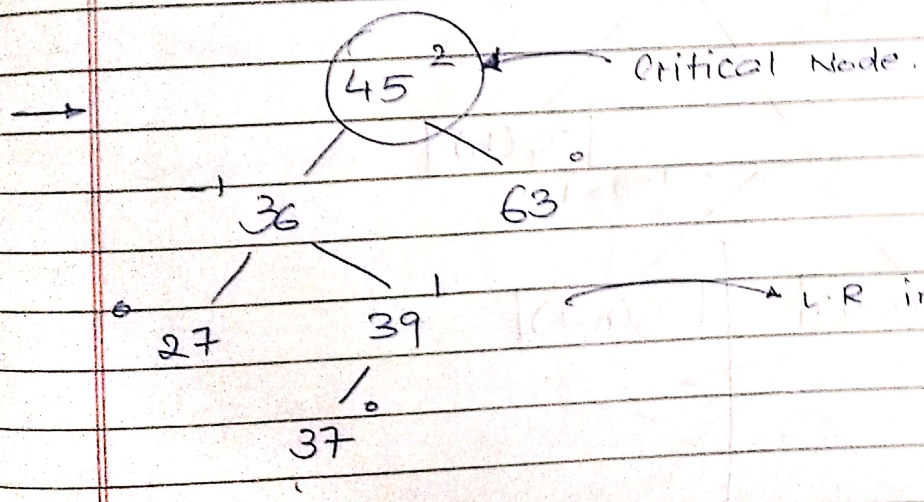


(ii) Step 2:- Right Rotation:-

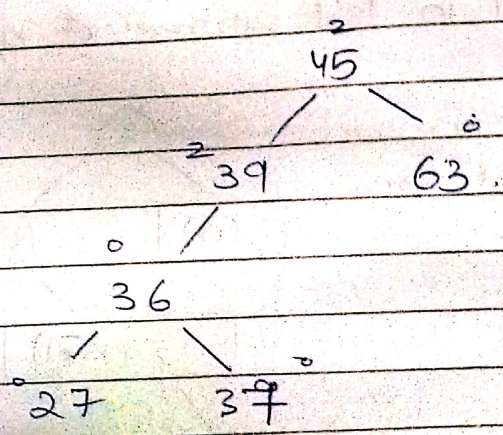


e.g. insert (37)

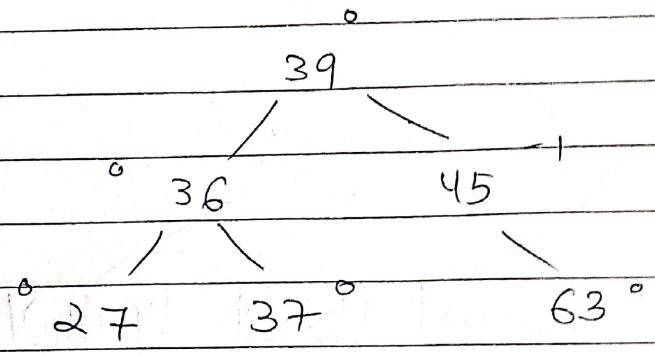




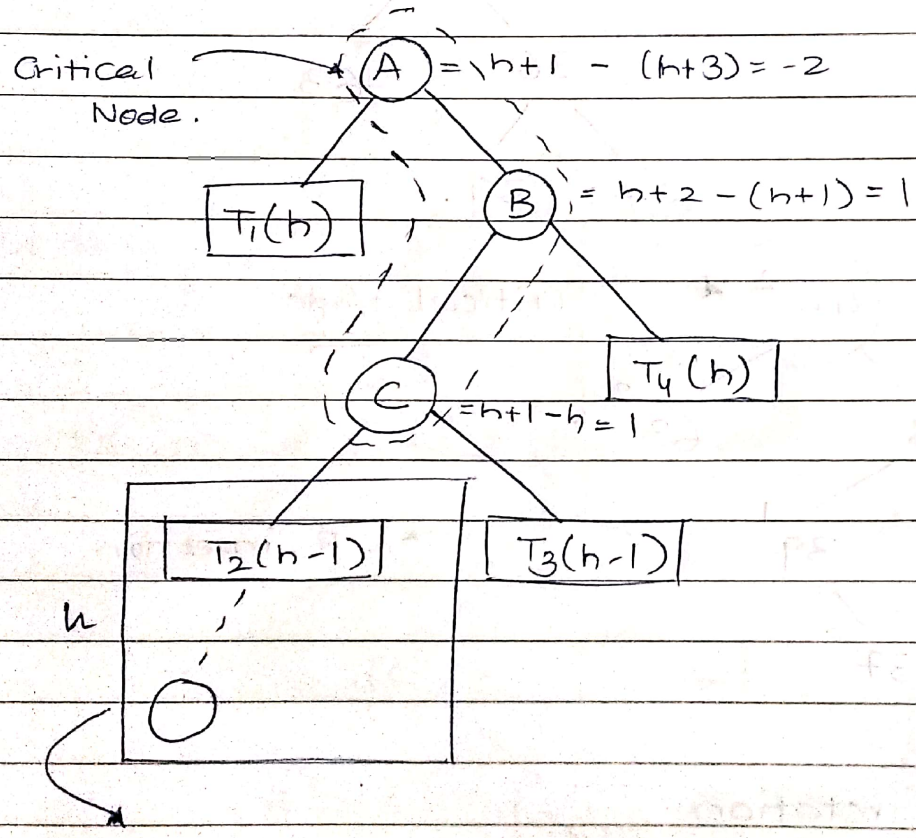
(i) Left rotation.



(ii) Right rotation.

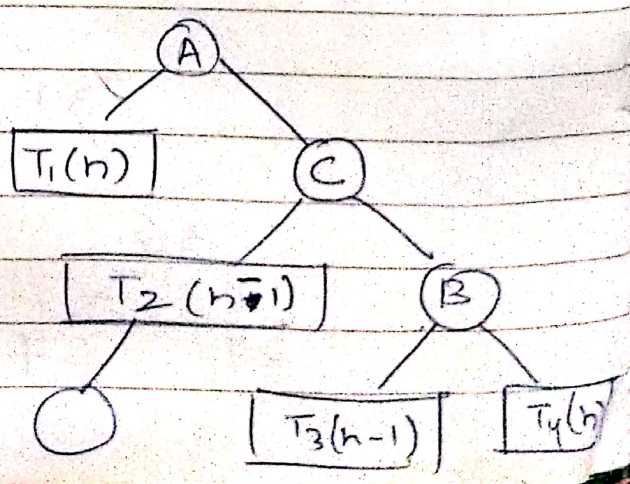


(iv) R.L Insertion:-

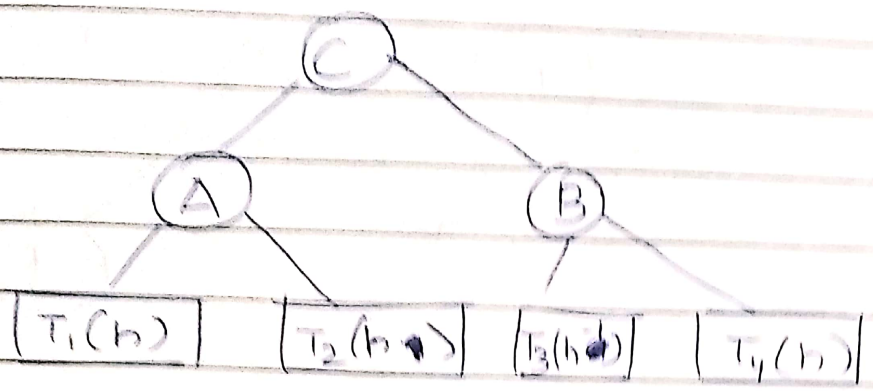


new node inserted in left subtree of right subtree of critical node.

Step (i) Right rotation:



step (ii) Left rotation:

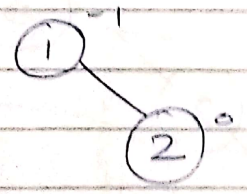


Q→ ~~Contruct~~ construct AVL tree for:

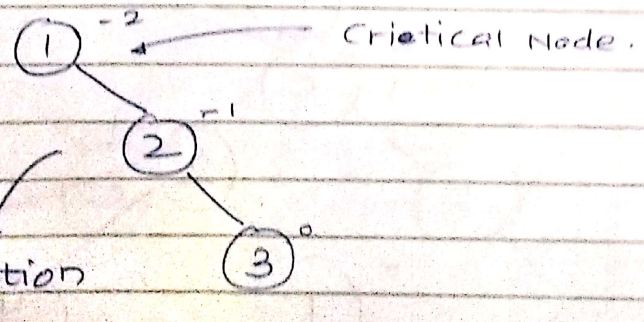
1, 2, 3, 4, 5, 6, 7, 8, 9, 10. show intermediate rotation at every step.

→ Step (i)

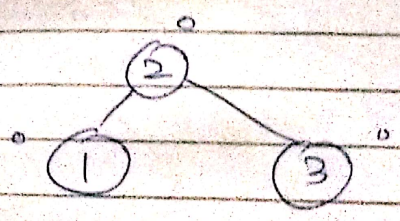
Step (ii)



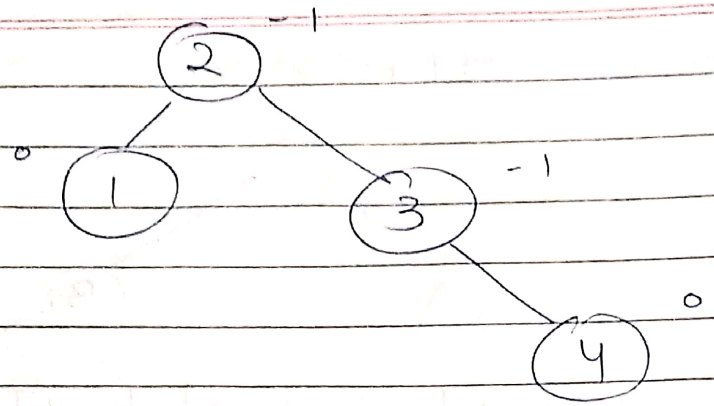
step (iii)



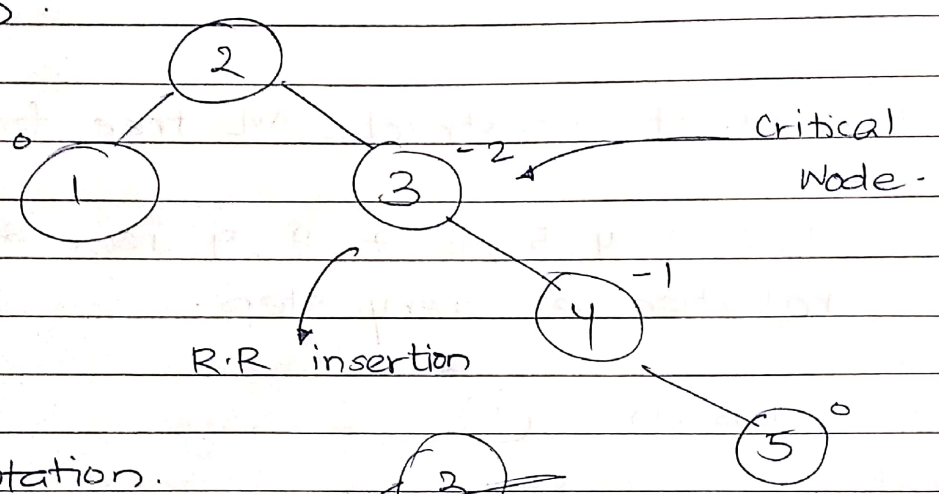
∴ Left rotation.



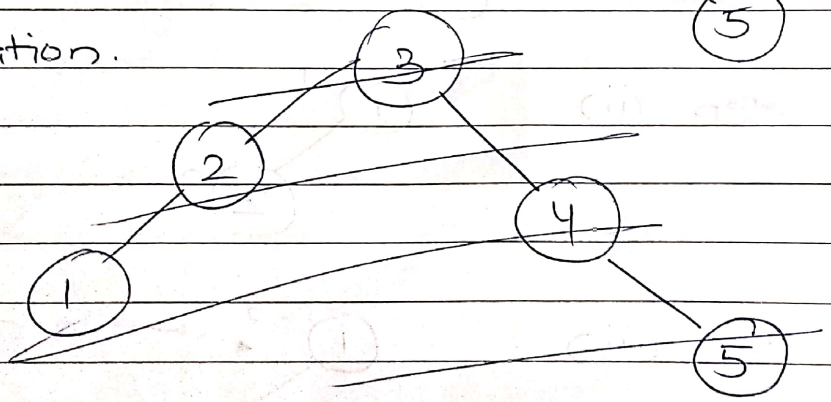
Step (iv)

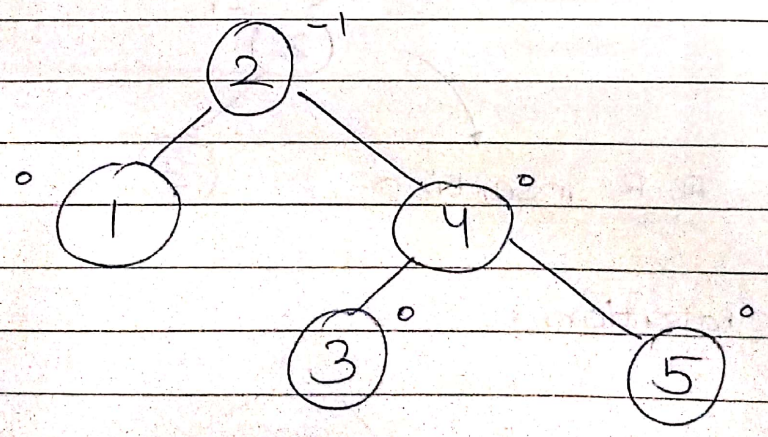


Step (v) :

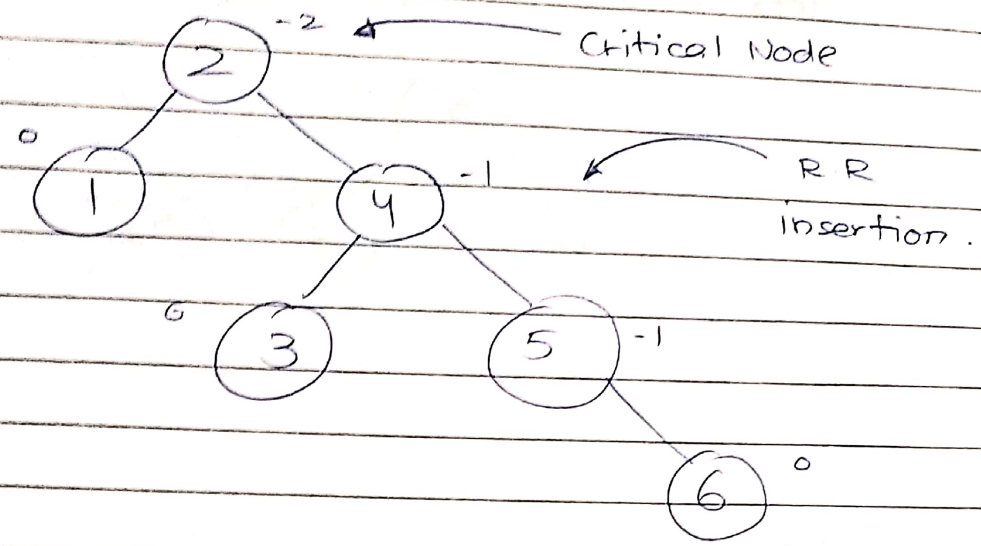


∴ Left rotation.

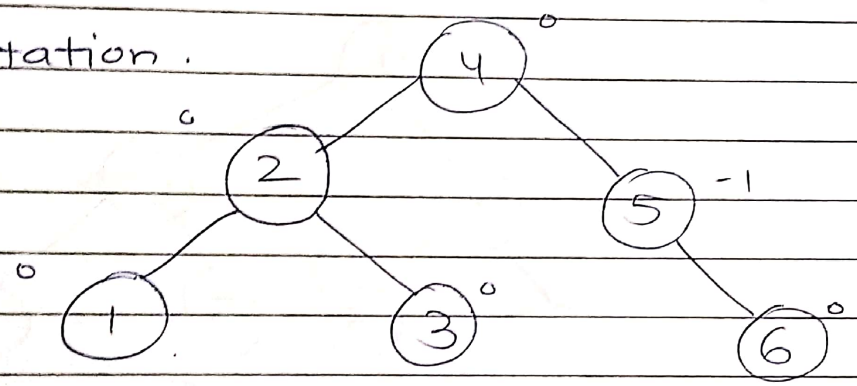




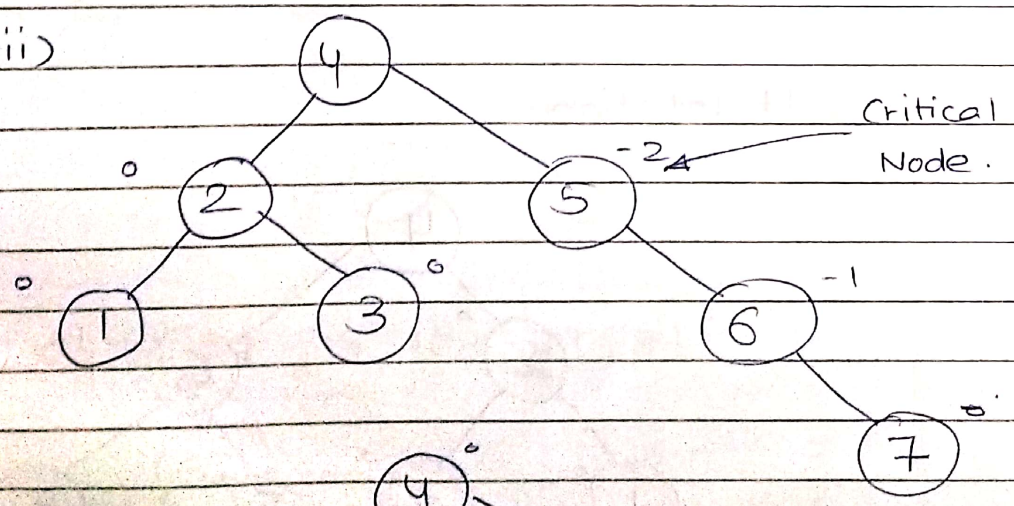
Step (vi)



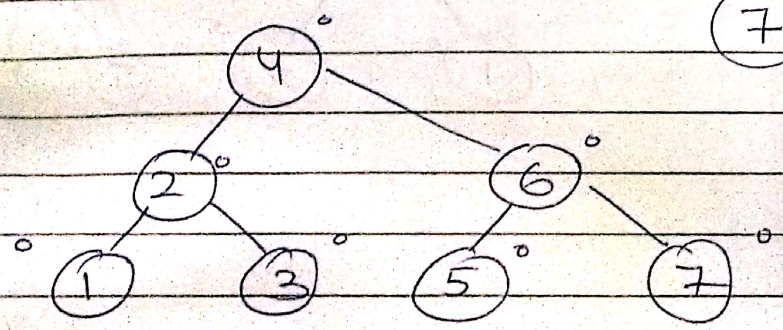
∴ Left rotation.



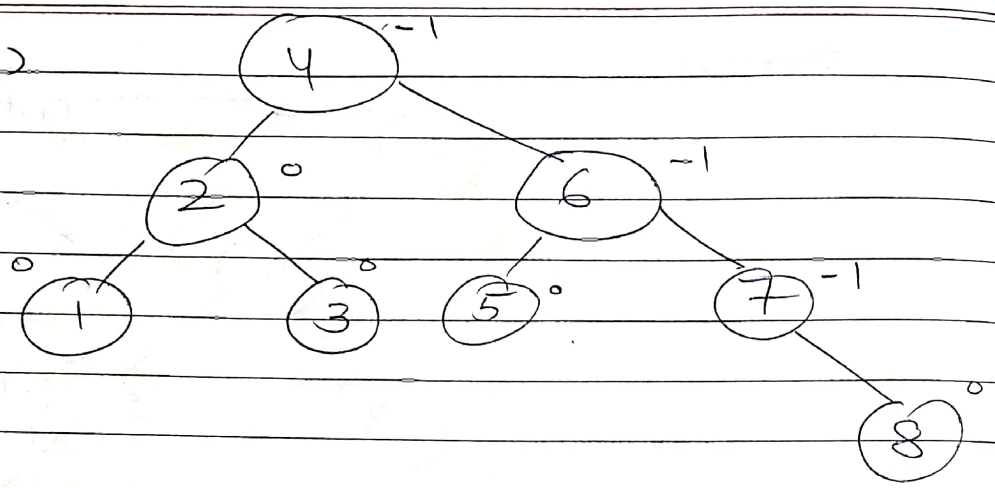
Step (vii)



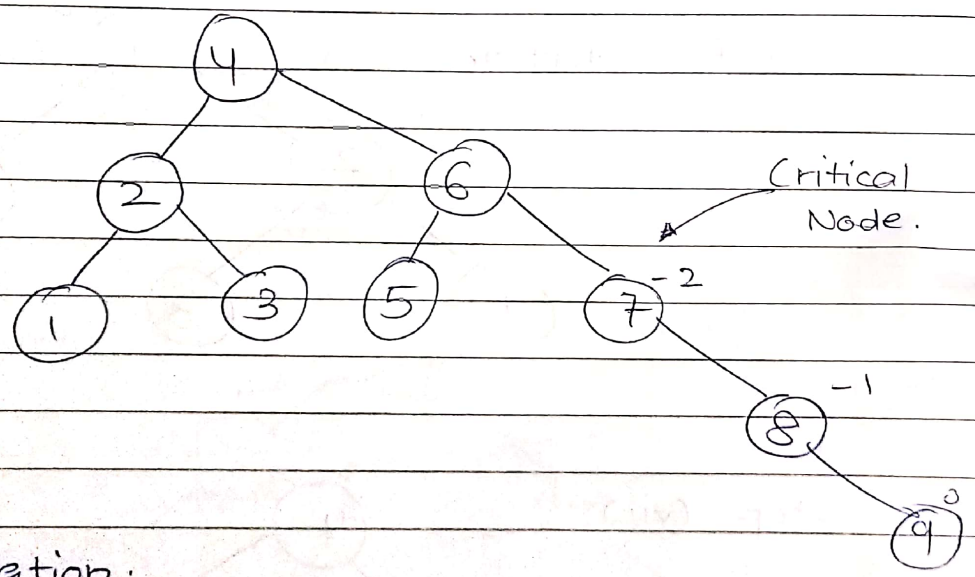
∴ Left rotation



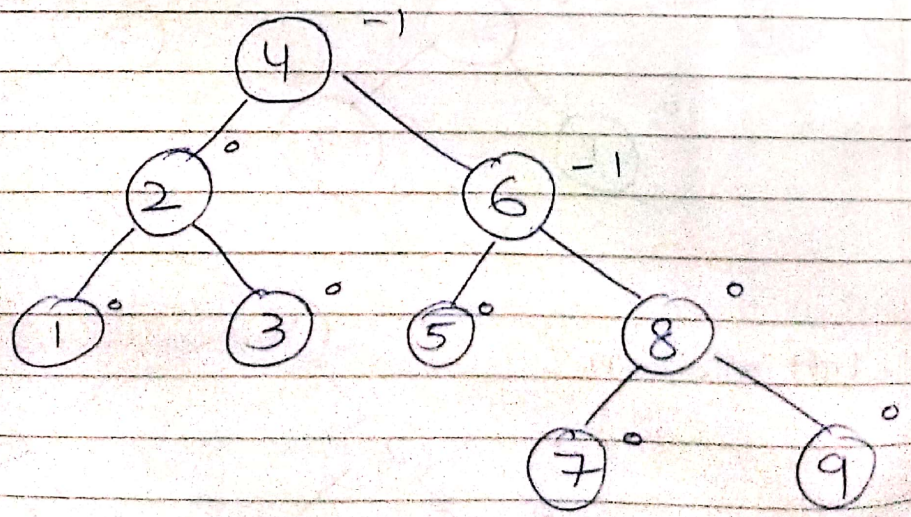
Step (viii)



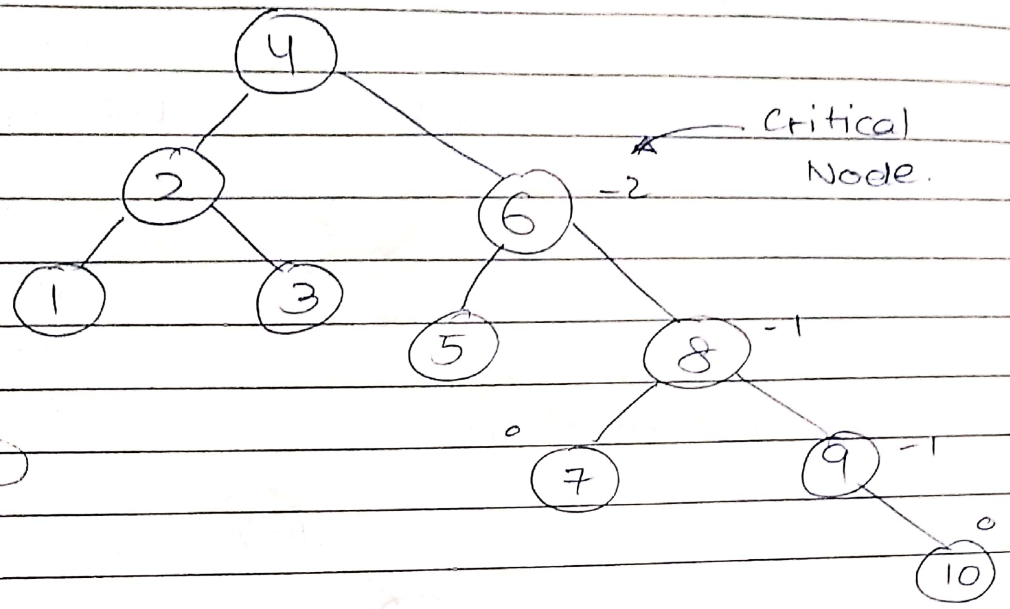
Step (ix)



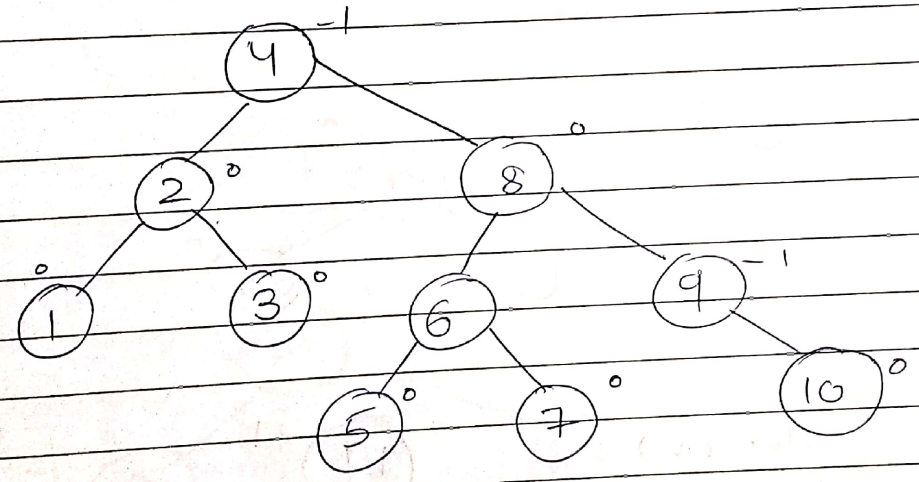
∴ Left rotation.



Step (x)

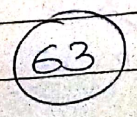


∴ Left rotation.

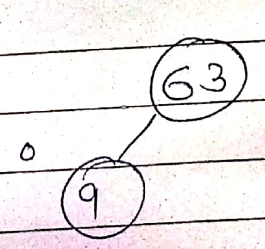


Q→ 63, 9, 19, 27, 18, 108, 99, 81.

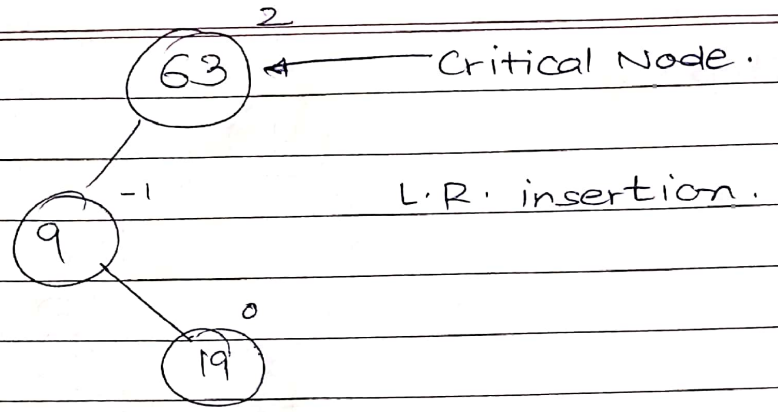
→ Step (i)



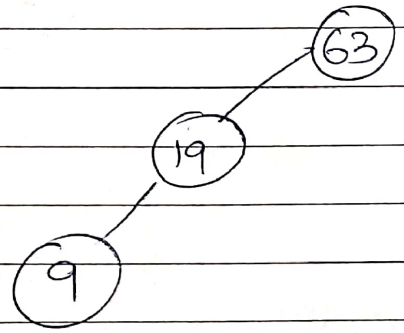
Step (ii)



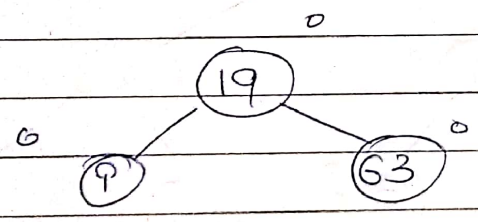
Step (iii)



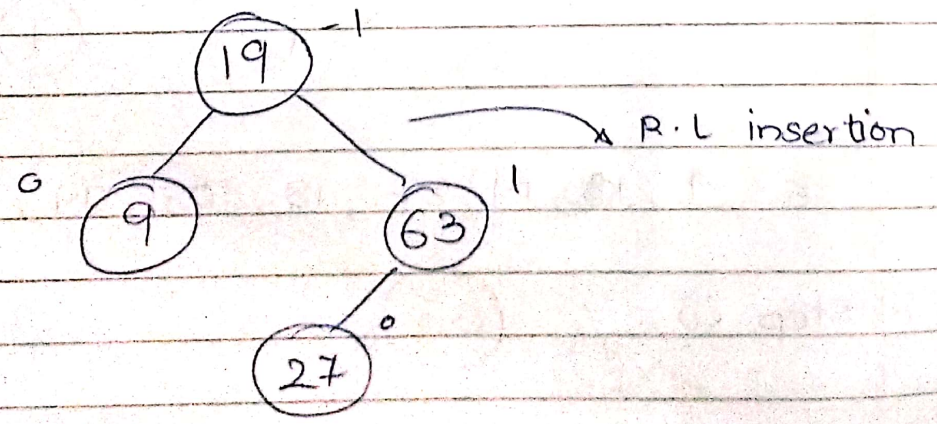
∴ Left rotation:



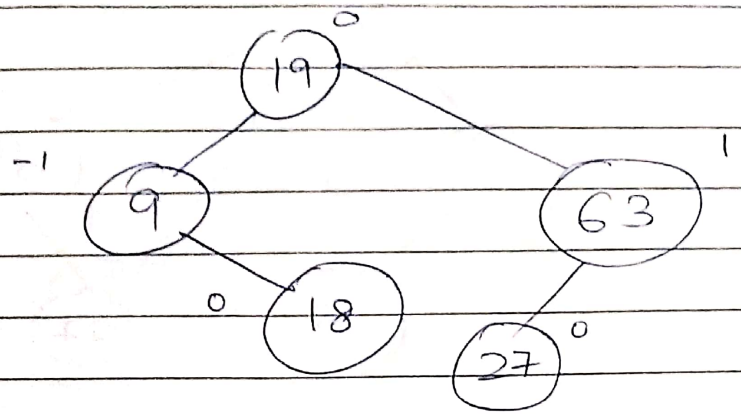
Right:



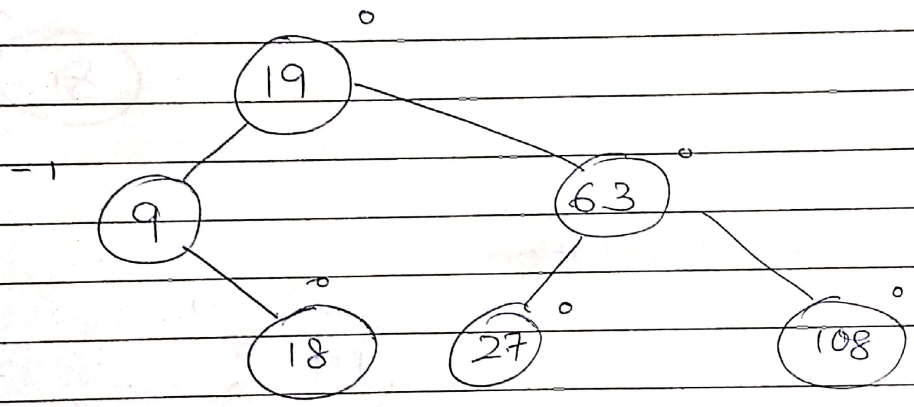
Step (iv)



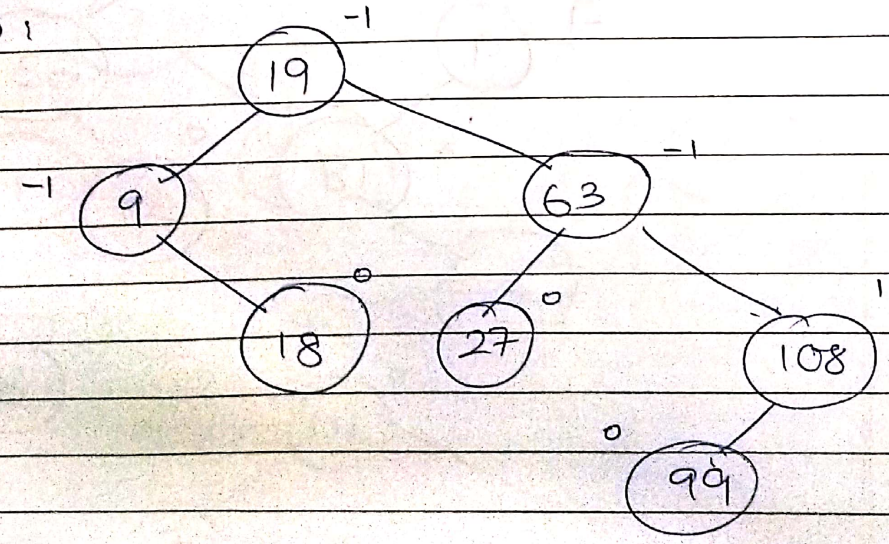
Step (v)



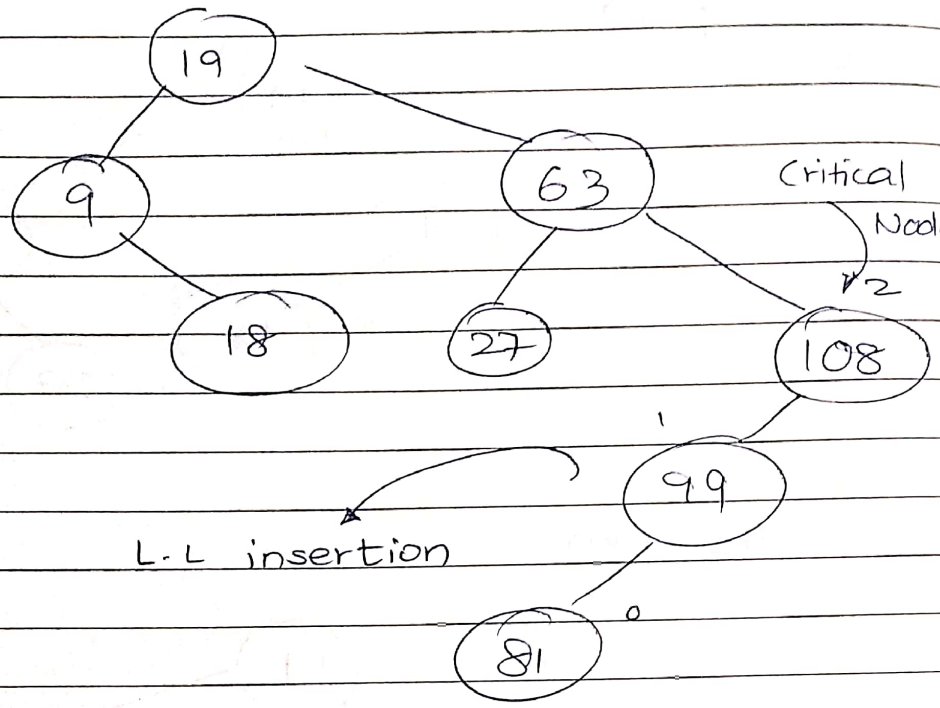
Step (vi) :



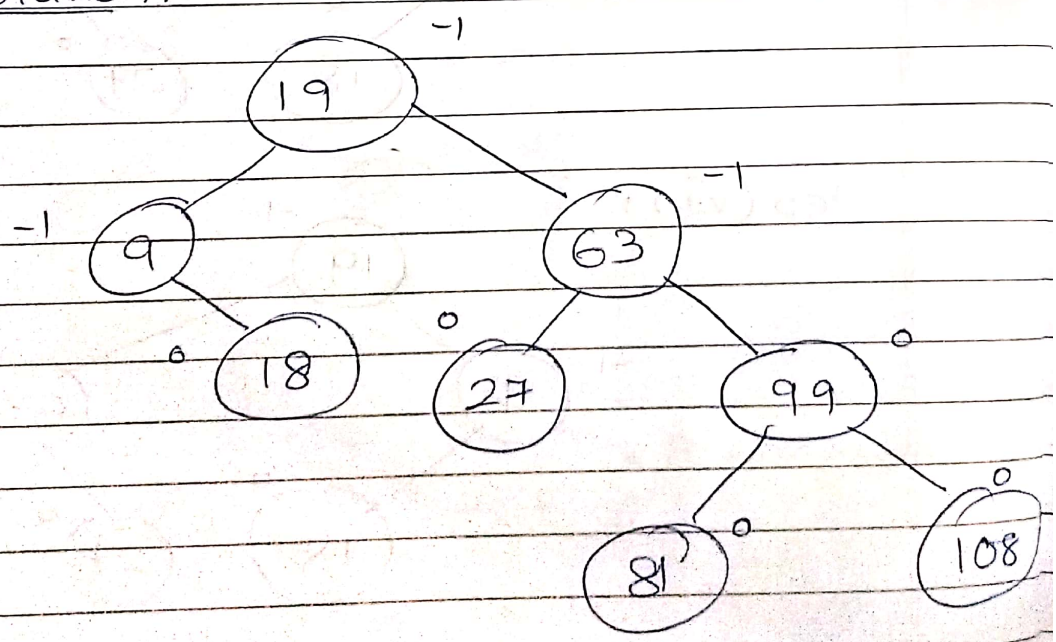
Step (vii) :



step (viii)

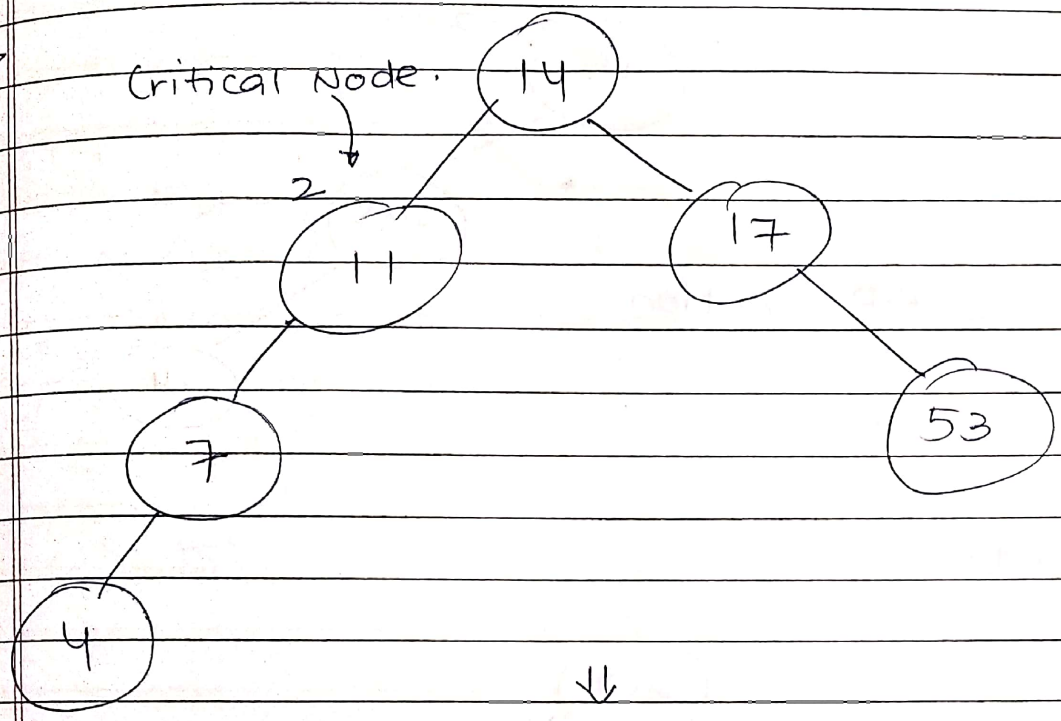


∴ Right rotation:

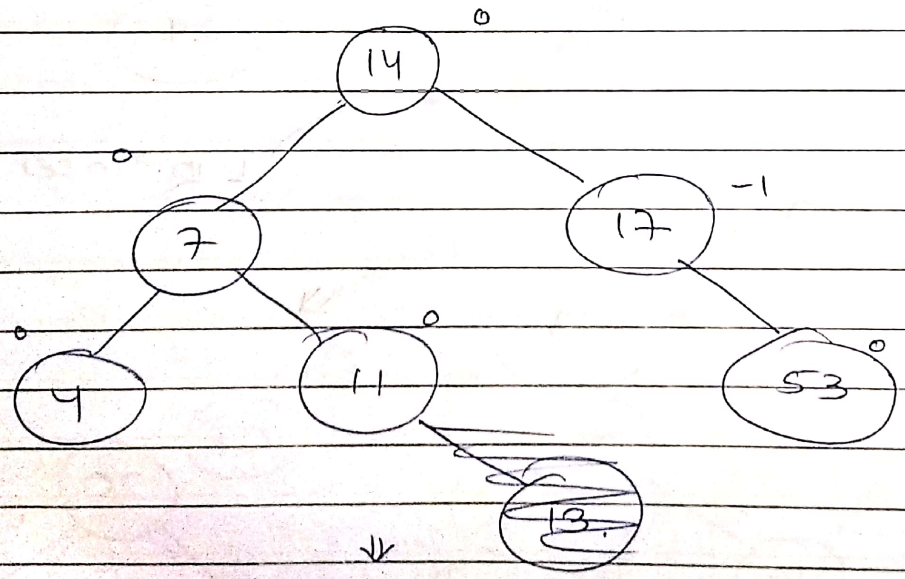


Q → 14, 17, 11, 7, 53, 4, 13.

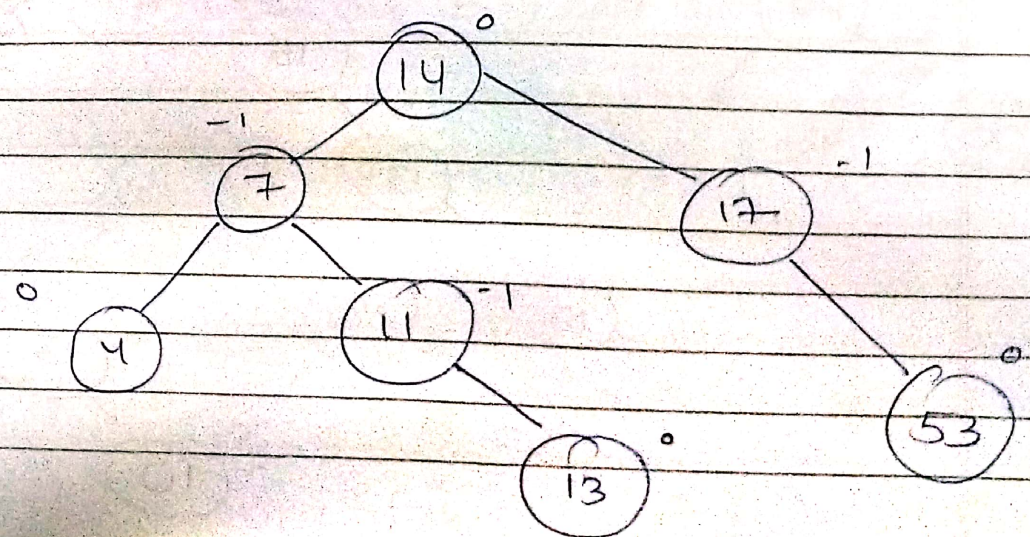
→



⇓

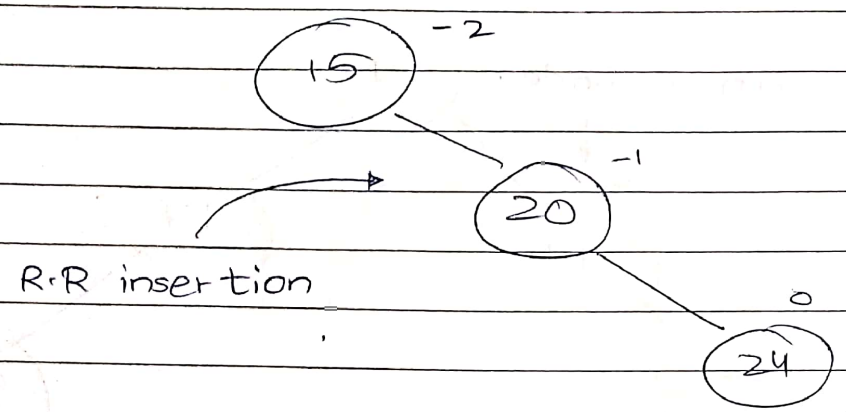


⇓

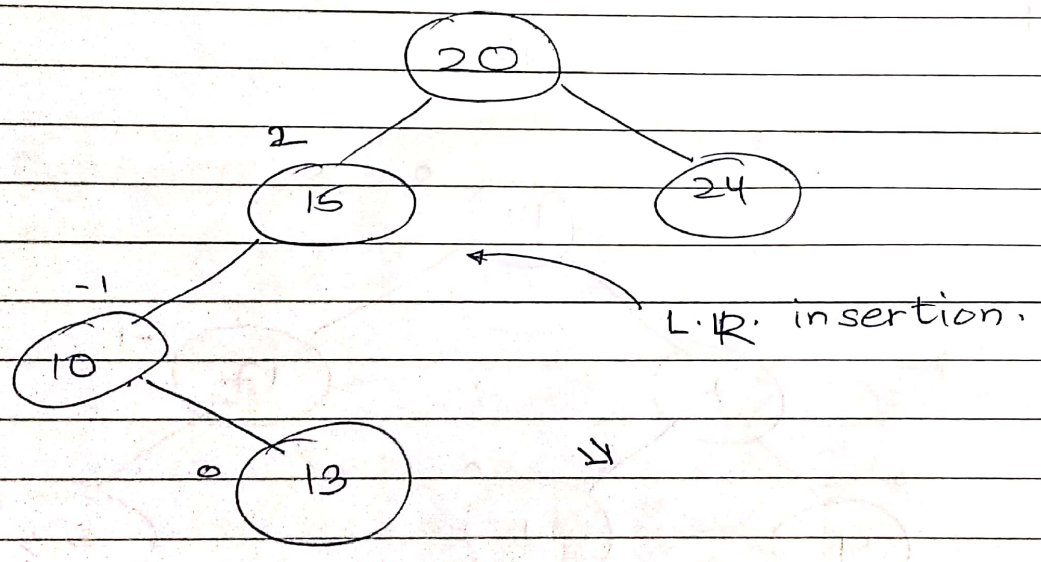


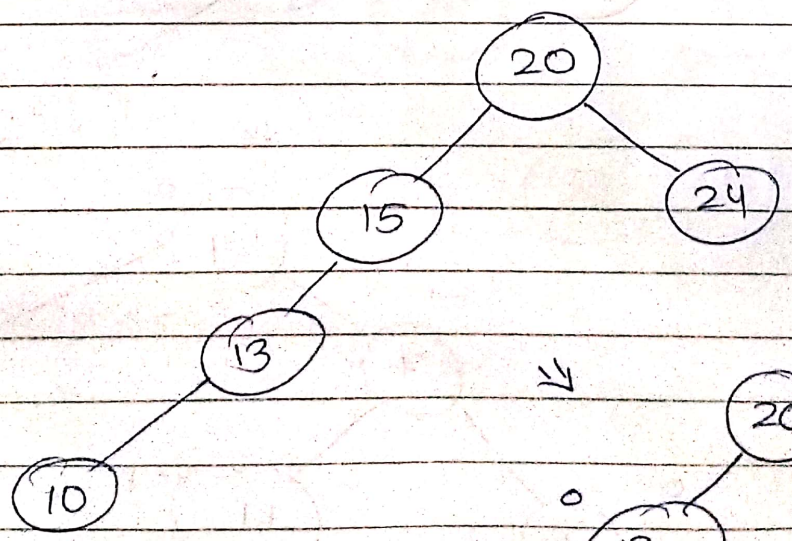
Q → 15, 20, 24, 10, 13, 7, 30, 36, 25.

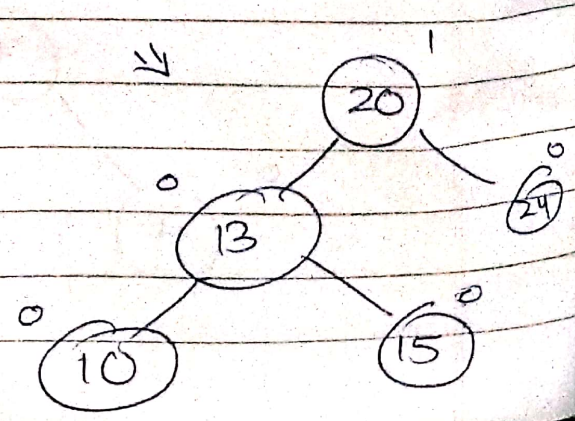
→

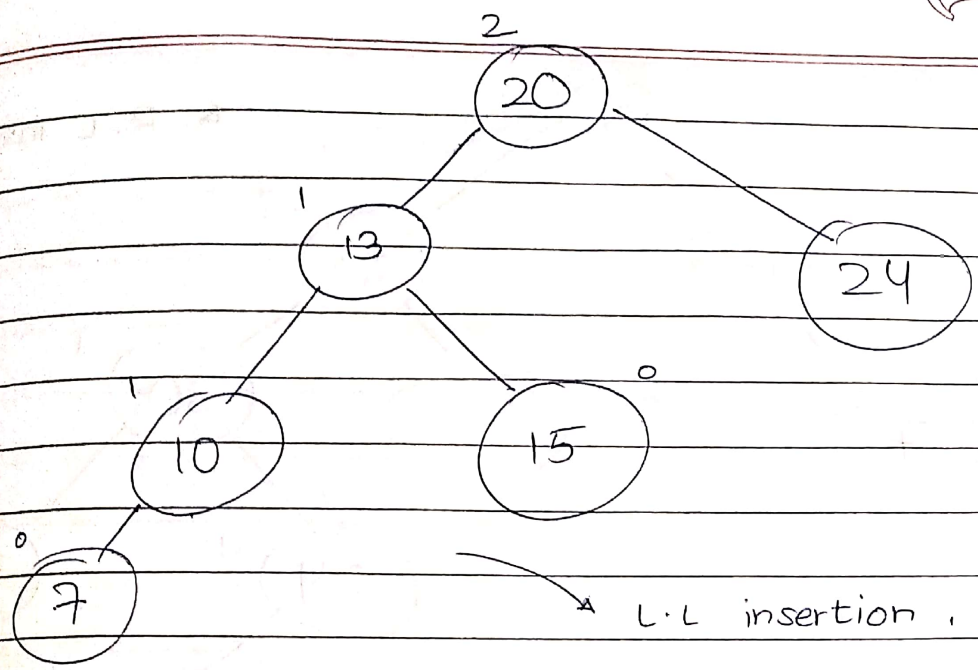


∴ L rot:



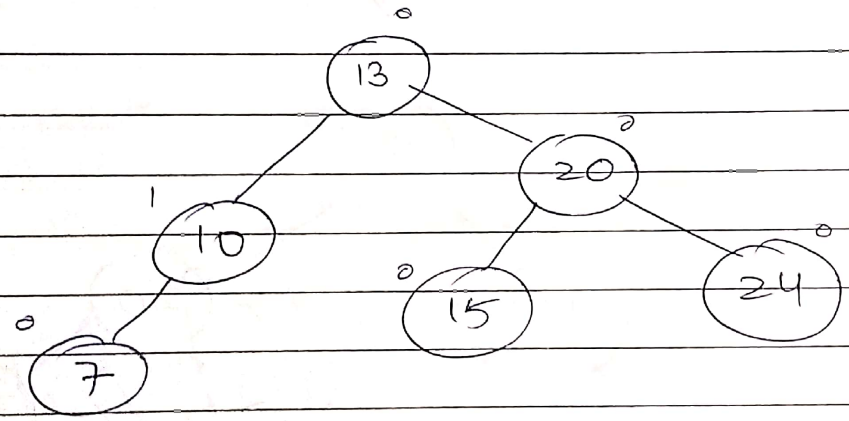


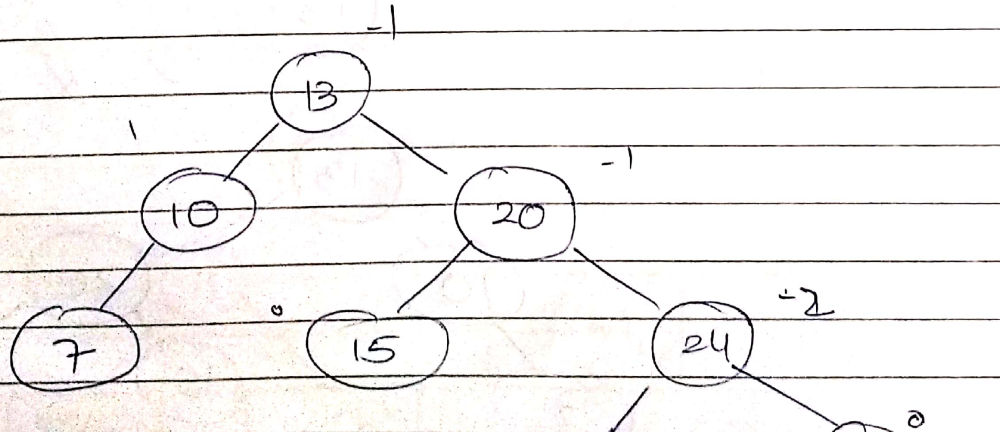




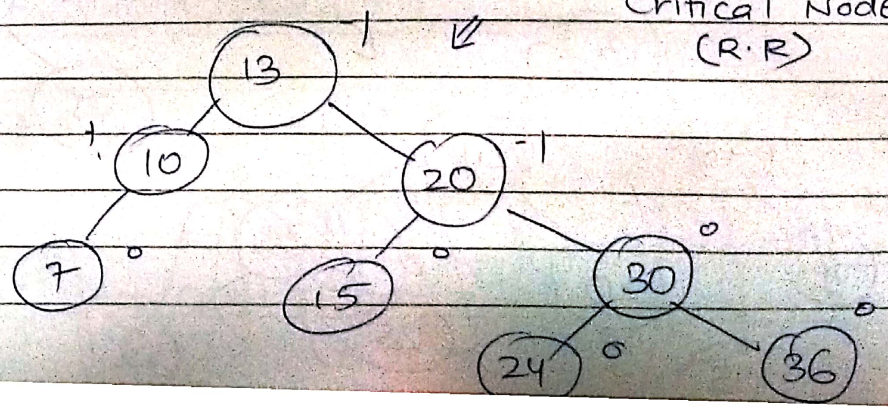
L·L insertion.

∴ R. rot:

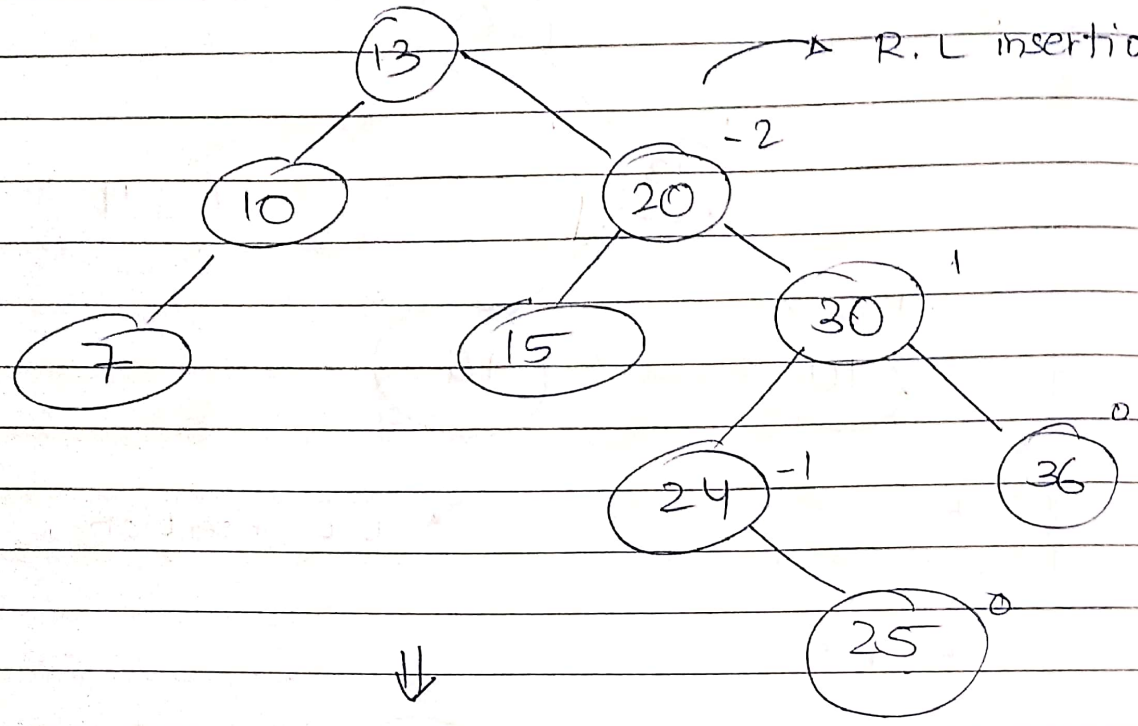


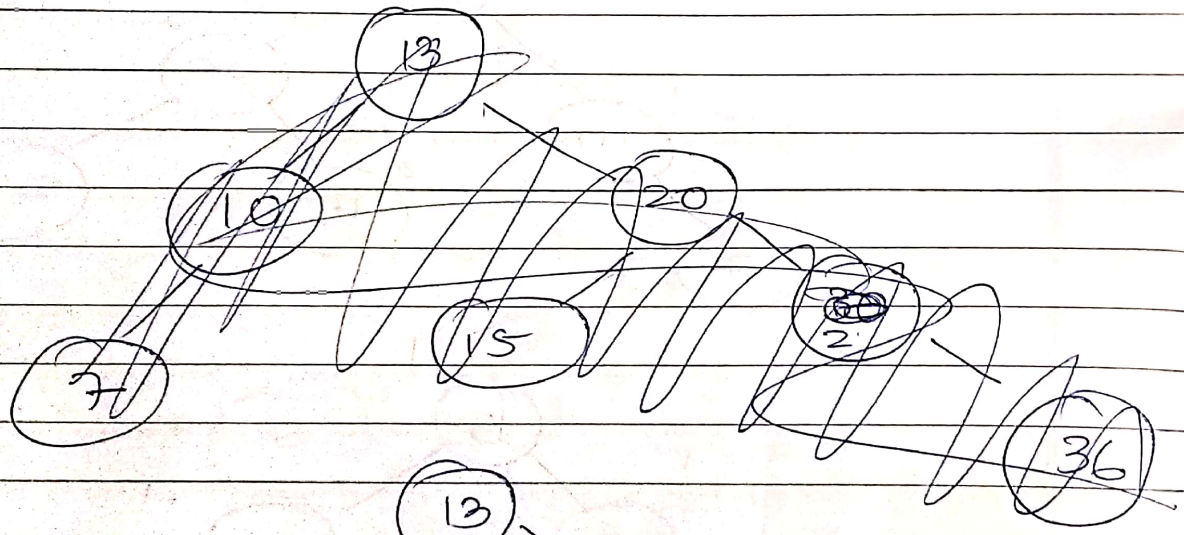


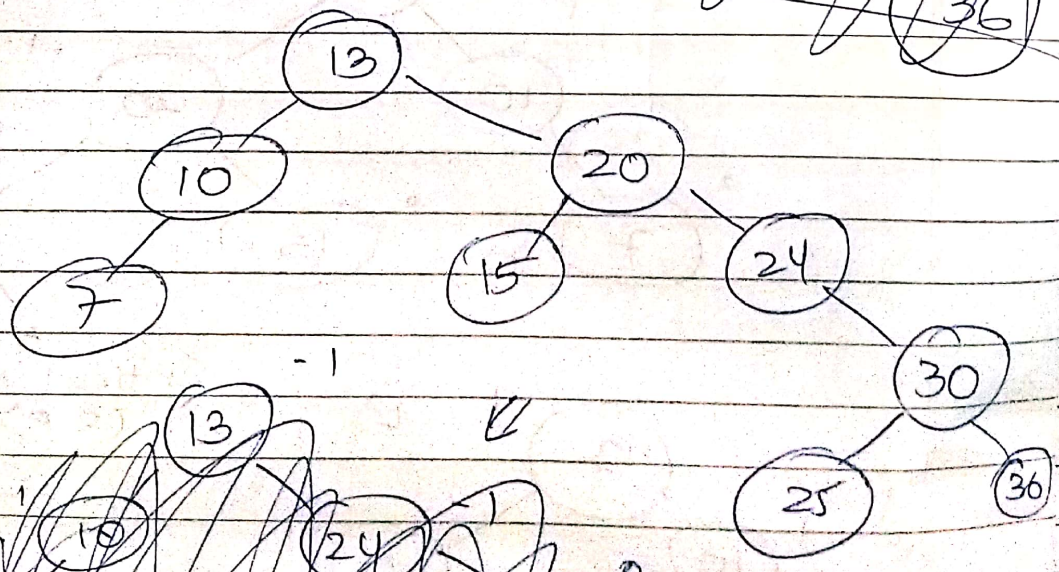
Critical Node.
(R·R)

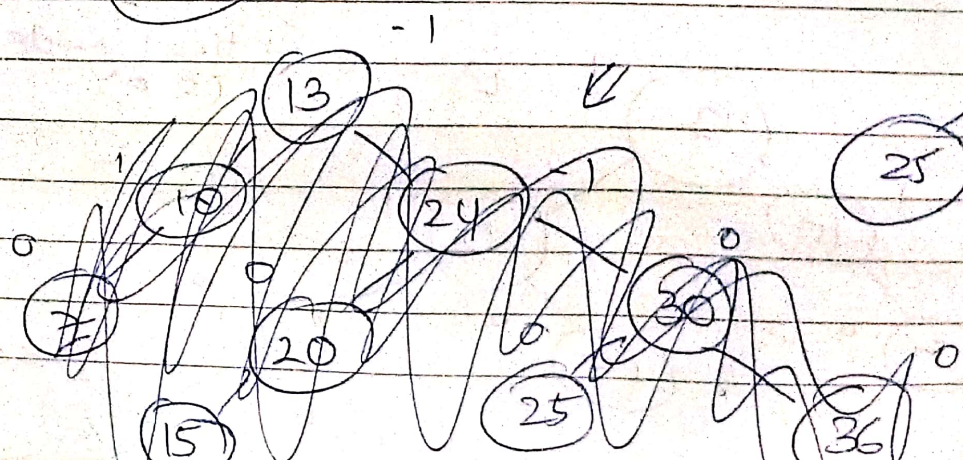


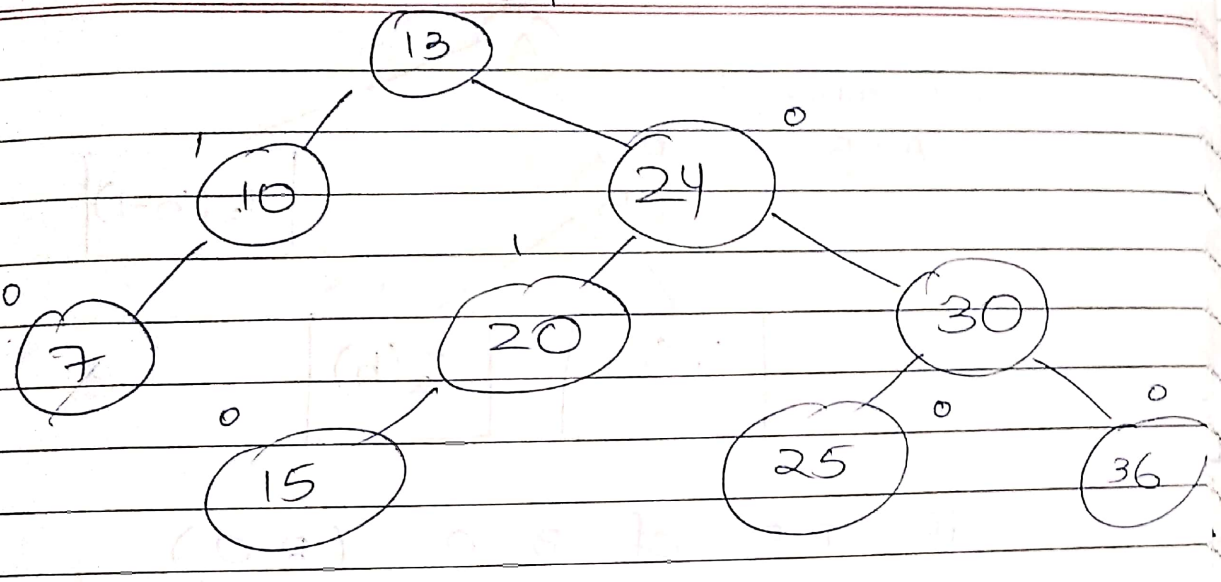
classmate
Date
Page











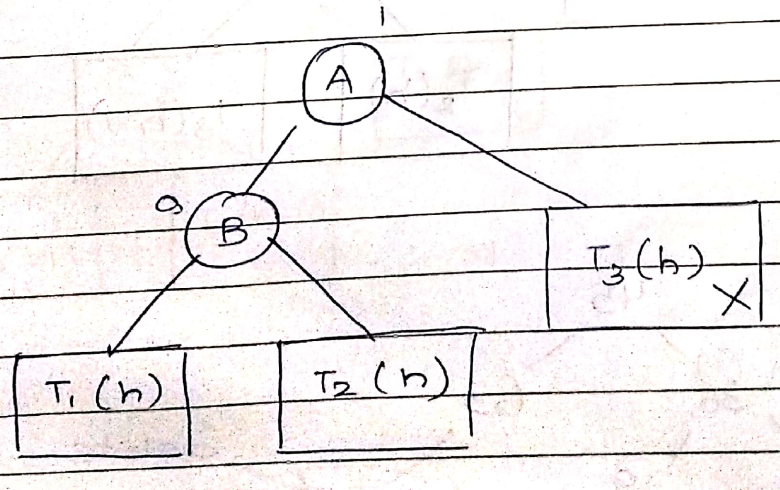
→ DELETION IN AVL TREE :-

If x is the node to be deleted.

Case (i): If x is in right subtree of C.N.

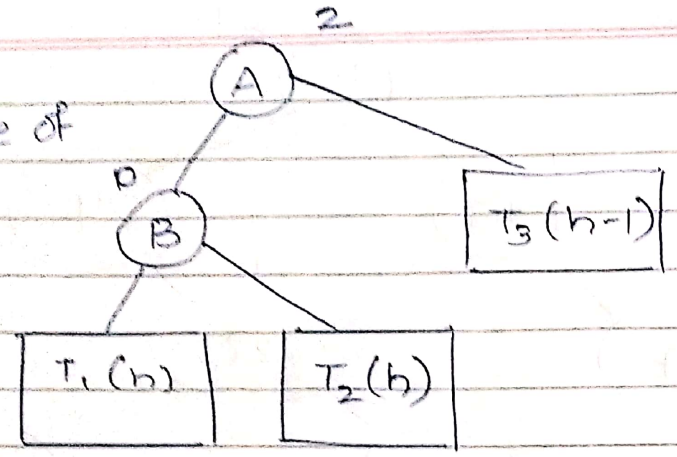
(a) R 0 (LL)
(b) R -1 (LR)
(c) R 1 (LL)

e.g.



AVL is a type of BST.
∴ Deletion will be according to deletion in BST itself.
After these steps, if there is a C.N., then follow next steps:

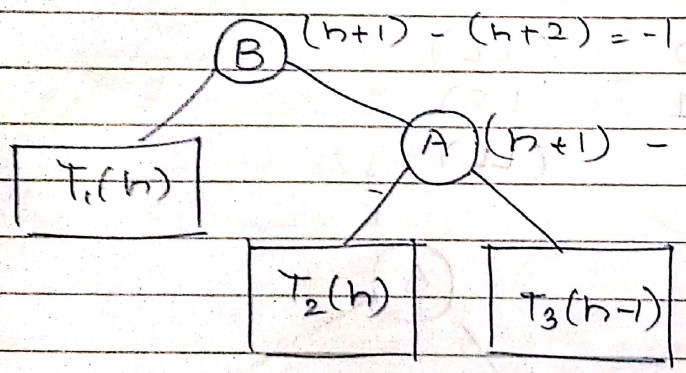
Left subtree of A = B.



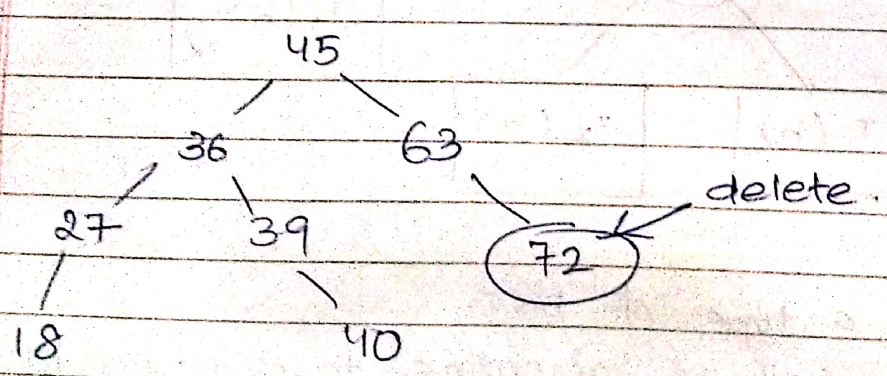
if B.F. of B = 0 (R O)

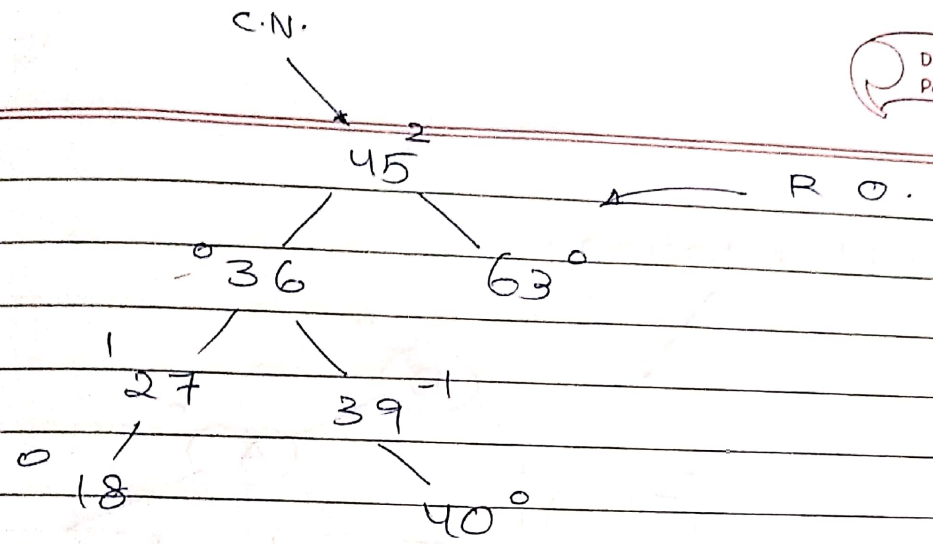
↳ then it is same like L.L insertion with A

R O . (Right rotation)

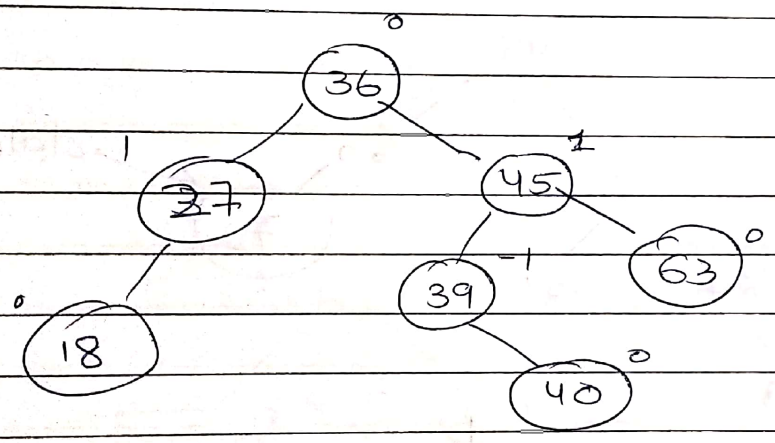


e.g.

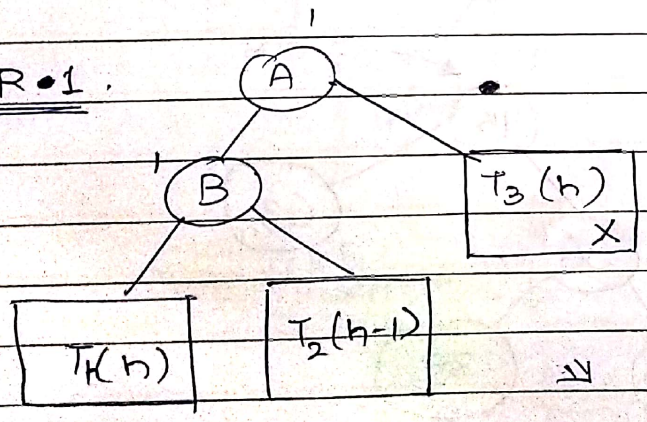




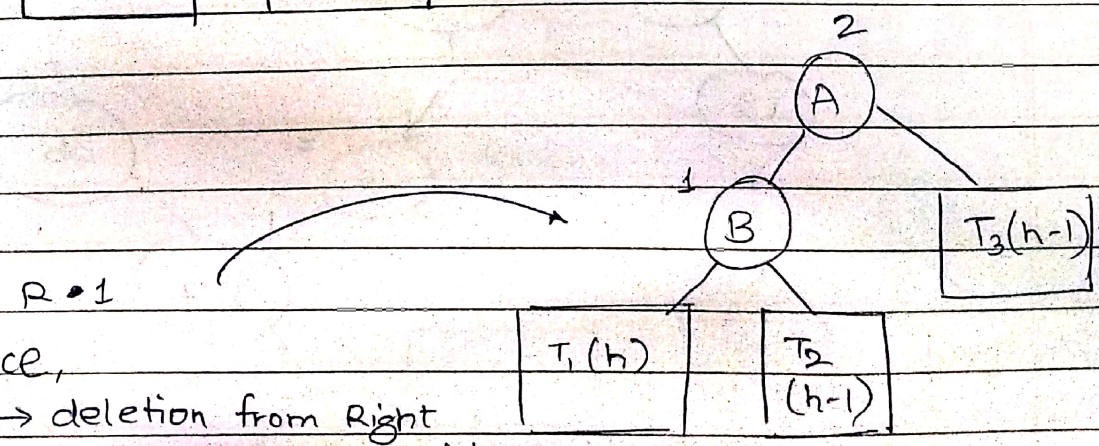
↓ Right rotation.



(b) R•1.



⇓



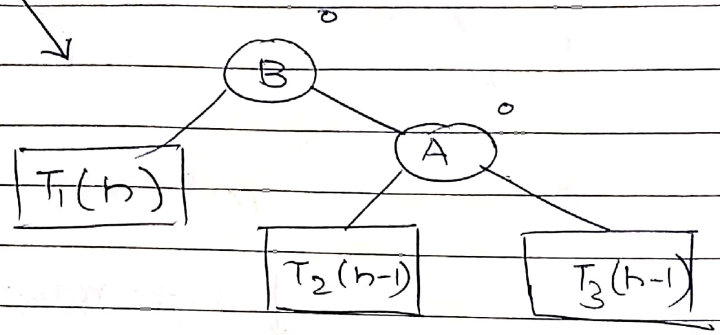
R•1

Since,

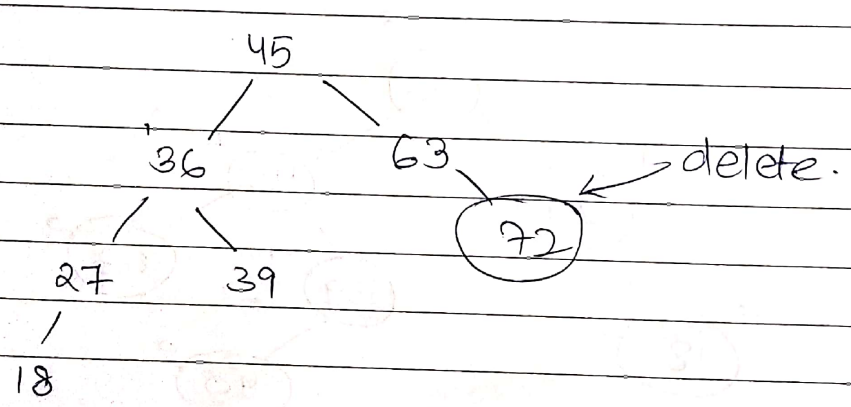
R → deletion from Right subtree

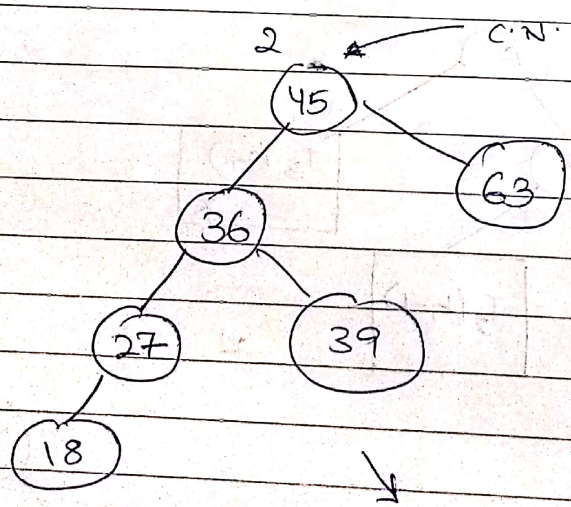
1 → B.F of B.

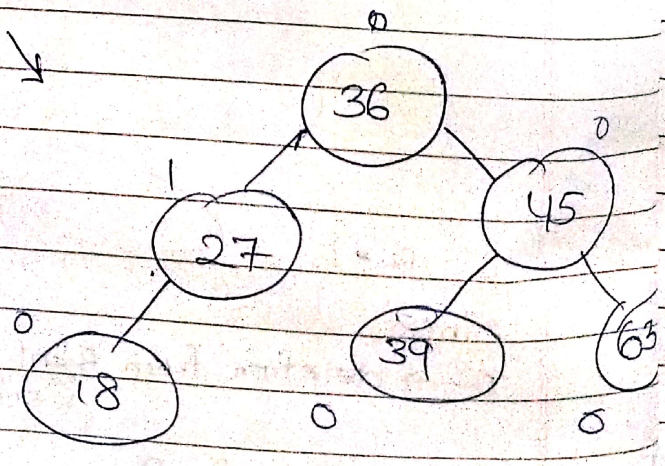
∴ right rotation.



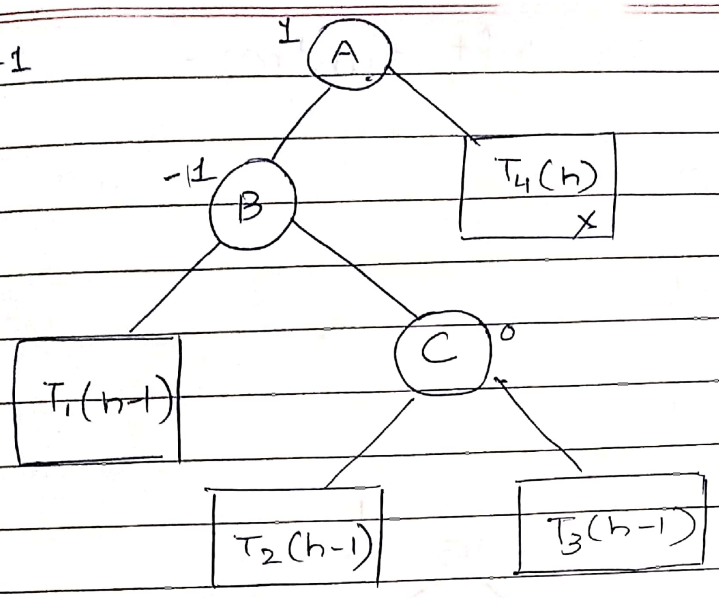
e.g.

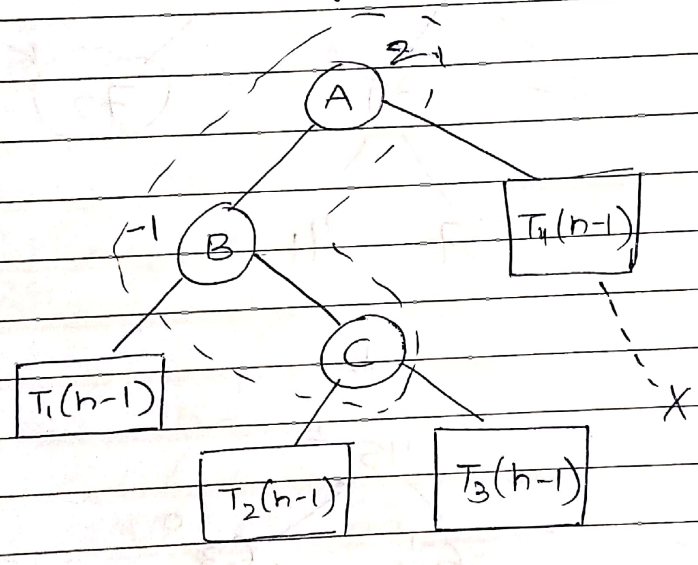






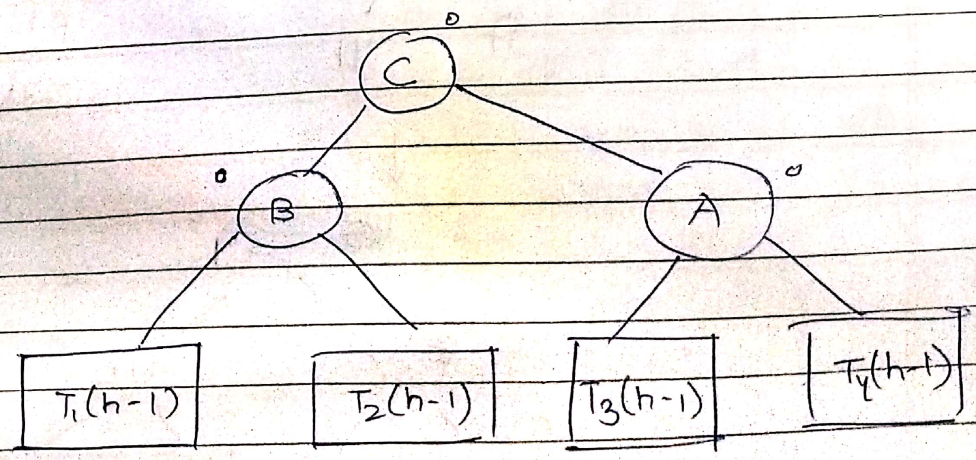
(c) R -1





L·R insertion.

R -1



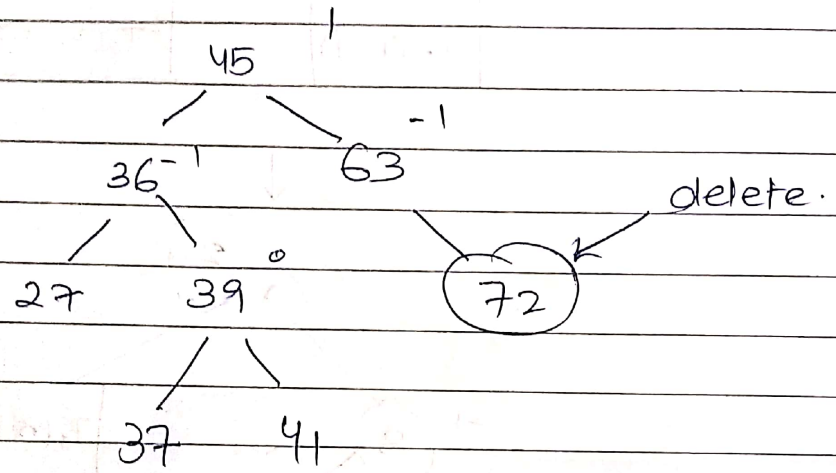
If x is in left subtree :-

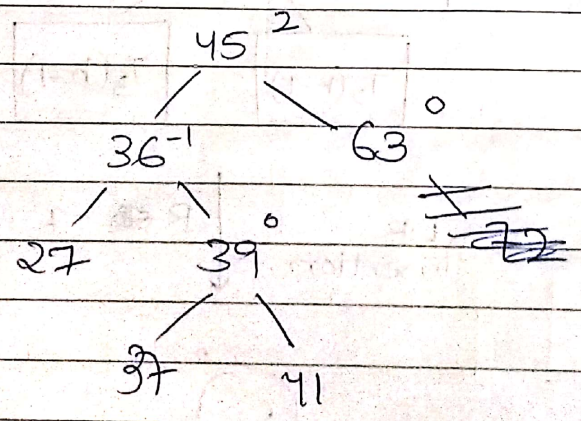
(a) L 0

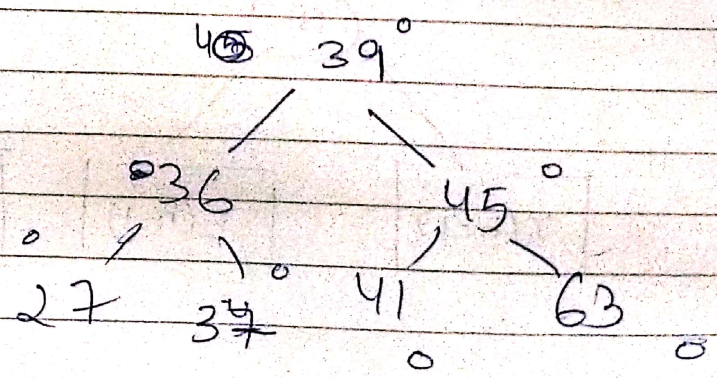
(b) L -1

(c) L 1

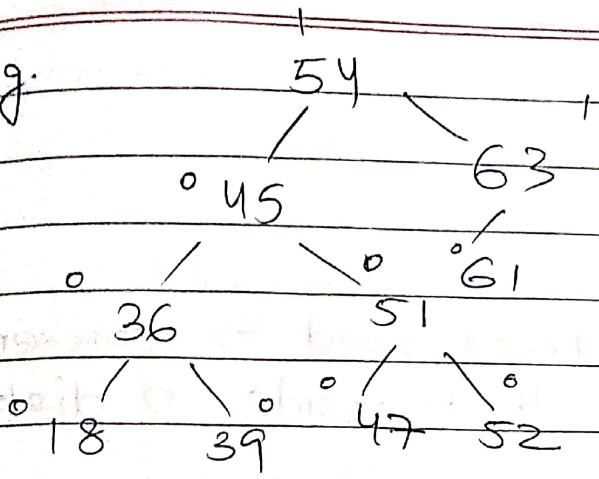
e.g.







e.g.



delete

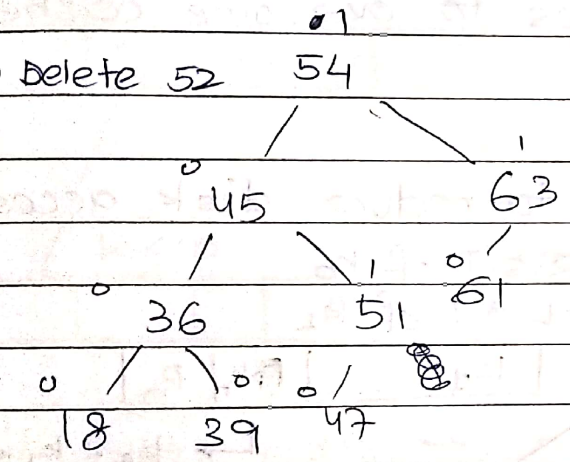
52

36

61

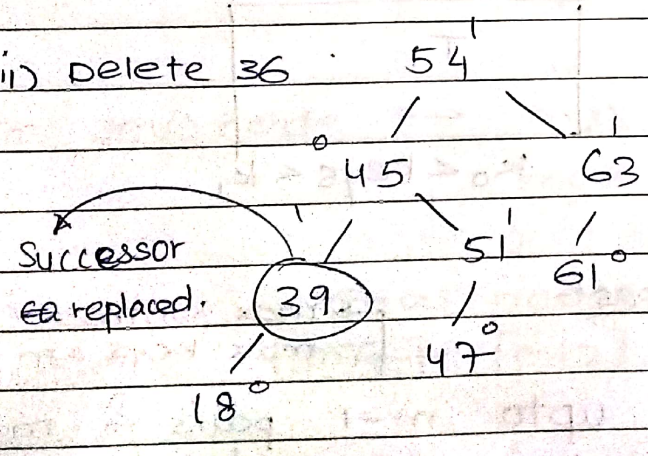
↳ step by step.

→ (i) Delete 52



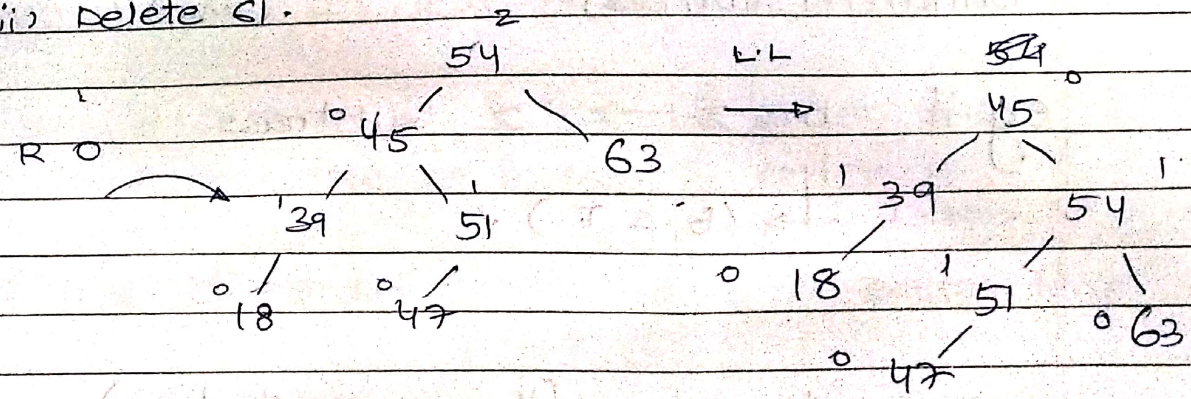
Balanced.

(ii) Delete 36



Balanced.

(iii) Delete 51.



B-TREE

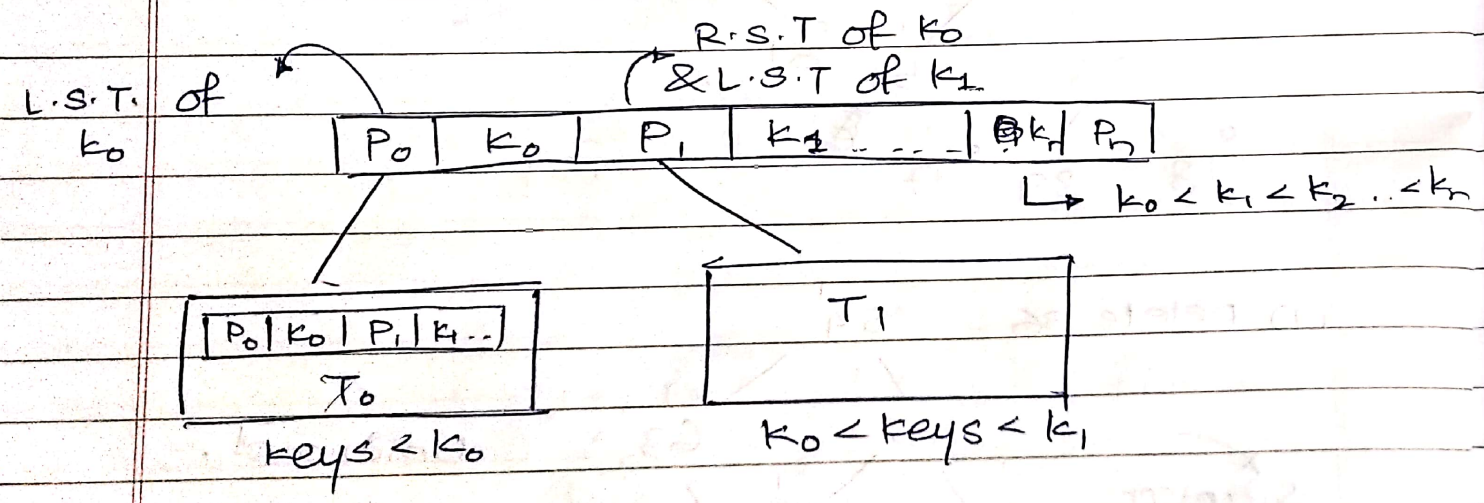
~~→ MULTIWAY TREES:-~~

~~→ MULTIWAY TREES:-~~

Large degree B-trees used to represent very large dictionaries that reside on disk.

Smaller degree B-trees used for internal memory dictionaries to overcome cache-miss penalties.

B trees are used to reduce disk access time.



→ m-way Search Trees:-

→ max subtrees = m
max keys = m-1

Each node has upto m-1 pairs (keys) & m children (subtrees)

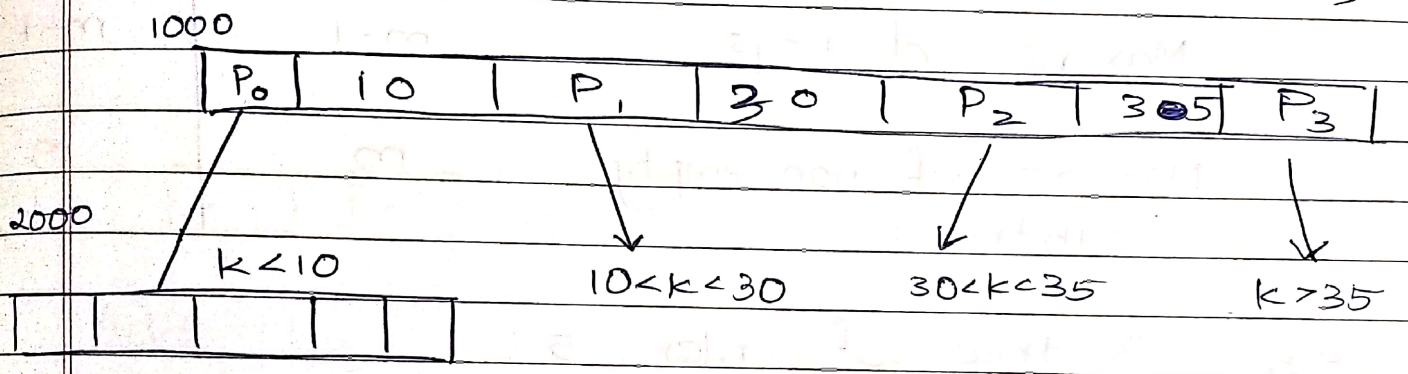
e.g. if m = 2 → 2 subtrees
↳ (B.S.T).

e.g. order = 4. (4 way search tree)

• 4-way search Tree:

P_0	k_0	P_1	k_1	P_2	k_2	P_3

↳ 4 subtrees (max).
& 3 keys (max) (keys arranged in ↑sing order)



For root node → min keys = 1
min subtrees = 2.

for any node → max keys = m − 1
max subtrees = m.

Remaining internal nodes (if any) should have atleast ~~⌈m/2⌉~~ ⌈m/2⌉ (ceil) children.

In B tree, all leaf nodes are at same level.

e.g. 2-3 tree is B-tree of order 3.
2-3-4 tree is B-tree of order 4.

↳ min no. of subtrees for all nodes except root node

B-tree of order 5 is 3-4-5 ~~order~~ tree.

∴ For a non-empty B-tree of order m.

	Root node	Non-root
Min. no. of keys	1	$\lceil m/2 \rceil - 1$
Min. no. of non-empty subtrees	2	$\lceil m/2 \rceil$
Max no. of keys	m-1	m-1
Max. no. of non-empty subtrees.	m	m

e.g. B-tree of order 5.

$\boxed{3, 14, 7, 1}$, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

→ ~~First~~ Take first (5-1) keys.

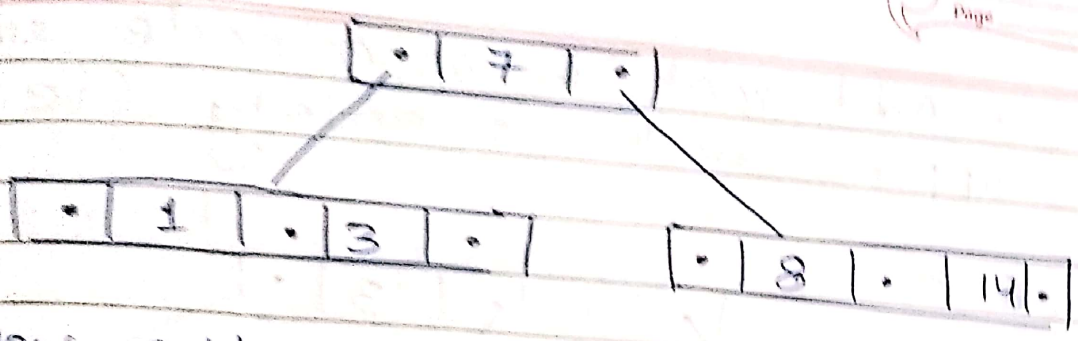
3, 14, 7, & 1. & arrange in ascending order.

•	1	•	3	•	7	•	14	•			

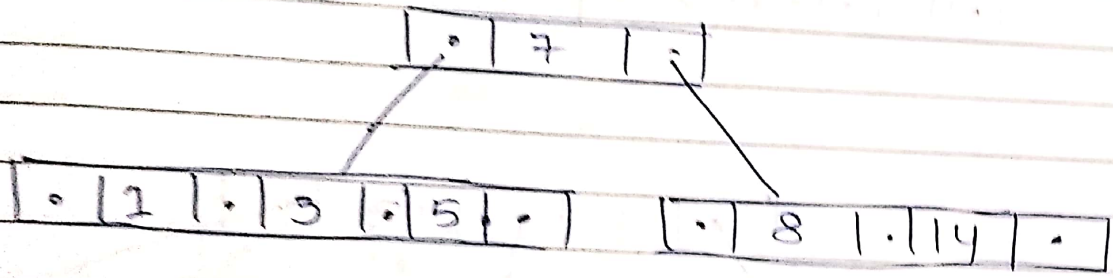
Now add 8.

We can add 8 in R.S.T of 7, but then $\lceil m/2 \rceil$ property won't get satisfied

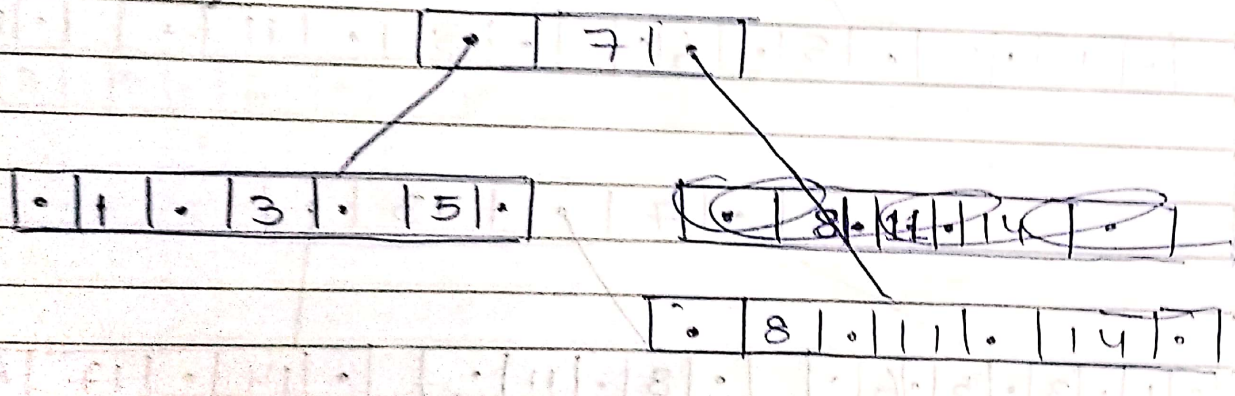
bcz $\lceil 5/2 \rceil = 3$ means we need atleast 3 sub trees in non-empty nodes.



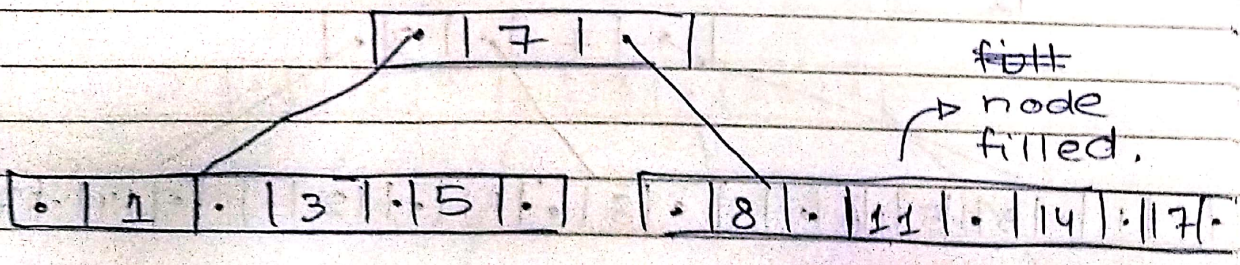
Now add 5 in L.S.T of 7, can be accomodated since maximal sub trees = 5.



Add 11,



Now add 17,



~~full~~ node filled.

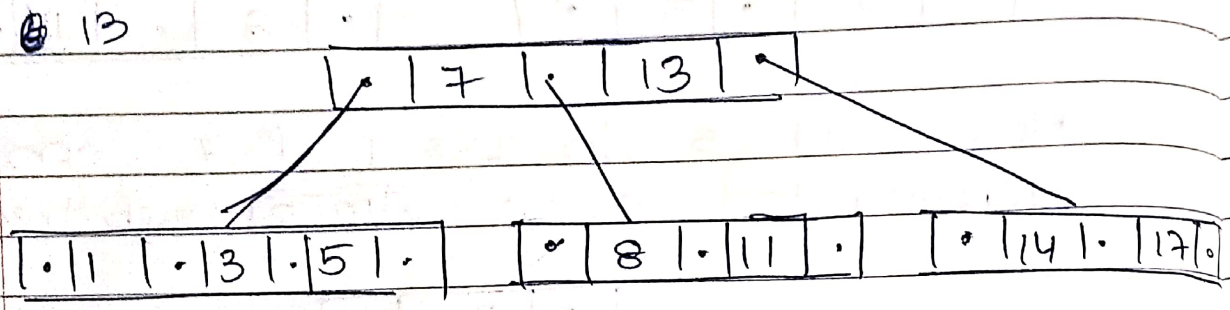
Now add 13,

cannot be accomodated in R.S.T of 7

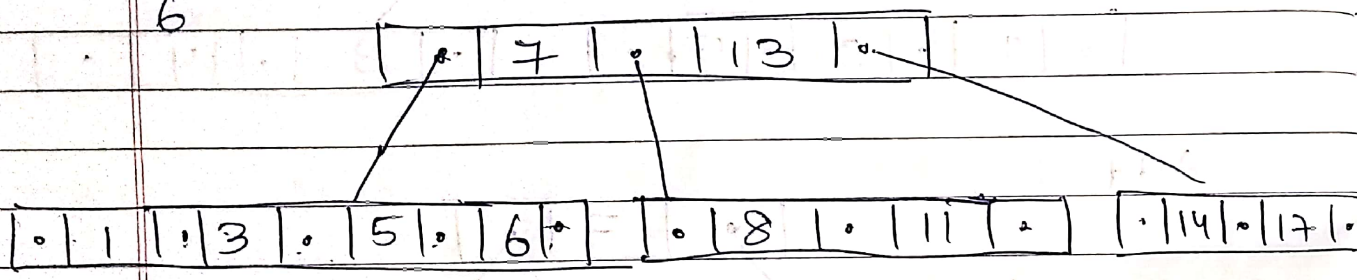
8, 11, 13, 14, 17 → root (∵ in middle)

∴ Add root in upper node since upper node is empty (i.e. $m<5$)

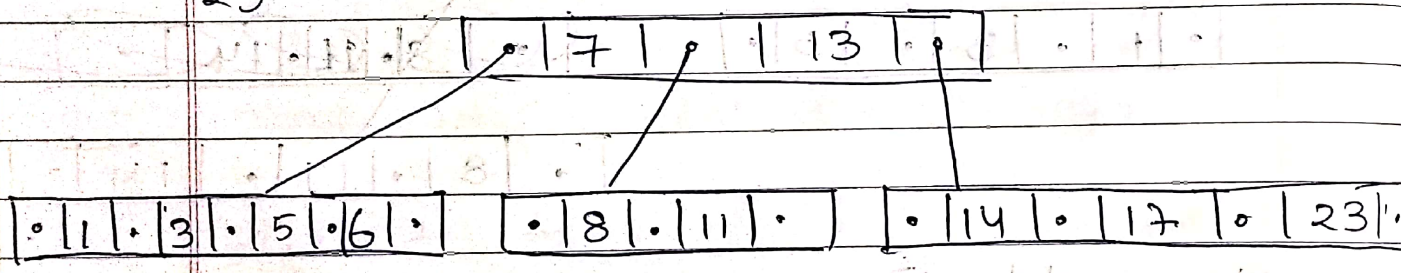
13



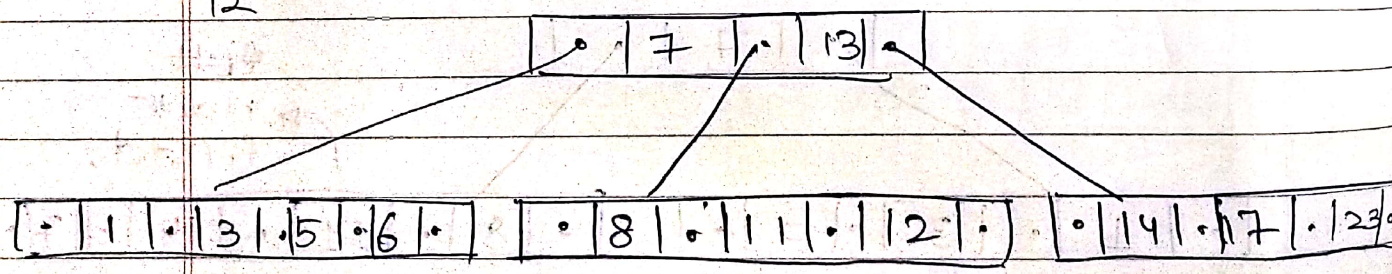
6



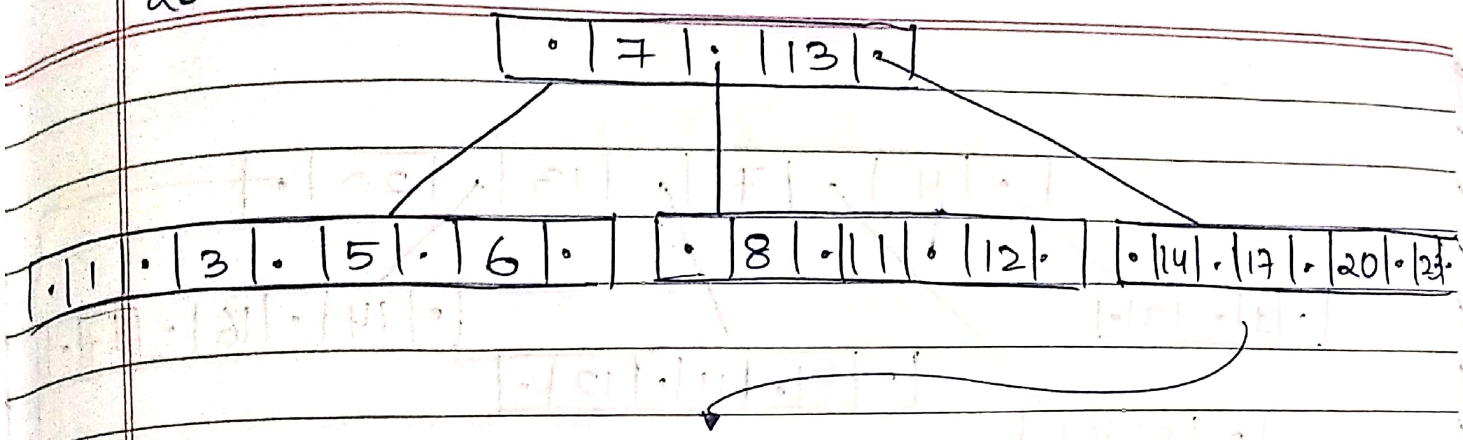
23



12



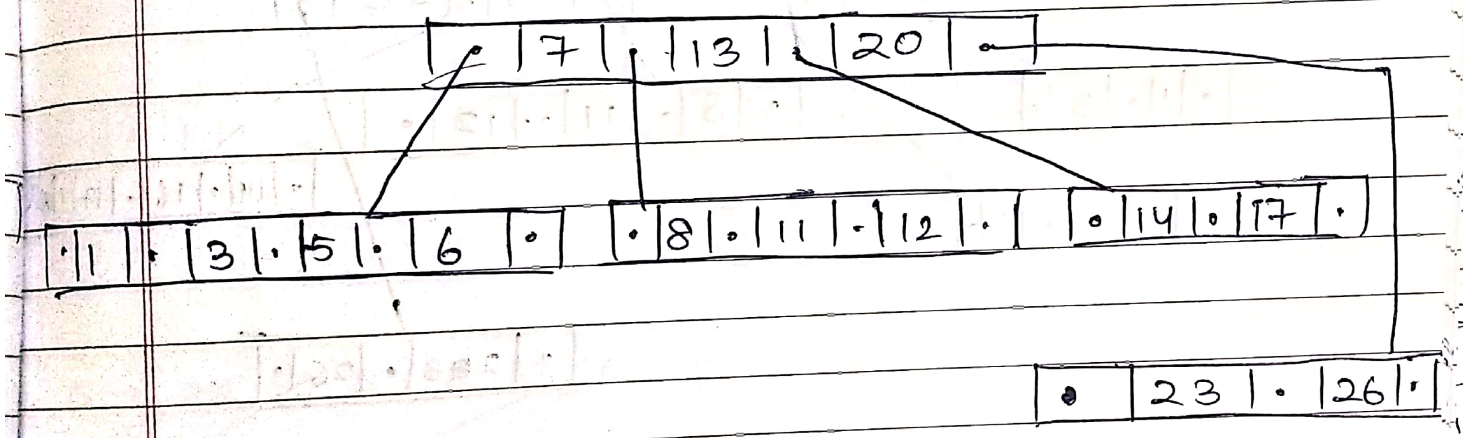
20



26

14, 17, 20, 23, 26

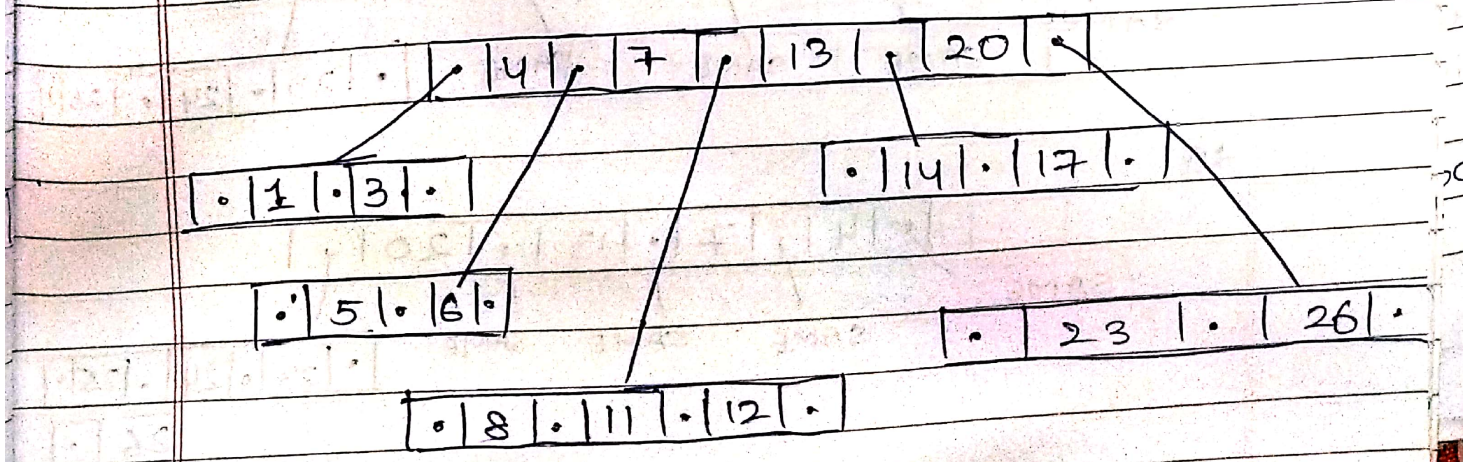
Root.



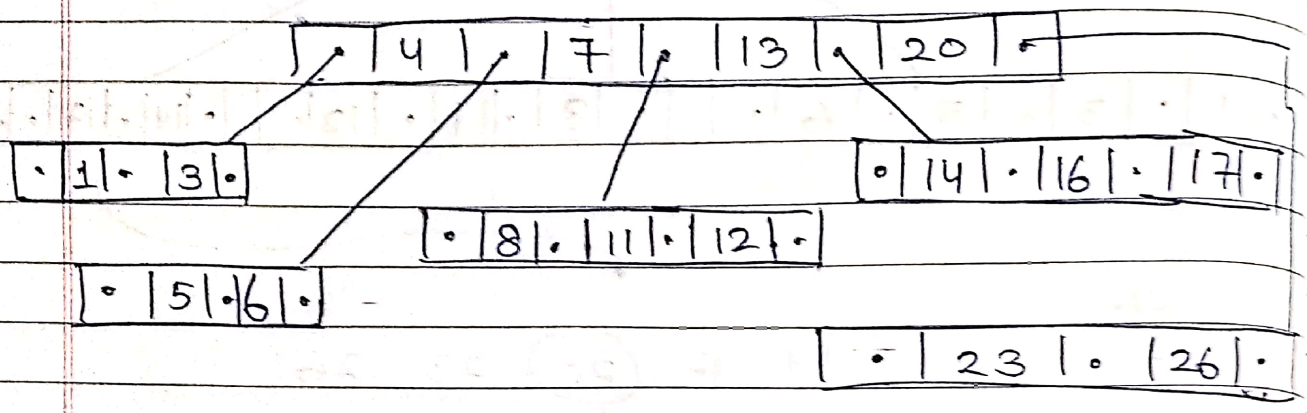
4

1, 3, 4, 5, 6

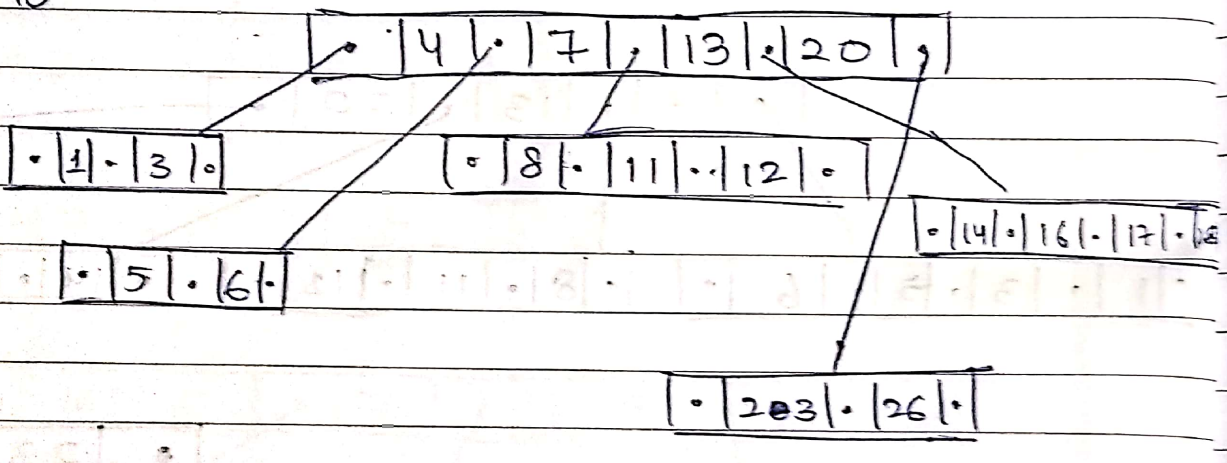
Root.



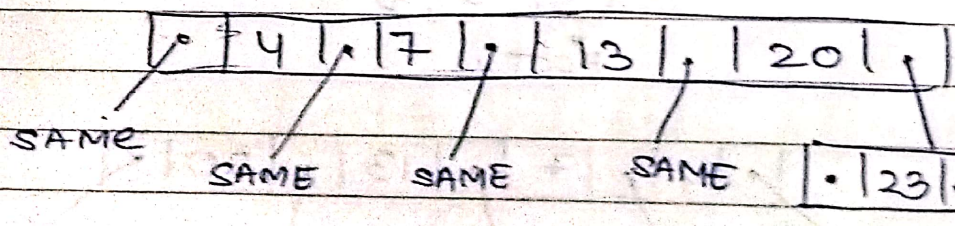
16


4
7
13
20
1
3
14
16
17
8
11
12
5
6
23
26

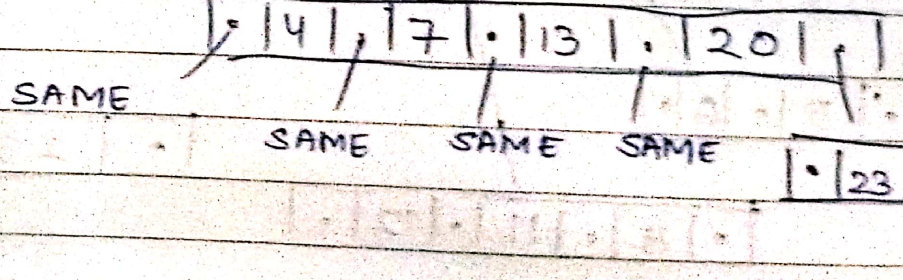
18


4
7
13
20
1
3
8
11
12
14
16
17
5
6
23
26

24


4
7
13
20
SAME
SAME
SAME
SAME
23
24
26

25


4
7
13
20
SAME
SAME
SAME
SAME
23
24
25
26

19.

14, 16, 17, 18, 19

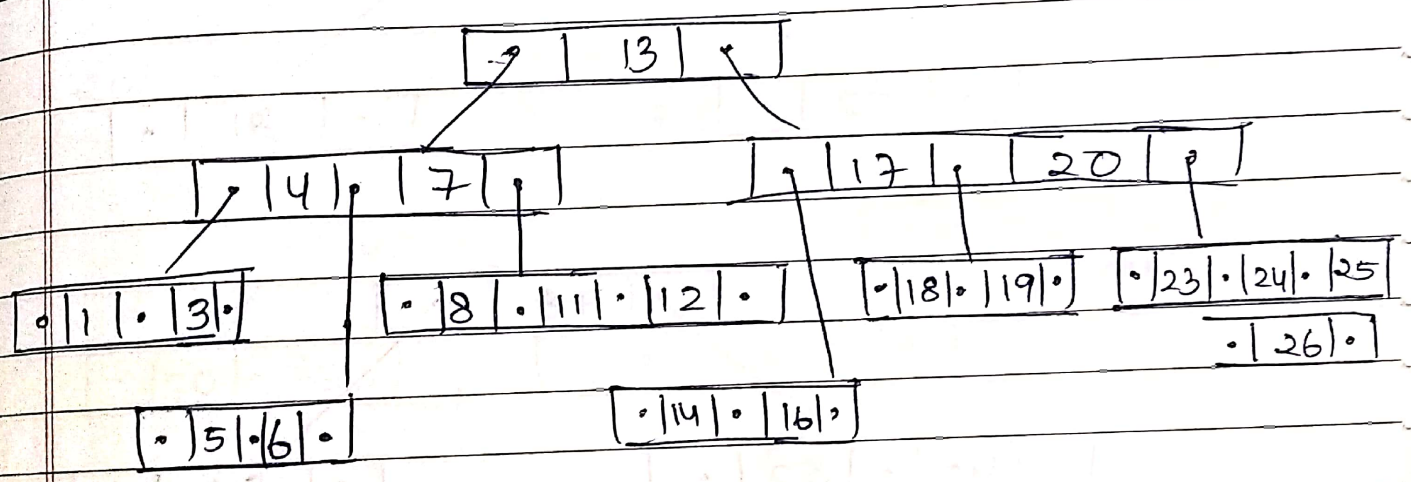
↓

Root

4, 7, 13, 17, 20

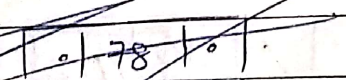
↓

Root.



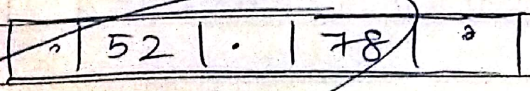
Q→ ~~78~~, ~~52~~, 81, 40, 33, ~~90~~, 85, 20, 30 Order - 3.

→ ~~78~~

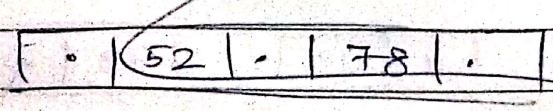


52

78, 52

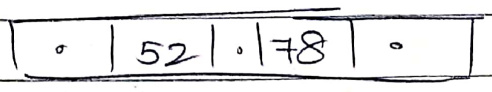


81

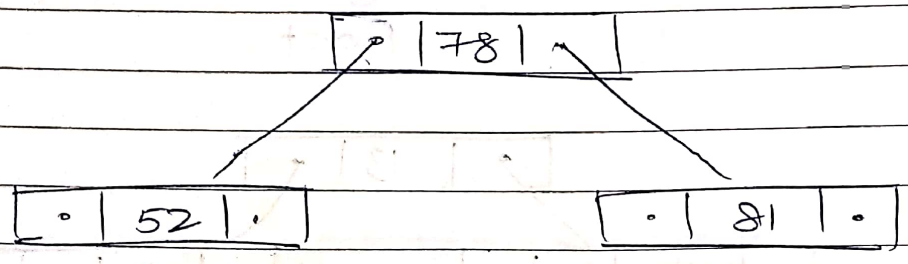


Q→ 78, 52, 81, 40, 33, 90, 85, 20, 30. Order-3

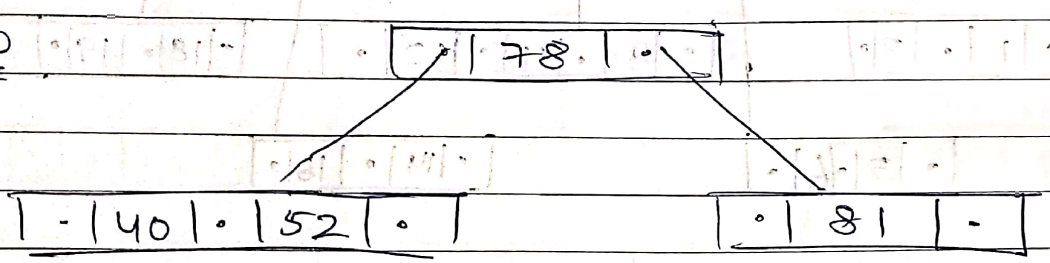
→ 78, 52



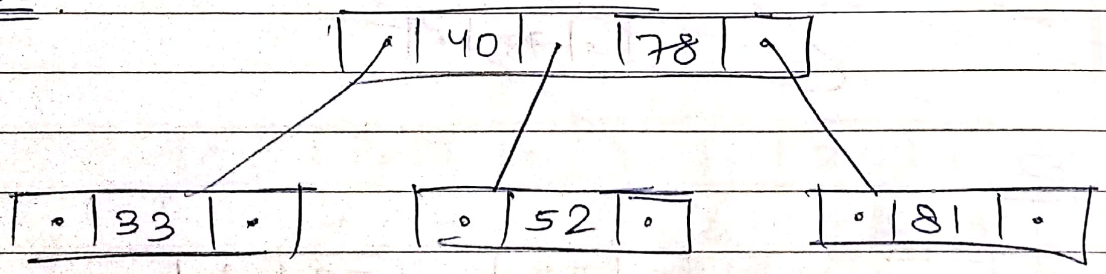
81



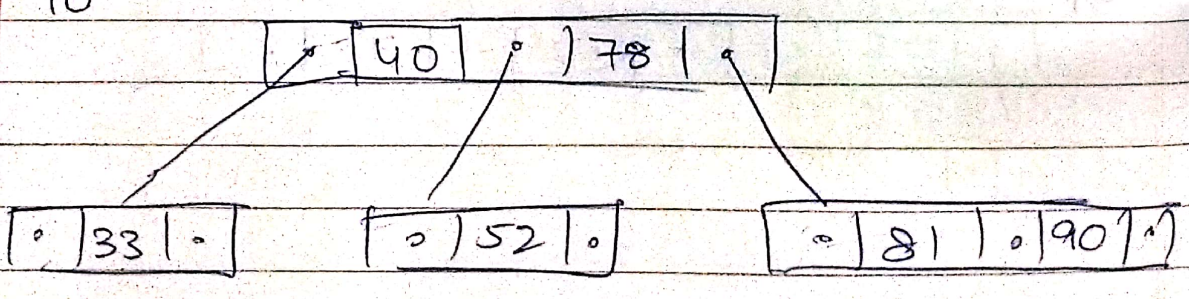
40



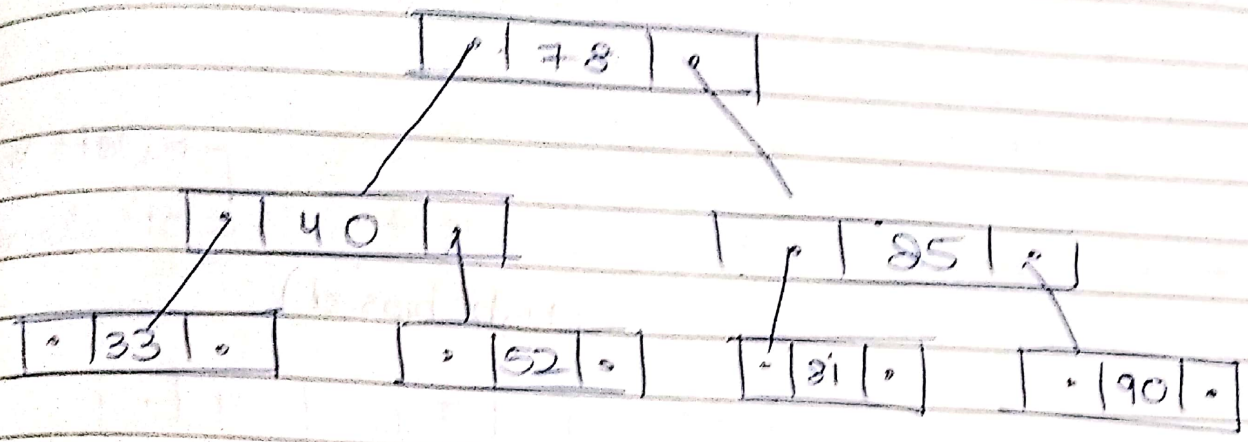
33



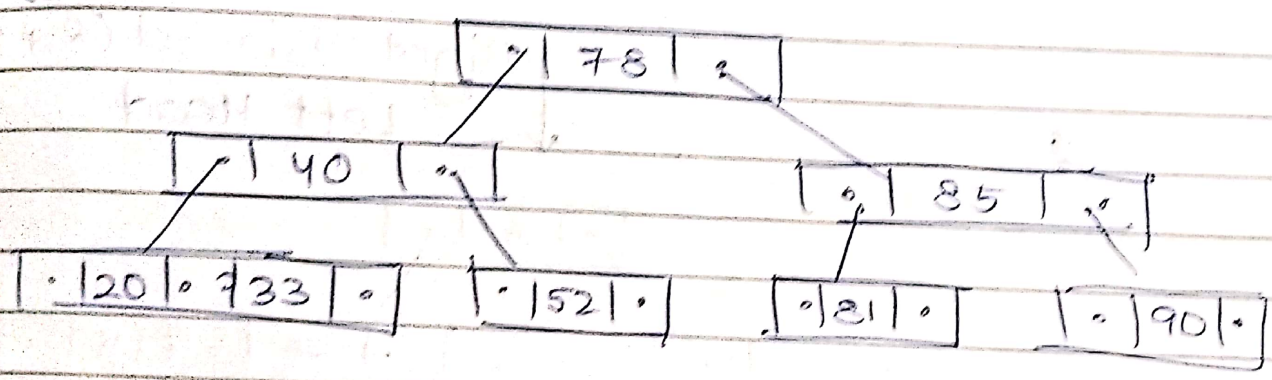
90



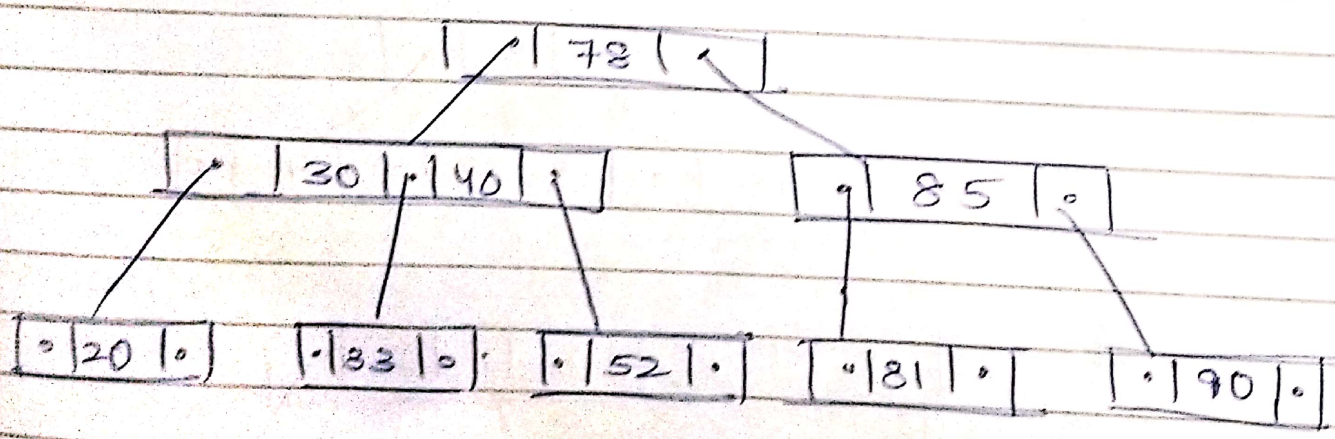
85



20



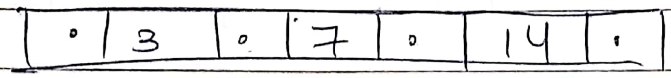
30



Q → 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

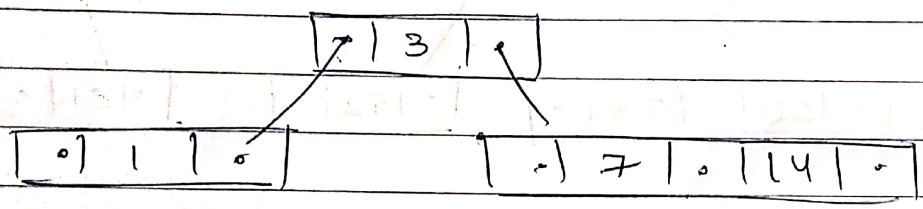
m = 4

→ Choose median as left (left biased) or right (right biased) element.

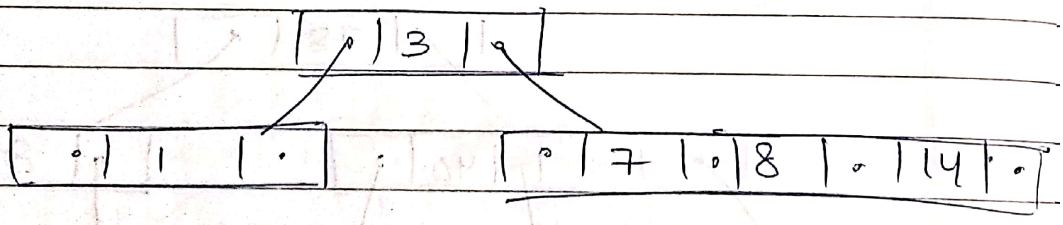


Right Biased (Right Heavy)
Left Head

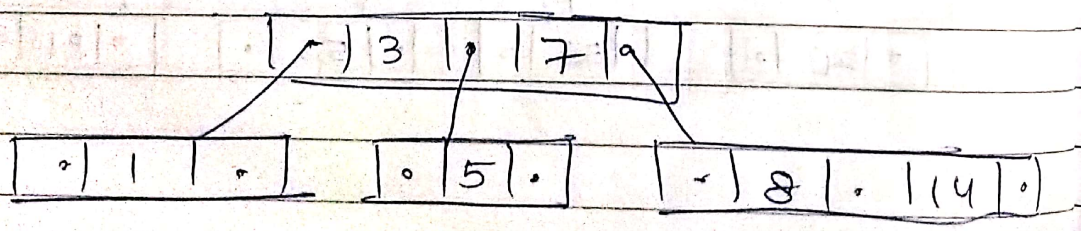
1



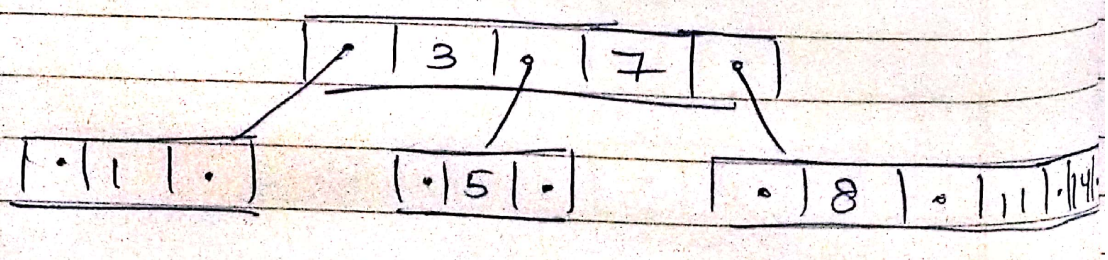
8



5



11.

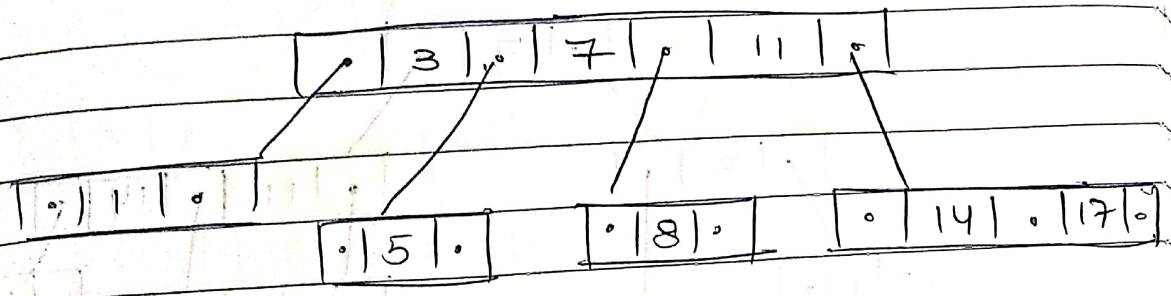


classmate

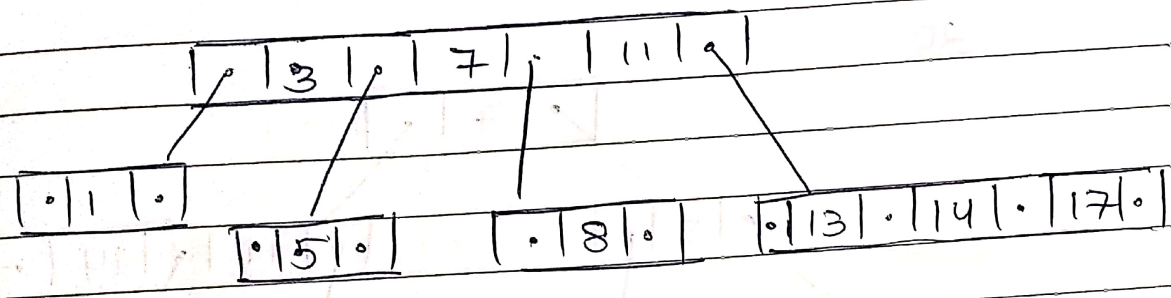
Date

Page

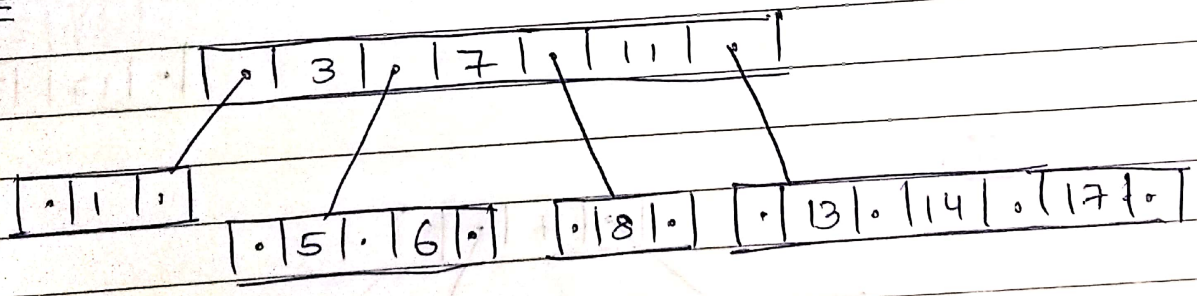
17.



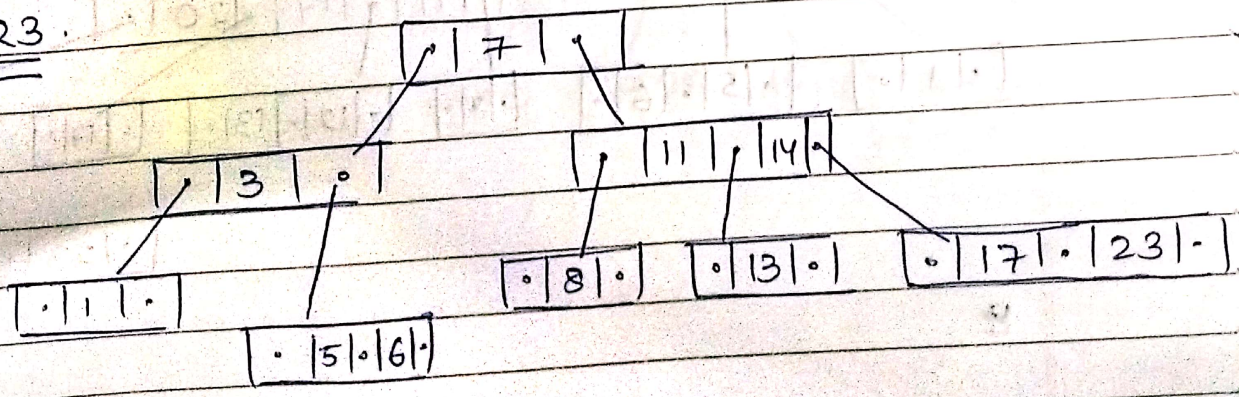
13.



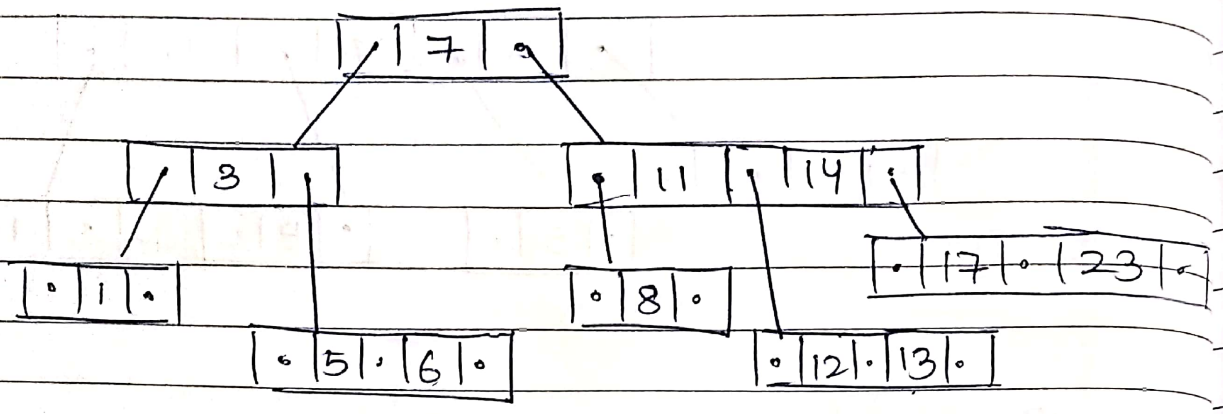
6.



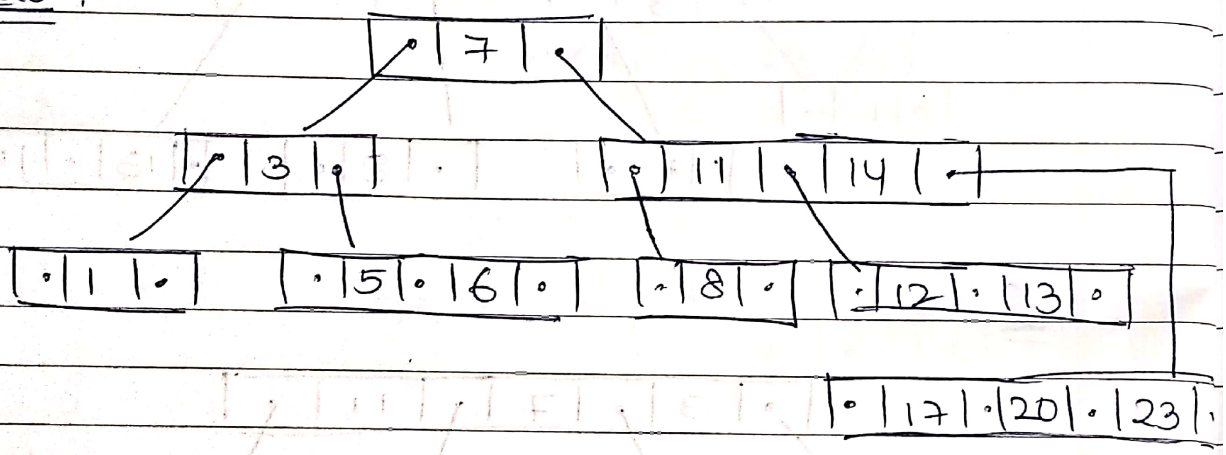
23.



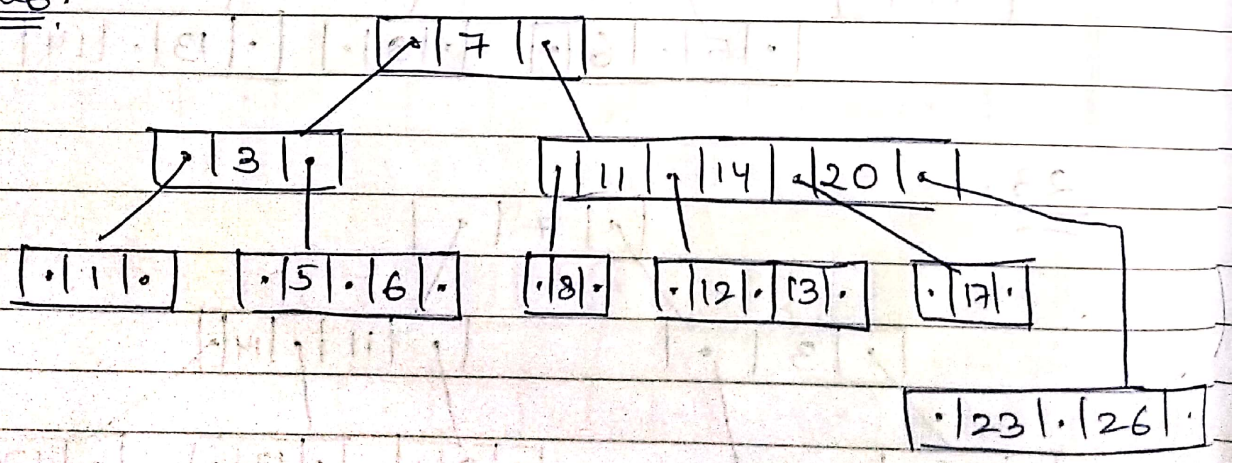
12.


7
3
11
14
1
5
6
8
12
13
17
23

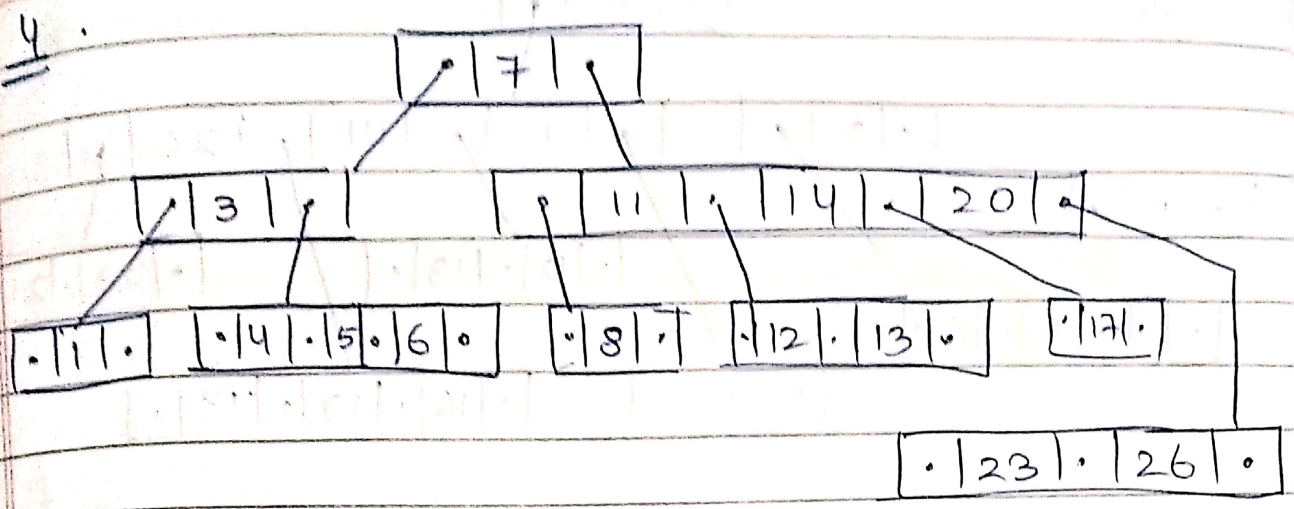
20.


7
3
11
14
1
5
6
8
12
13
17
20
23

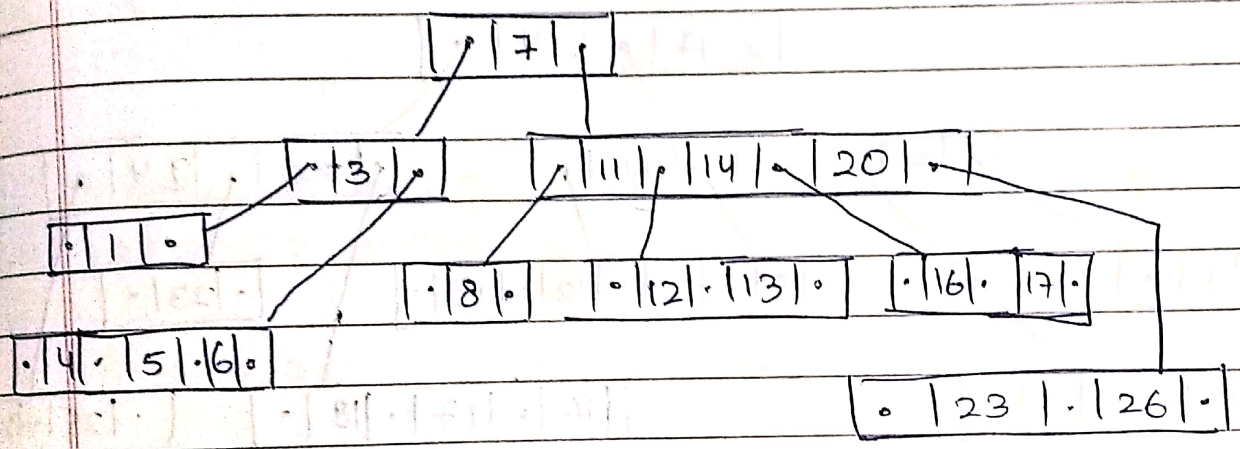
26.


7
3
11
14
20
1
5
6
8
12
13
17
23
26

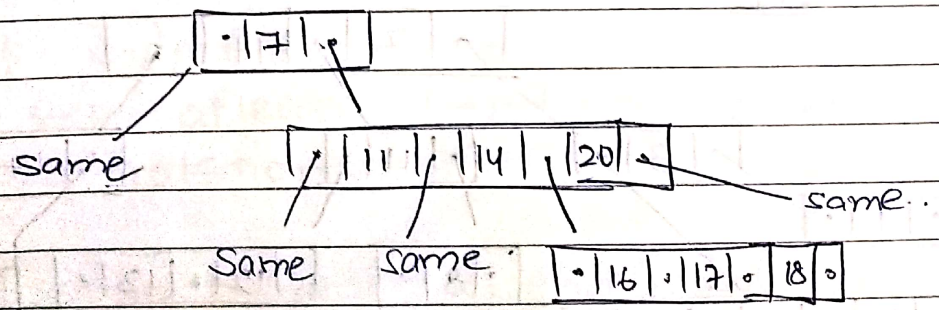
4.


7
3
11
14
20
1
4
5
6
8
12
13
17
23
26

16.

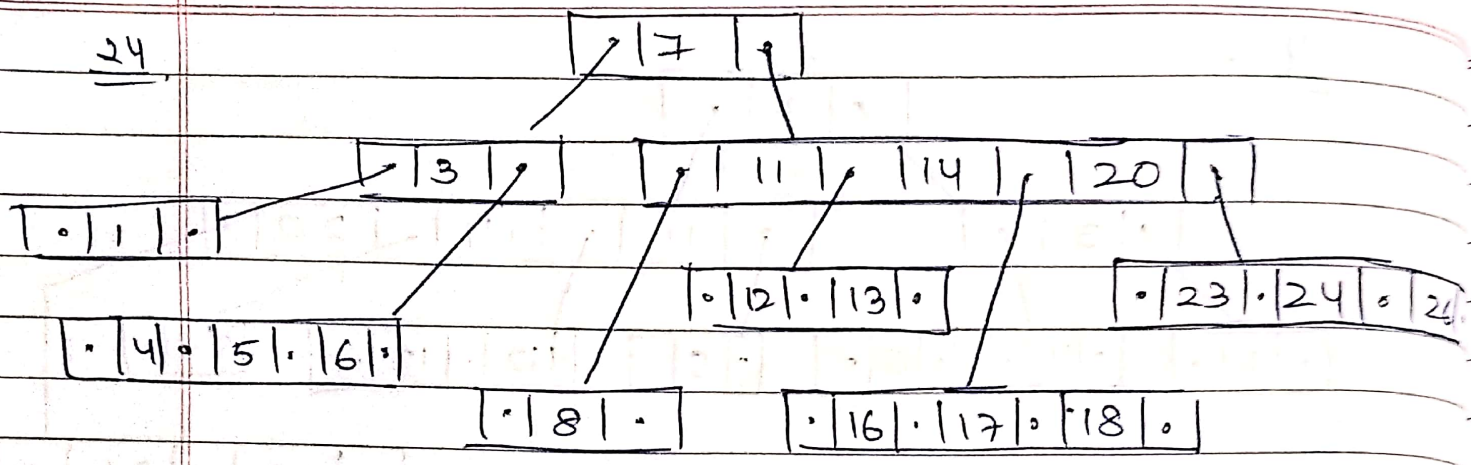

7
3
11
14
20
1
8
12
13
16
17
4
5
6
23
26

18.

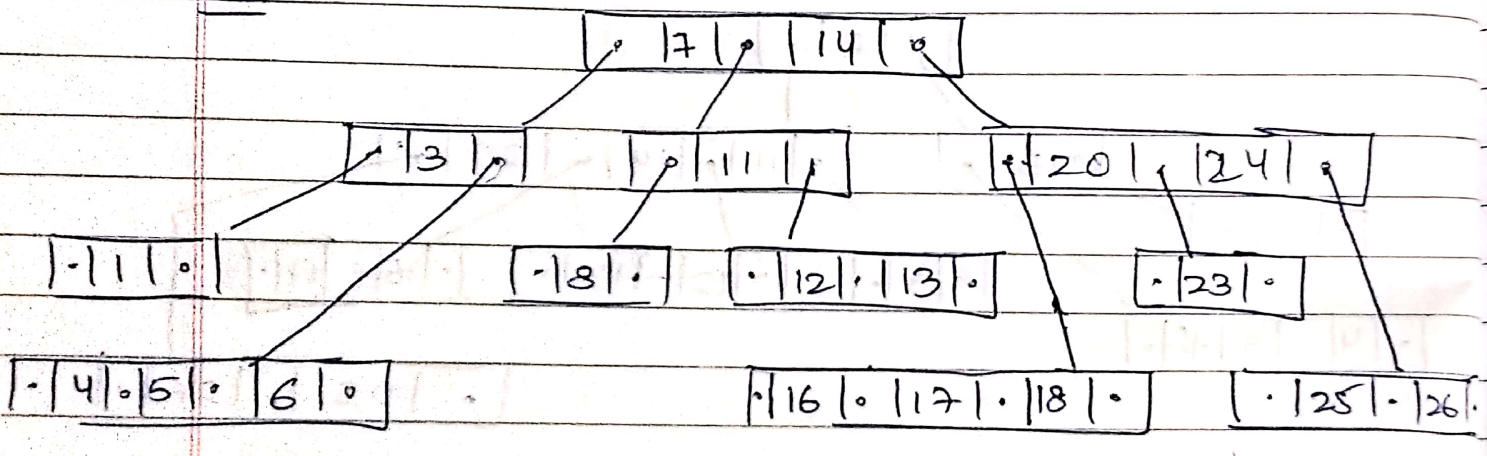

7
same
11
14
20
same
Same
same
16
17
18

24.

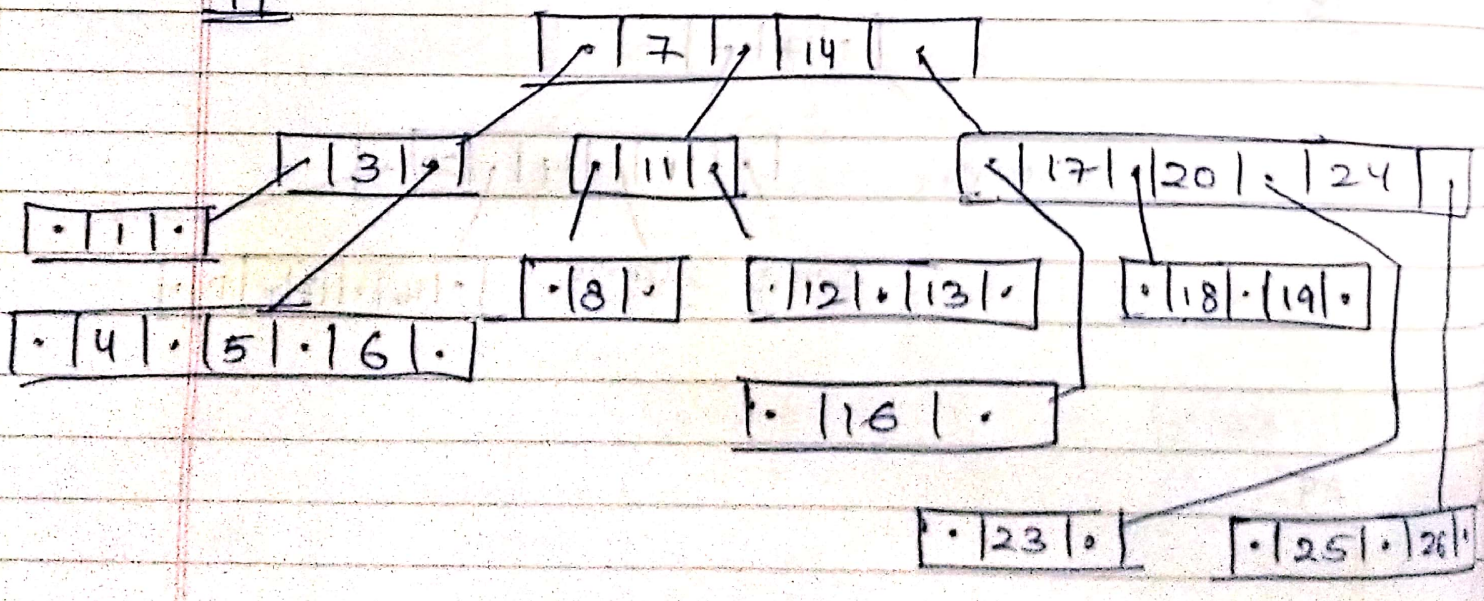
24.


7
3
11 14 20
1
12 13
23 24
4 5 6
8
16 17 18

25.


7 14
3
11
20 24
1
8
12 13
23
4 5 6
16 17 18
25 26

19.


7 14
3
11
17 20 24
1
8
12 13
18 19
4 5 6
16
23
25 26

DELETION IN B-TREES:-

Underflow condition:

↓

~~Af~~ It occurs when a non-root node contains $\lceil m/2 \rceil - 2$ keys after a key deletion.

If a node underflows, rotate the appropriate key from the adjacent right or left-~~sibling~~ sibling if the sibling contains atleast $\lceil m/2 \rceil$ keys otherwise perform merging.

A key rotation must always be attempted before a merging.

There are 5 deletion cases:

(i) The leaf node does not underflow.

(ii) The leaf underflows & the adjacent right sibling has atleast $\lceil m/2 \rceil$ keys perform a left key-rotation.

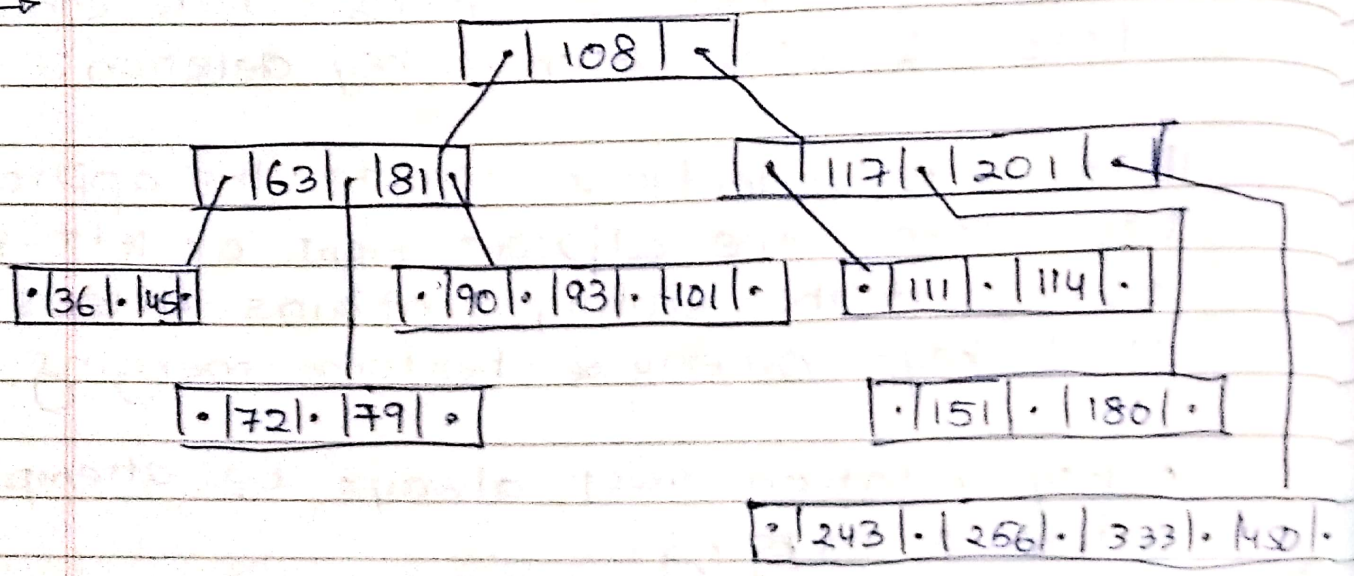
(iii) The leaf underflows & the adjacent left sibling has atleast $\lceil m/2 \rceil$ keys perform a right-key rotation.

(iv) The leaf underflows & each of the adjacent right sibling & the adjacent left sibling has atleast $\lceil m/2 \rceil$ keys. perform either a left or a right key rotation.

(v) The leaf underflows & each adjacent sibling has $\lceil m/2 \rceil - 1$ keys perform a merging.

Q→

m = 5.



Delete (i) 93
(ii) 201
(iii) 180
(iv) 72

→ Min keys in every non-root $= \lceil m/2 \rceil - 1 = \lceil 5/2 \rceil - 1$

$= \underline{\underline{2}}$.

HEAP DATA STRUCTURE

A heap is a nearly complete binary tree with the following 2 properties:

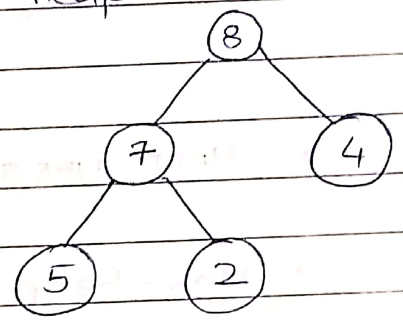
- Structural Property: all levels are full, except possibly the last one, which is filled from left to right.

- Order (heap) property :- For any node x,

$$\text{Parent}(x) \geq x. \leftarrow \text{Max. heap.}$$

From the heap property, it follows that:

"The root is the maximum element of the heap!".



Max. Heap.

A heap is a binary tree that is filled in order.

→ ARRAY REPRESENTATION OF HEAP:-

A heap can be stored as an array A.

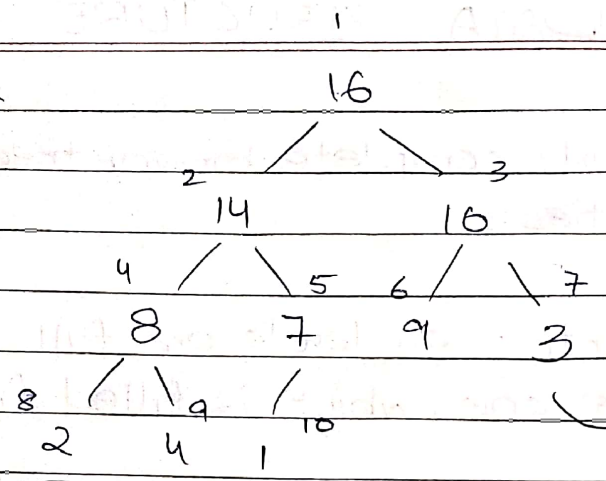
- Root of tree is $A[1]$
- Left child of $A[i] = A[2i]$
- Right " " " $= A[2i+1]$
- Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
- Heapsize $[A] \leq$ Length $[A]$.

Heapsize = no. of elements in the heap.

n = no. of nodes.

The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves.

e.g.



Heapsize = 10

length of array = 11.

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10.

→ HEAP TYPES:-

- Max-heaps (largest element at root):

for all nodes i, excluding the root,

$$A[Parent(i)] \geq A[i]$$

- Min-heaps (smallest element at root):

for all nodes i, excluding the root

$$A[Parent(i)] \leq A[i].$$

→ ADDING / DELETING NODES:-

New nodes always inserted at the bottom level (left to right)

Nodes are removed from the bottom level.

→ OPERATIONS ON HEAP:-

(i) MAX - HEAPIFY (Maintain / Restore the max-heap property)

(ii) BUILD - MAX - HEAP (Create a max-heap from an unordered array)

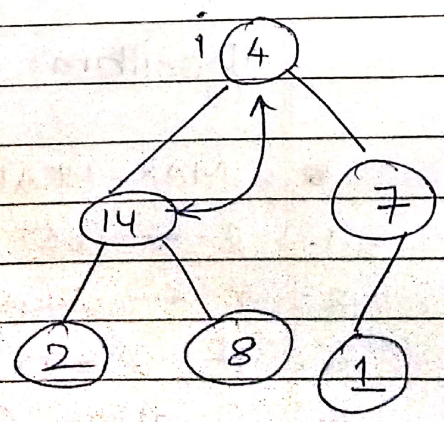
(iii) HEAPSORT (sort an array in place)

(iv) Priority queues.

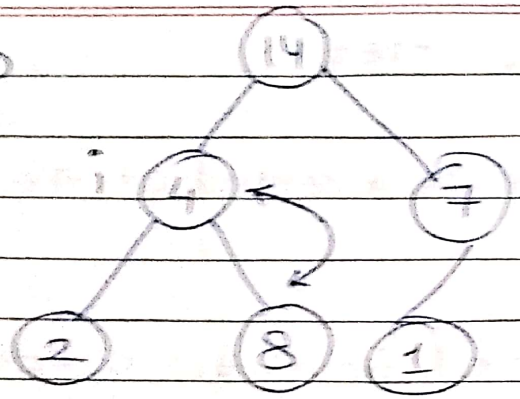
→ MAINTAINING THE HEAP PROPERTY (MAX - HEAPIFY):-

- suppose a node is smaller than a child
  - left & right subtrees of i are max-heaps.

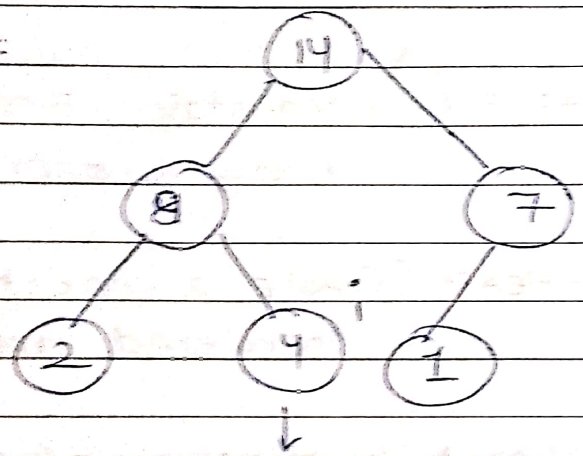
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree.
  - Continue until node is not larger ~~smaller~~ than children.



Step (i)



Step (ii) :



No exchanges possible now, so stop.

Assumptions :

- Left & right subtrees of i are max-heaps & a[i] may be smaller than its children.

Algorithm :

n = no. of nodes.
i = node ka index.
A = array storing node values.

```
MAX-HEAPIFY (A, i, n)
l ← LEFT(i)          // storing indexes
r ← RIGHT(i)         // storing indexes
If l ≤ n & A[l] > A[i]
    then largest ← l
    else largest ← i
```

l & r are indexes of left & right nodes. A[l], A[r] are uske values.

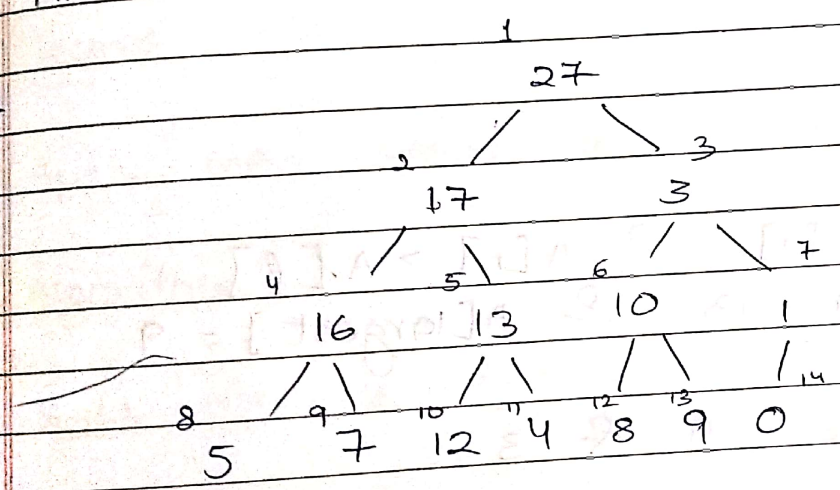
largest = index of largest value.

if $r \le n$ & $A[r] > A[largest]$
then largest $\leftarrow$ r.

if largest $\neq$ i
then exchange $A[i] \leftrightarrow A[largest]$
MAX-HEAPIFY (A, largest, n).

A = {27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0}.

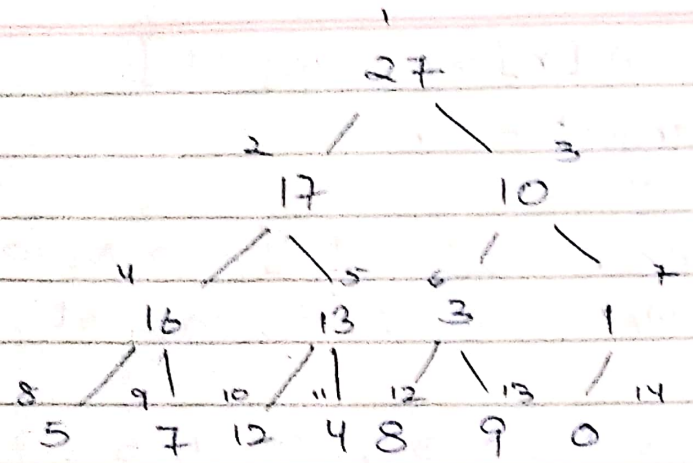
MAX-HEAPIFY (A, 3).



n = 14. , i = 3

(i) MAX-HEAPIFY (A, 3, 14)
i = 3.
l = 6 (2i)
r = 7 (2i + 1)

A[l] > A[i] & A[l] > A[r]
∴ largest = 6. & A[largest] = 10
∴ exchange A[largest] & A[i]

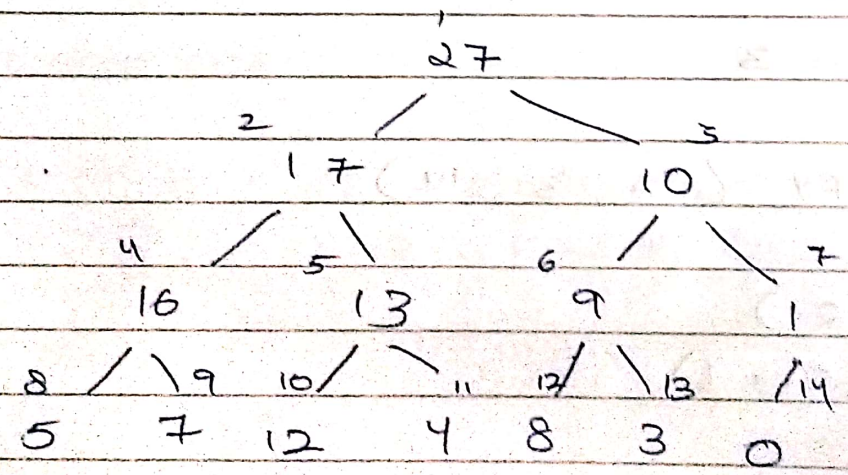


Step (ii) MAX- HEAPIFY (A, 6, 14)

$i = 6$
$l = 12$
$r = 13$.

$A[l] > A[i]$ & $A[r] > A[l]$
$\therefore$ largest $= r = 13$ & $A[\text{largest}] = 9$

$\therefore$ exchange 9 & 3.



Step (iii) MAX - HEAPIFY (A, 13, 14)

$i = 13$
$l = 26$ (& $26 \nless 14$) $\therefore$ EXIT if loop.
$r = 27$ (& $27 \nless 14$) $\therefore$ " " "
$\therefore$ largest $= i = 13$ $\therefore$ EXIT if loop
$\therefore$ EXIT FUNCTION.

MAX - HEAPIFY Run Time:-

↳ Time complexity : $O(\log n)$.

BUILDING A HEAP:-

Convert an array $A[1 \ldots n]$ into a max-heap ($n = \text{length}[A]$).

The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ are leaves.

Apply MAX-HEAPIFY on elements b/w 1 & $\lfloor n/2 \rfloor$

Algorithm:

BUILD - MAX - HEAP (A)

1. $n = \text{length}[A]$
2. for $i \leftarrow \lfloor n/2 \rfloor$ downto 1.
3. do MAX - HEAPIFY (A, i, n)

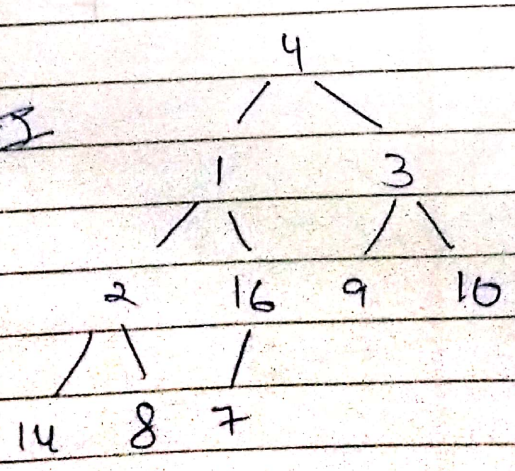
e.g. A :

4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

heap size.

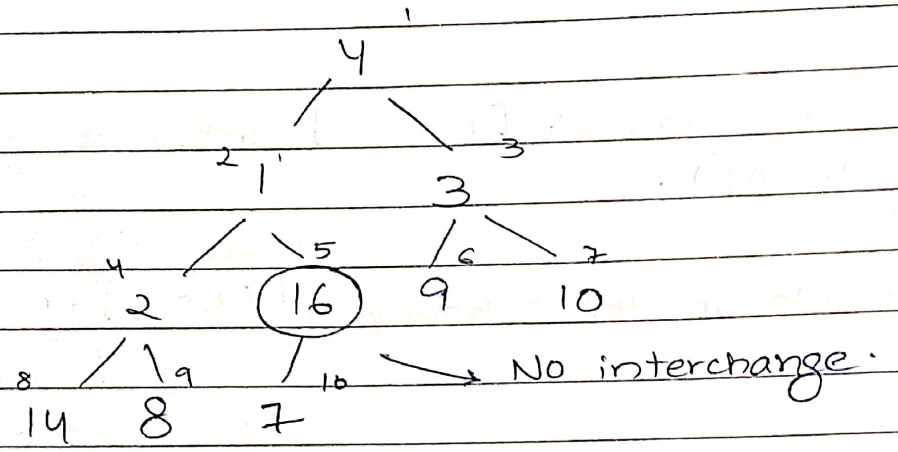
n = 10 (10 elements) ~~+1~~ ~~(0th index)~~

Step(i) Create ~~random~~ binary tree for given array.



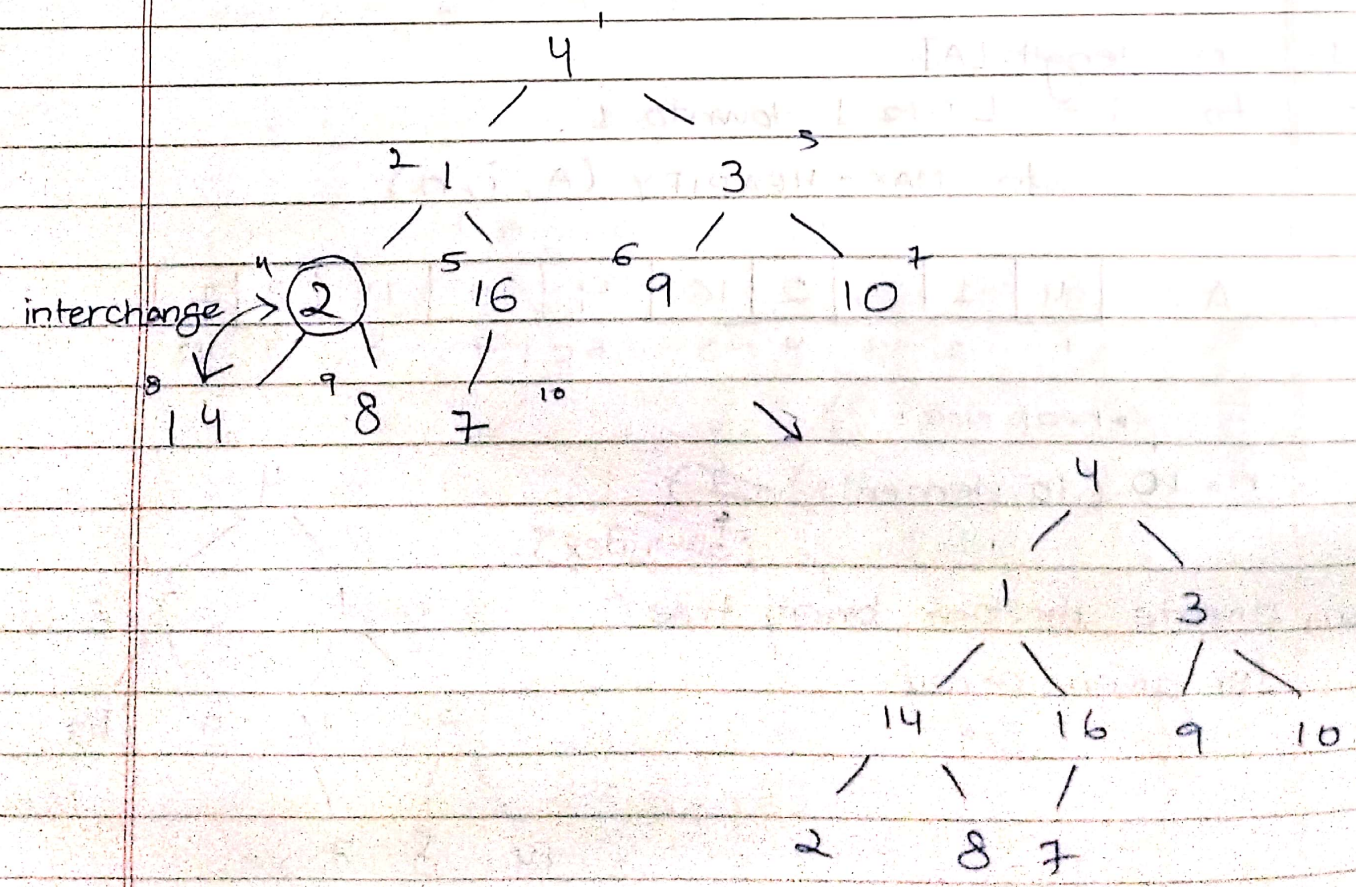
Step (ii) i = 5 to 1.

MAX - HEAPIFY (A, 5, 10)

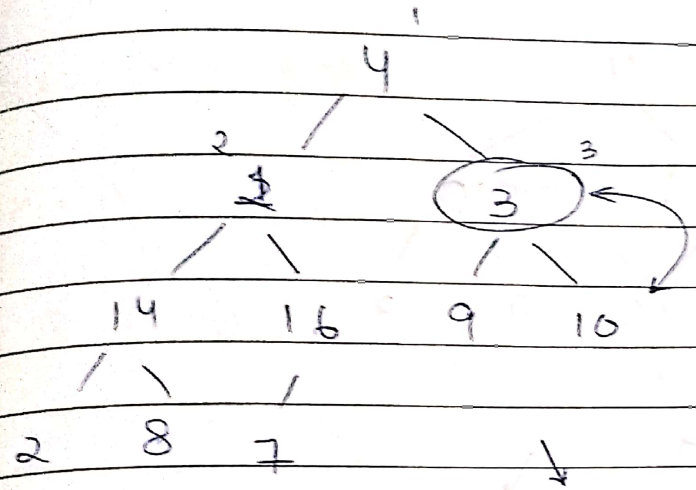


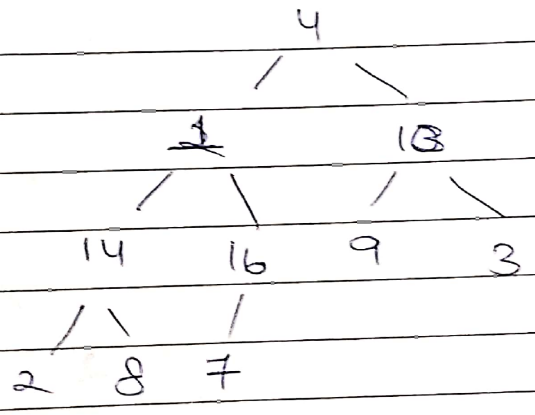
Step (iii) i = 4.

MAX - HEAPIFY (A, 4, 10)

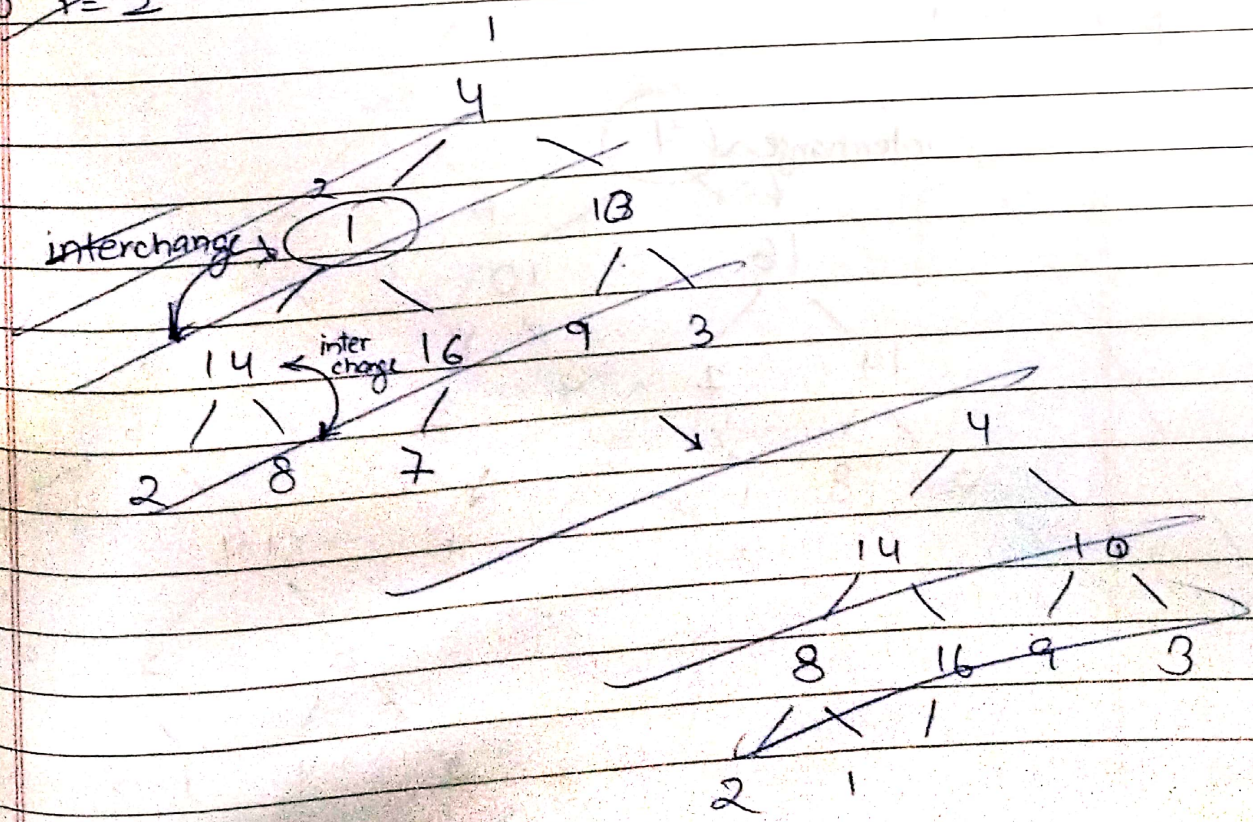


i = 3.

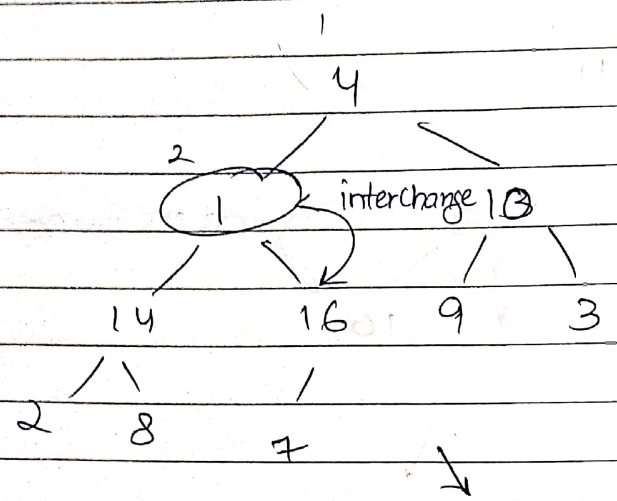

1
4
2
1
3
3
interchange.
14
16
9
10
2
8
7

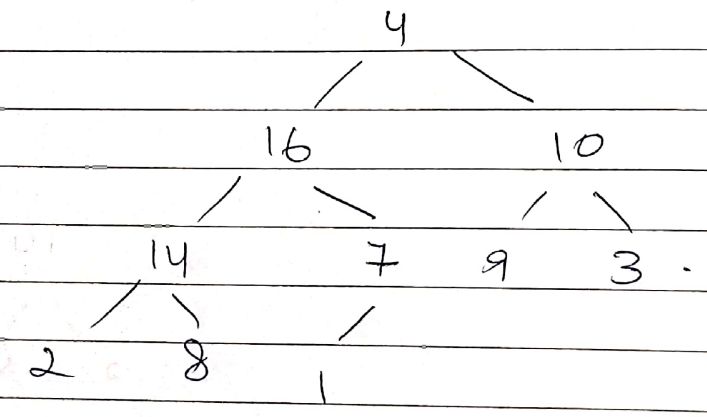

4
1
10
14
16
9
3
2
8
7

~~i = 2~~

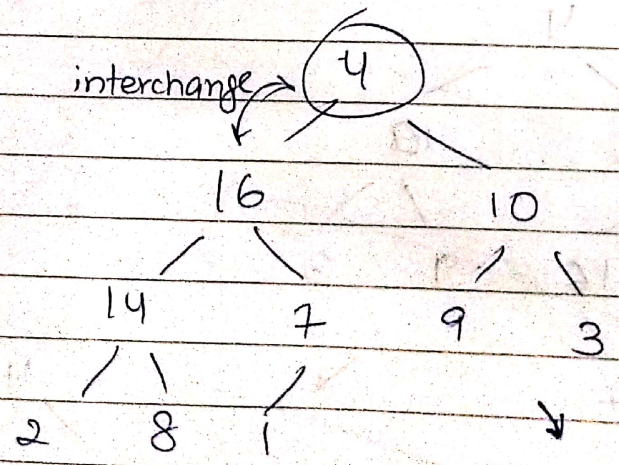

1
4
2
1
10
interchange
14
inter change
16
9
3
2
8
7
4
14
10
8
16
9
3
2
1

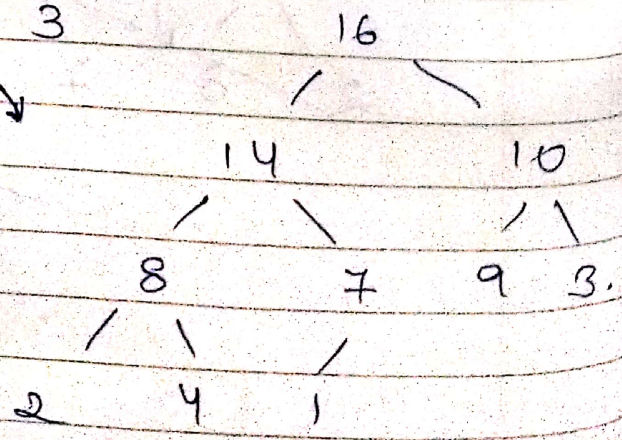
Step(v) i=2





Step(vi) i=1





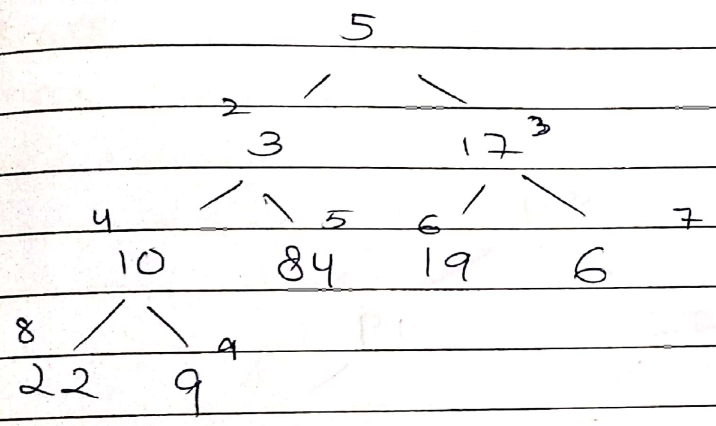
A : [5, 3, 17, 10, 84, 19, 6, 22, 9]

$n = 9$.

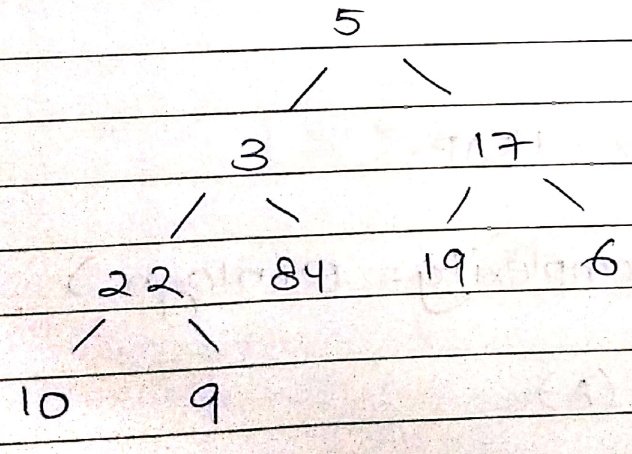
$\lfloor n/2 \rfloor = 4$.

$\therefore$ $i = 4$ to 1.

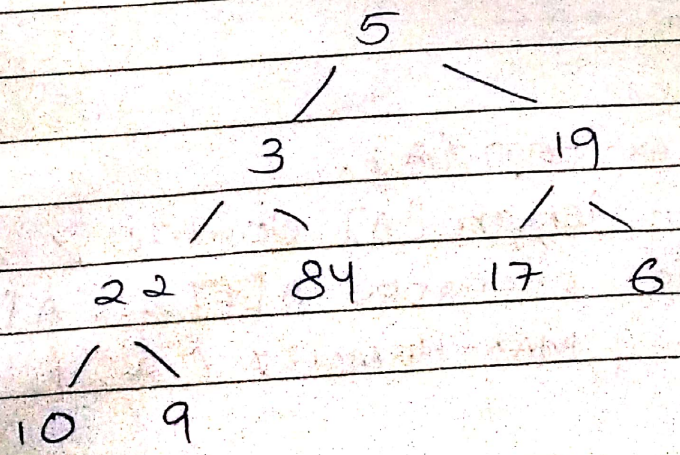
step(i)



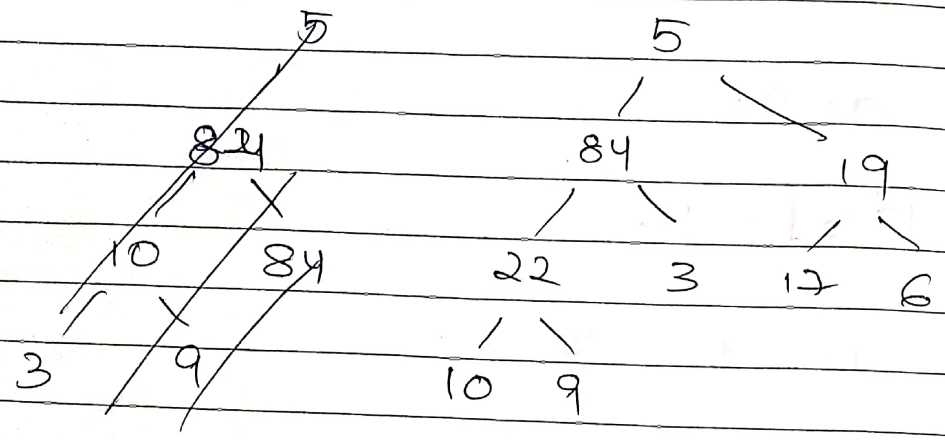
step(ii) $i = 4$.



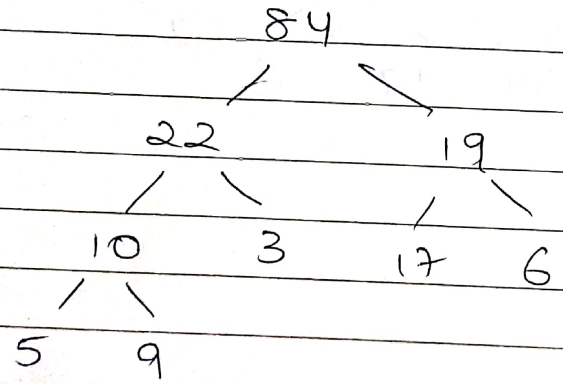
step(iii) $i = 3$



Step (iv) i = 2



step (v) i = 1



BUILD - MAX - HEAP:

↳ Time complexity: $O(n \log n)$.

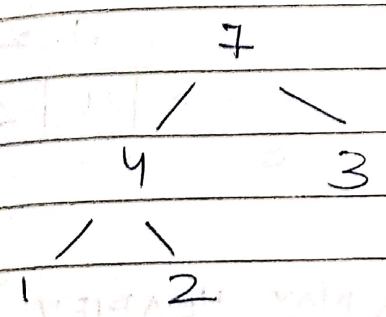
→ HEAP-SORT (A):-

Algorithm:

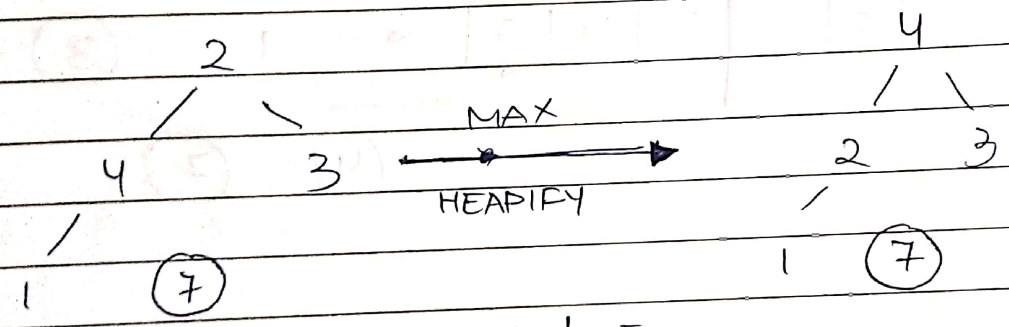
1. BUILD-MAX-HEAP (A).
2. for i ← length [A] downto 2.
3. do exchange A[1] ↔ A[i]
4. MAX-HEAPIFY (A, 1, i-1)

Date
Page

e.g. A = [7, 4, 3, 1, 2]



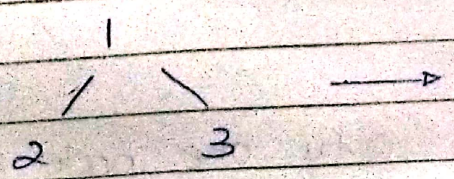
Step(i) swap root & last node & remove root & max-heapify remaining heap.



1	2	3	4	5
2	4	3	1	7

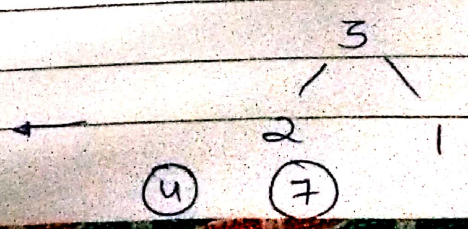
1	2	3	4	5
~~4~~	~~2~~	3	1	7

Step(ii) Exchange 4 with 1.



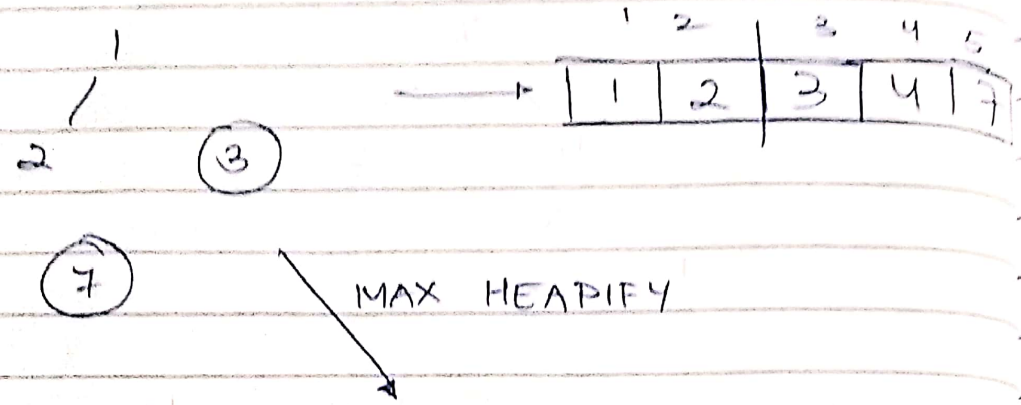
1	2	3	4	5
1	2	3	4	7

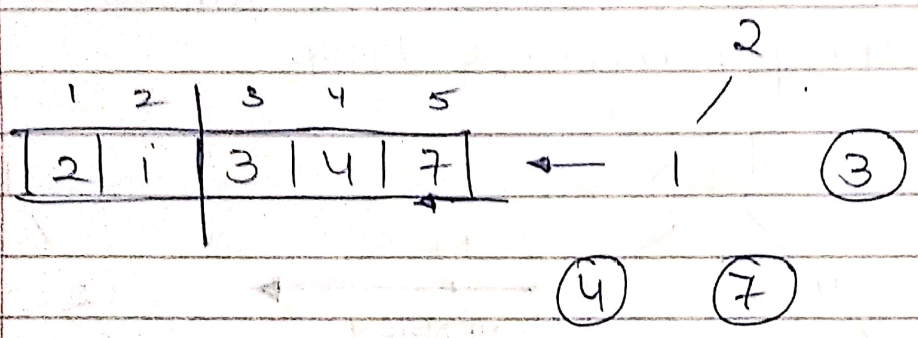
MAX HEAPIFY



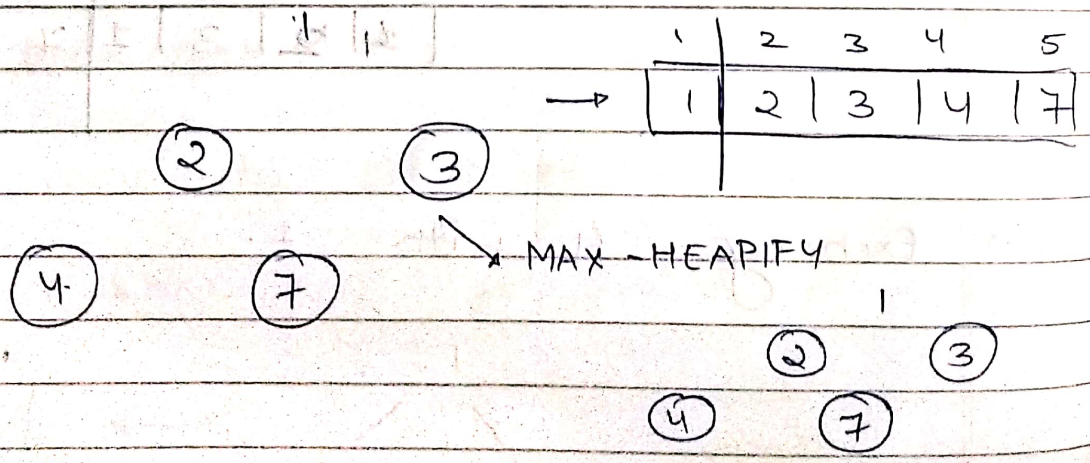
1	2	3	4	5
3	2	1	4	7

Step(iii) Replace 3 with 1.

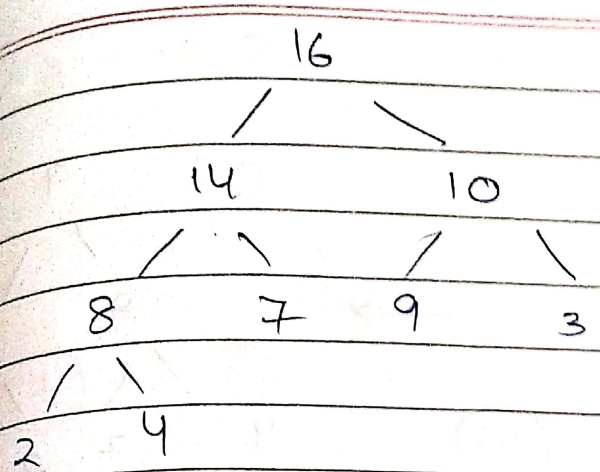




Step(iv) Replace 2 with 1.

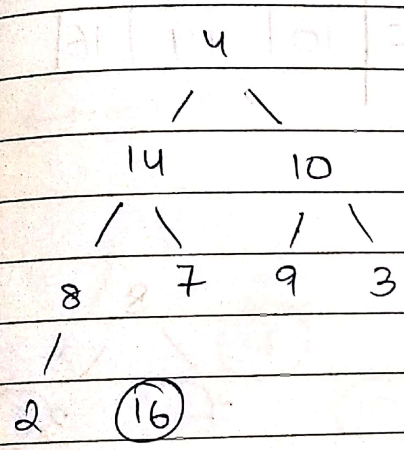


Stop when only one node is remaining.

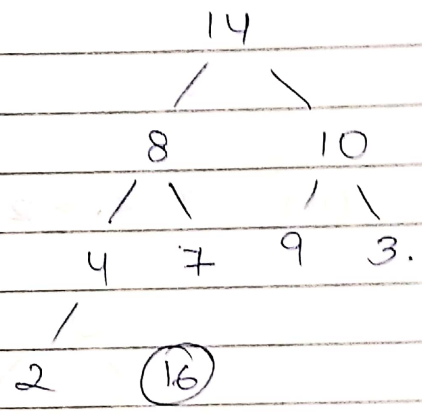


→ sort.

Step (i) Replace 16 with 4.

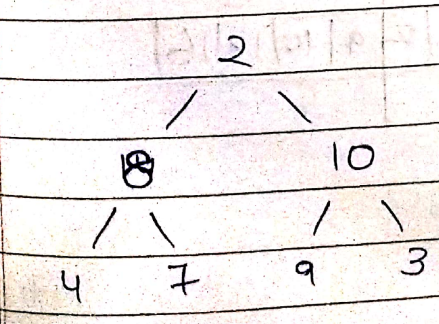


MAX-HEAPIFY →

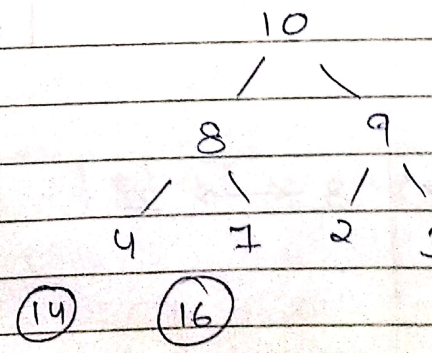


14	8	10	4	7	9	3	2	16

(ii) Replace 14 with 2.

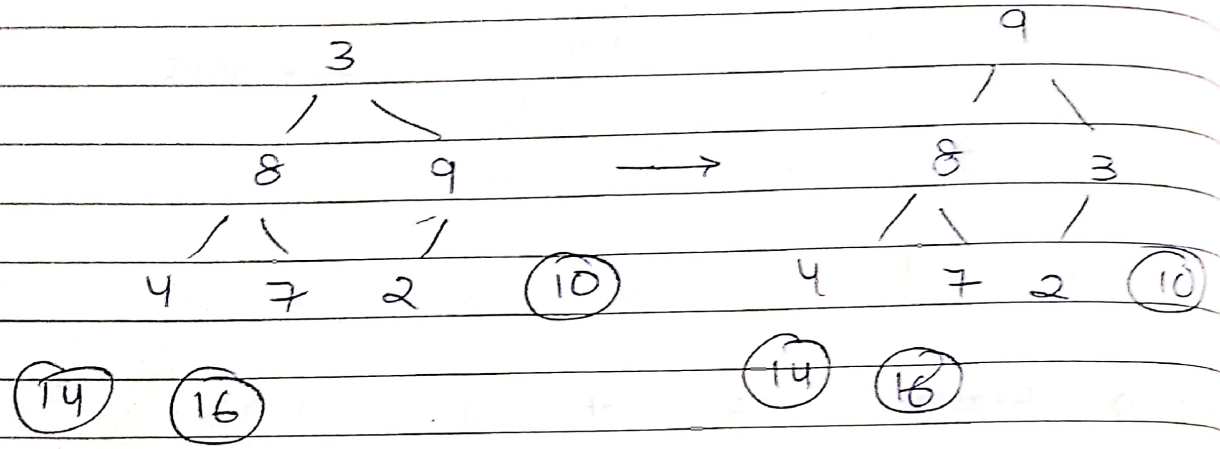


→



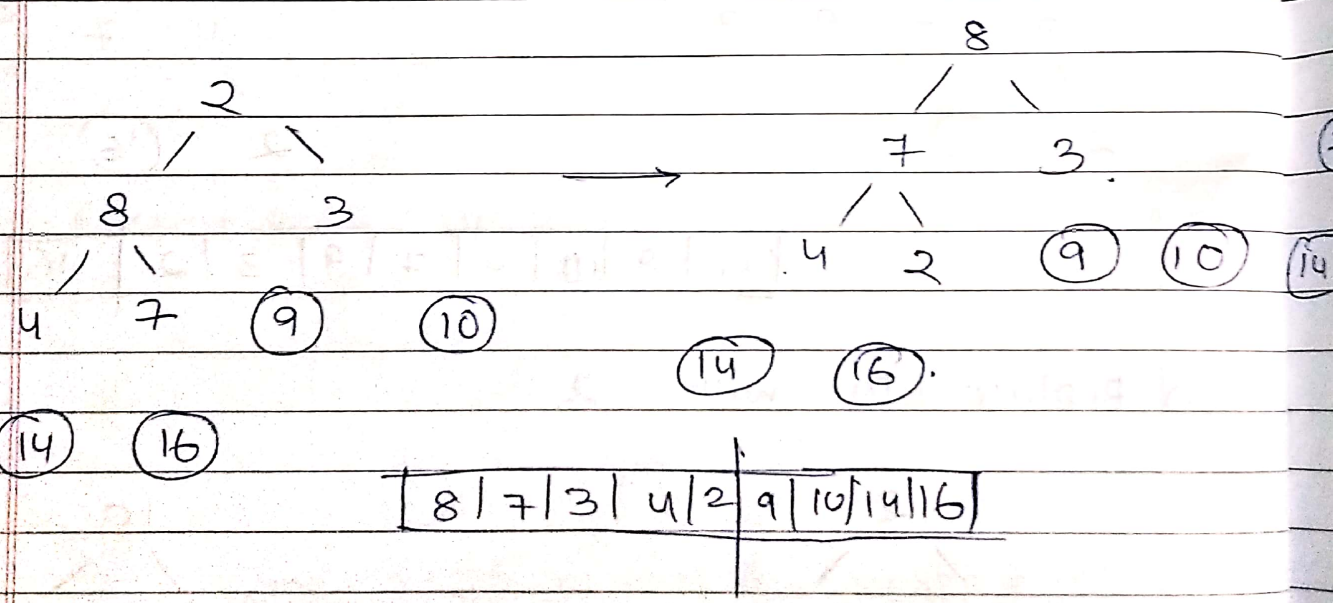
10	8	9	4	7	2	3	14	16

(iii) $10 \longleftrightarrow 3$.



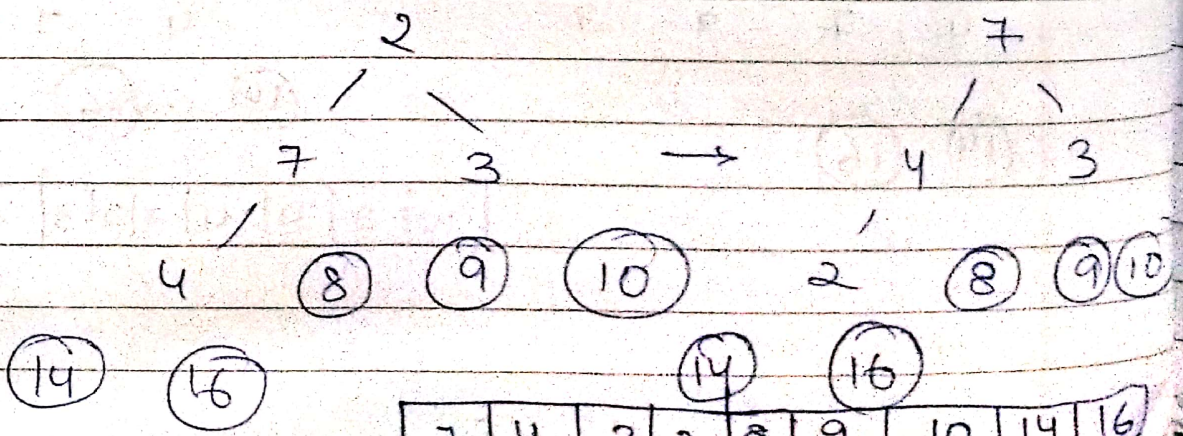
9	8	3	4	7	2	10	14	16

(iv) $9 \longleftrightarrow 2$



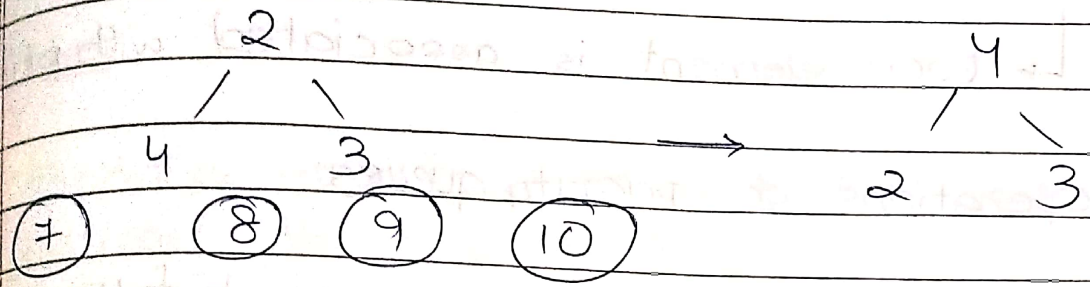
8	7	3	4	2	9	10	14	16

(v) $8 \longleftrightarrow 2$.



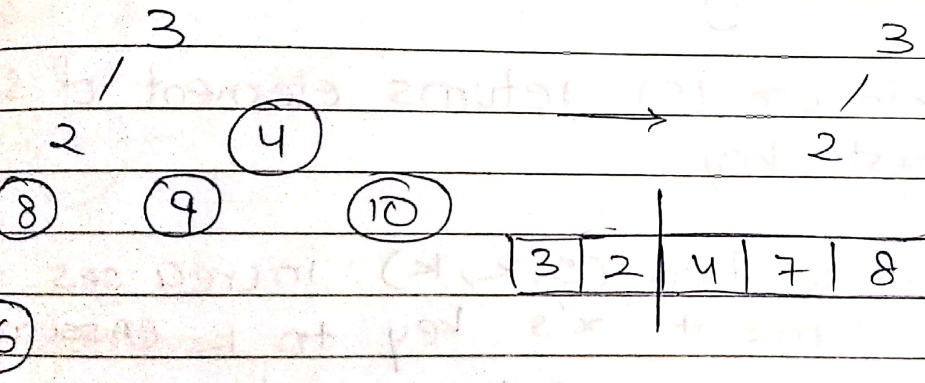
7	4	3	2	8	9	10	14	16

(vi) $7 \longleftrightarrow 2$.



4	2	3	7	8	9	10	14	16

(vii) $4 \longleftrightarrow 3$.



3	2	4	7	8	9	10	14	16

(viii) $2 \longleftrightarrow 3$

2

2	3	4	7	8	9	10	14	16

→ PRIORITY QUEUES:-

↳ Each element is associated with priority.

→ Operations on priority queues:-

Max-priority queues support following operations:

(i) Insert (S, x) : inserts element x into set S.

(ii) Extract-max(S): removes & returns element of S with largest key

(iii) Maximum (S): returns element of S with largest key.

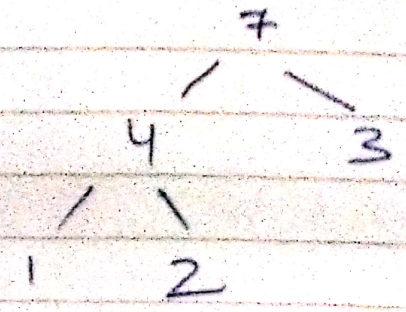
(iv) Increase-key (S, x, k): increases value of element x's key to k (Assume k ≥ x's current key value).

→ HEAP-MAXIMUM:-

Algorithm: HEAP-MAXIMUM (A)

1. return A[1];

e.g. Heap A



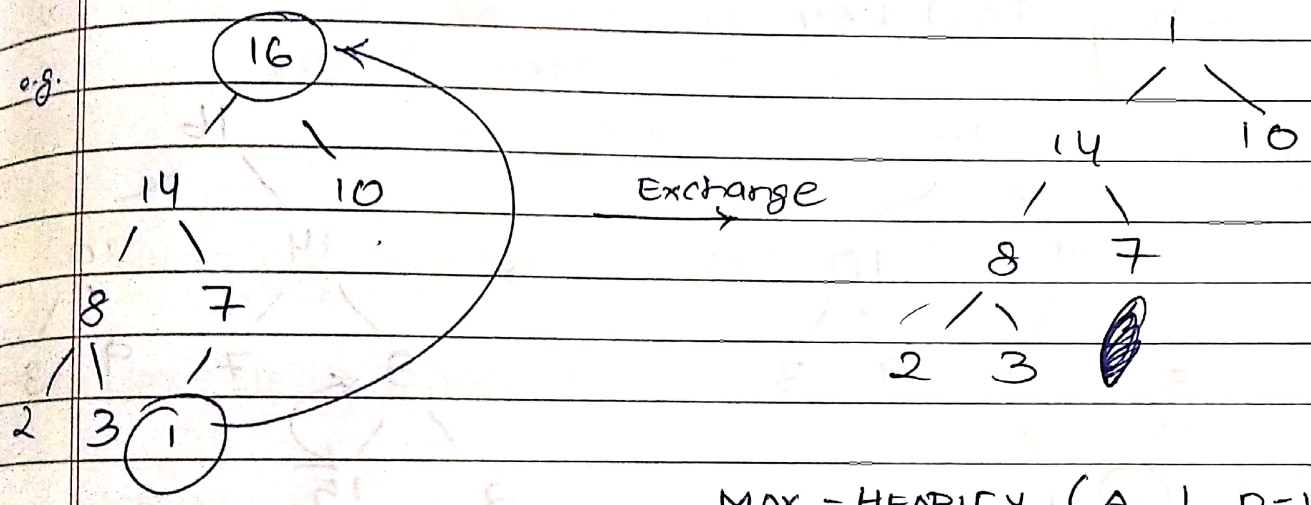
Heap-Maximum(A) returns 7.

HEAP - EXTRACT - MAX:-

Idea:

- Exchange root element with the last
- Decrease the size of heap by 1.
- Call MAX - HEAPIFY on the new root, on a heap of size n-1.

e.g.



MAX - HEAPIFY (A, 1, n-1).

Algorithm:

HEAP - EXTRACT - MAX (A, n)

```
if n < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[n]
MAX - HEAPIFY (A, 1, n-1)
return max.
```

Running - time : $O(\log n)$

↓

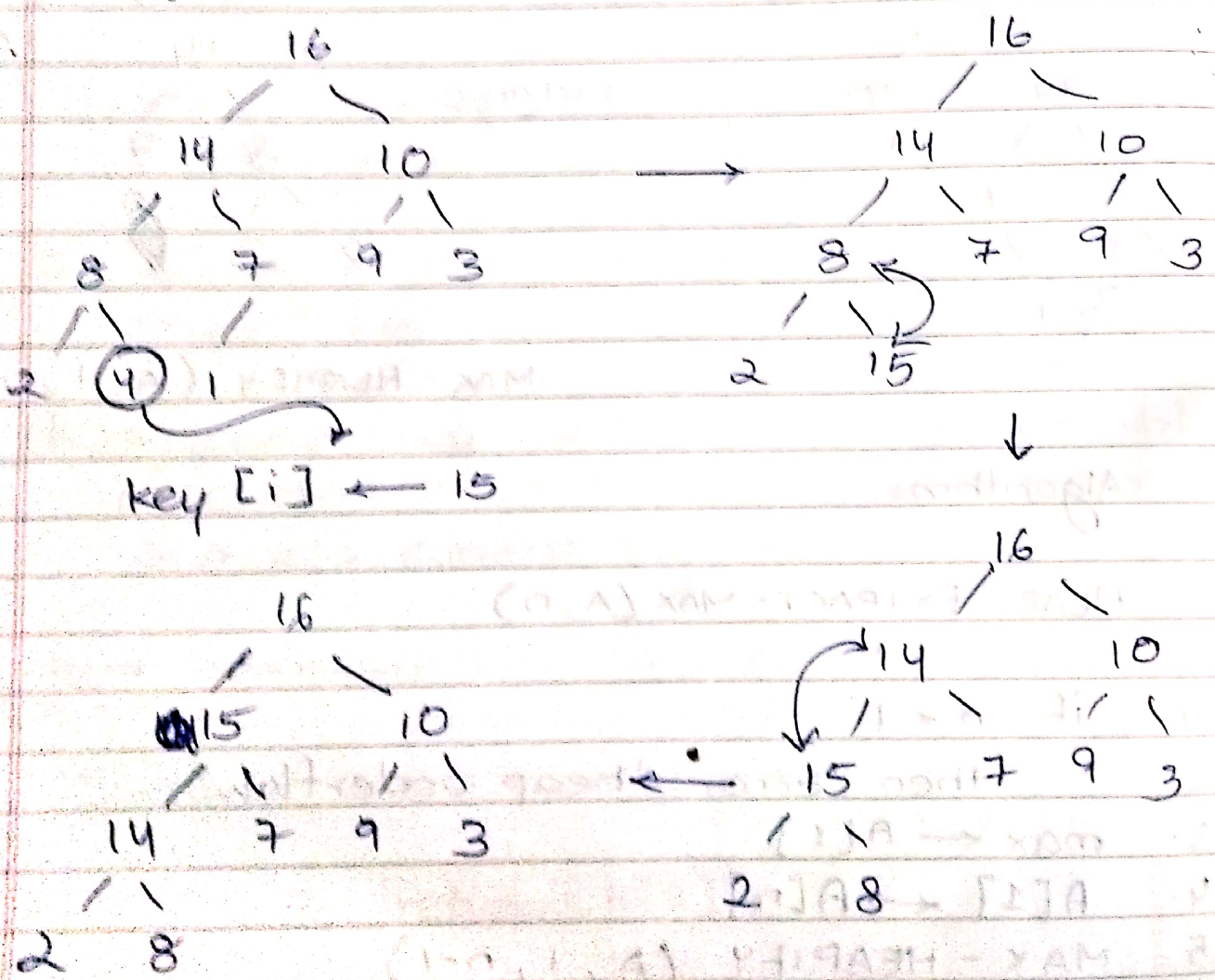
Height of heap.

→ HEAP - INCREASE - KEY :-

Idea :

- Increment the key of A[i] to its new value.
- If the max-heap property does not hold anymore : traverse a path toward the root to find the proper place for the newly fixed key.

e.g.



Algorithm:

HEAP - INCREASE - KEY (A, i, key)

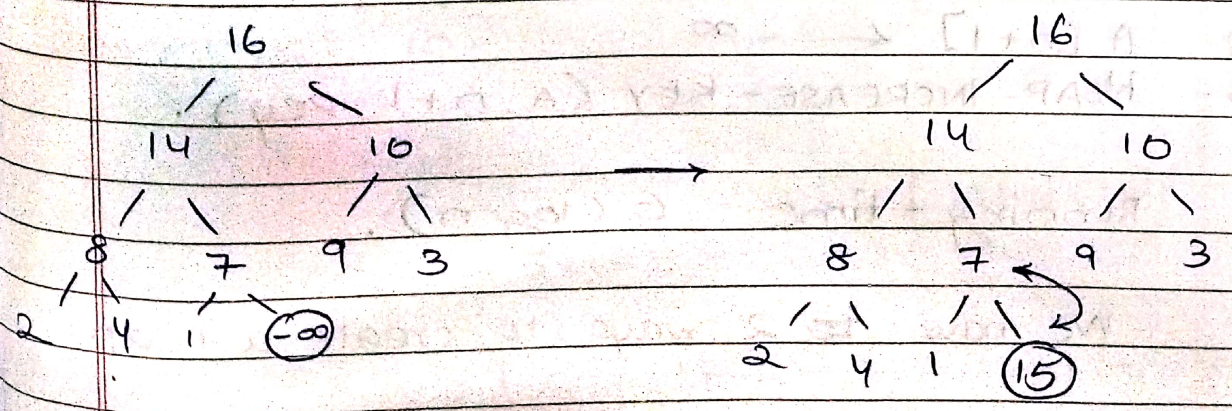
```
if key < A[i]
    then error "new key is smaller than current key".
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do EXCHANGE A[i] ↔ A[PARENT(i)]
       i ← PARENT(i)
```

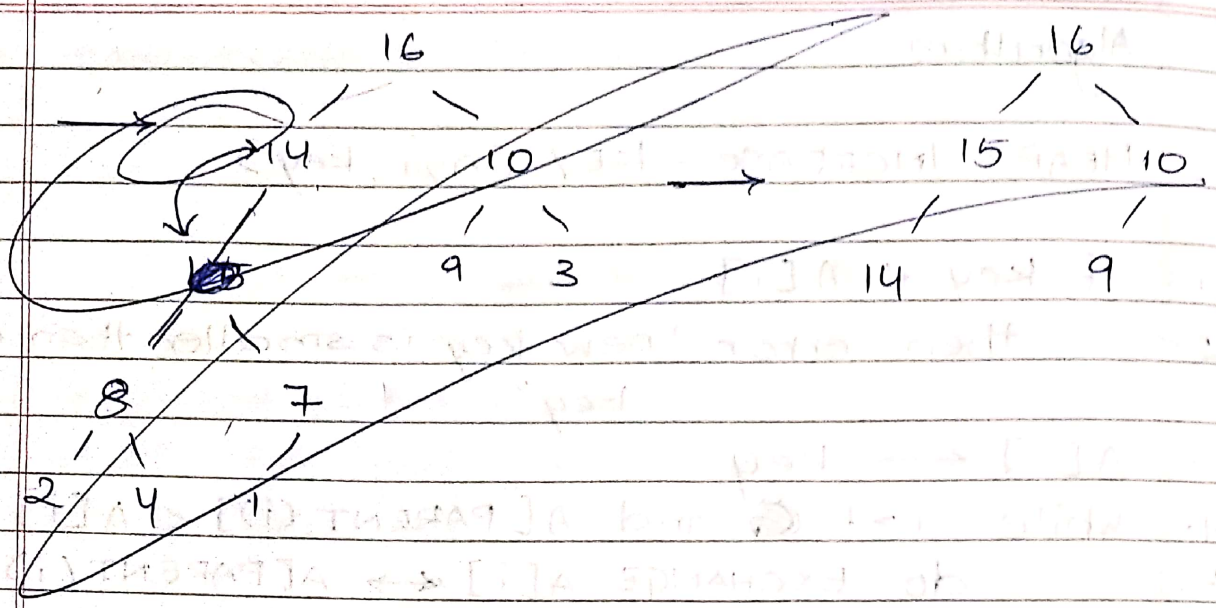
Running - time : $O(\log n)$.

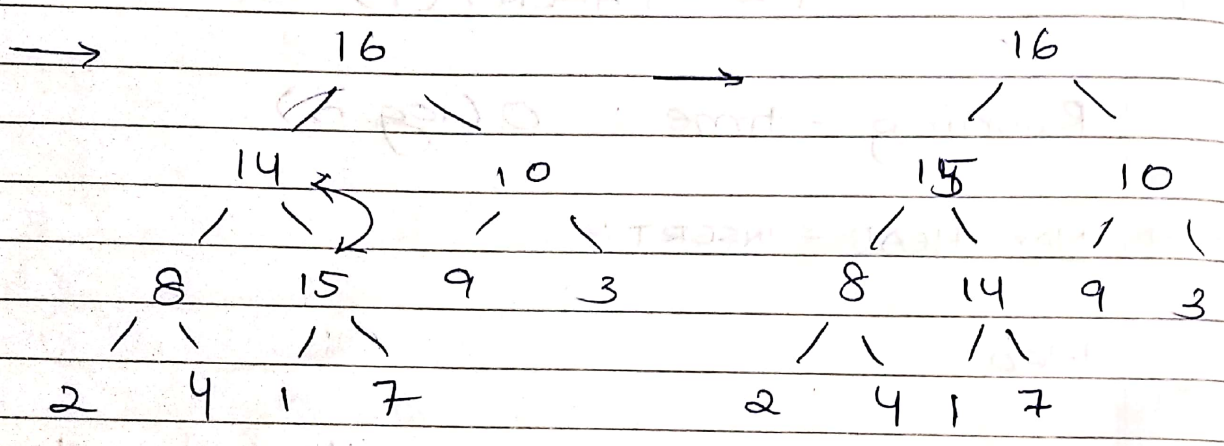
→ MAX- HEAP - INSERT :-

Idea:

- Expand the max-heap with a new element whose key is $-\infty$.
- Calls HEAP - INCREASE - KEY to set the key of the new node to its correct value & maintain the max-heap property.







Algorithm:

MAX-HEAP-INSERT (A, key, n)

1. heap-size [A] ← n + 1;
2. A [n+1] ← −∞
3. HEAP-INCREASE-KEY (A, n+1, key).

Running-time : O (log n).

We have ~~to~~ 2 ways to create a heap:

(i) BUILD MAX HEAP , (ii) SUCCESSIVE INSERTION

Q→ A = [5, 3, 17, 10, 84, 19, 6, 22, 9]

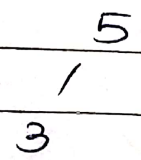
↓

Create max heap by successive insertion.

→ (i) Insert 5.

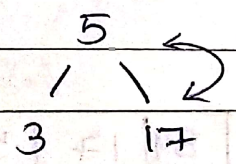
A = | 5 |

(ii) Insert 3.



A = | 5 | 3 |

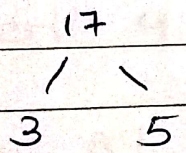
(iii)



heap-increase-key

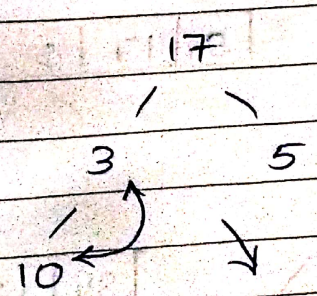
A = | 5 | 3 | 17 |

↓

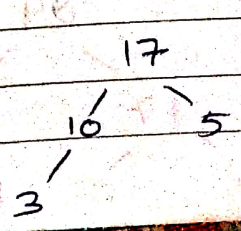


A = | 17 | 3 | 5 |

(iv) Insert 10.

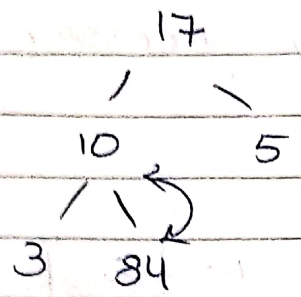


A = | 17 | 3 | 5 | 10 |

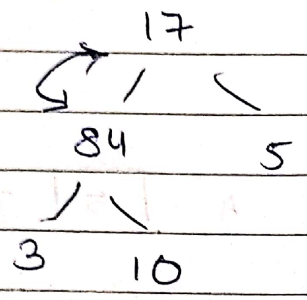


| 17 | 10 | 5 | 3 |

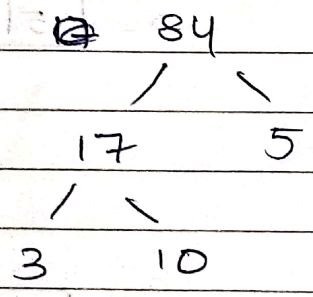
(v) insert 84



17	10	5	3	84

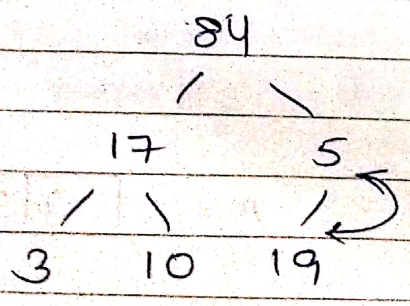


17	84	5	3	10

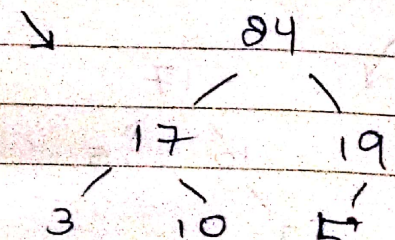


84	17	5	3	10

(vi) Insert 19

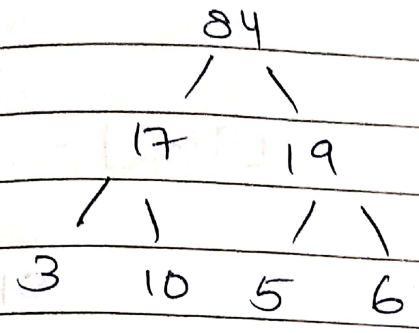


84	17	5	3	10	19



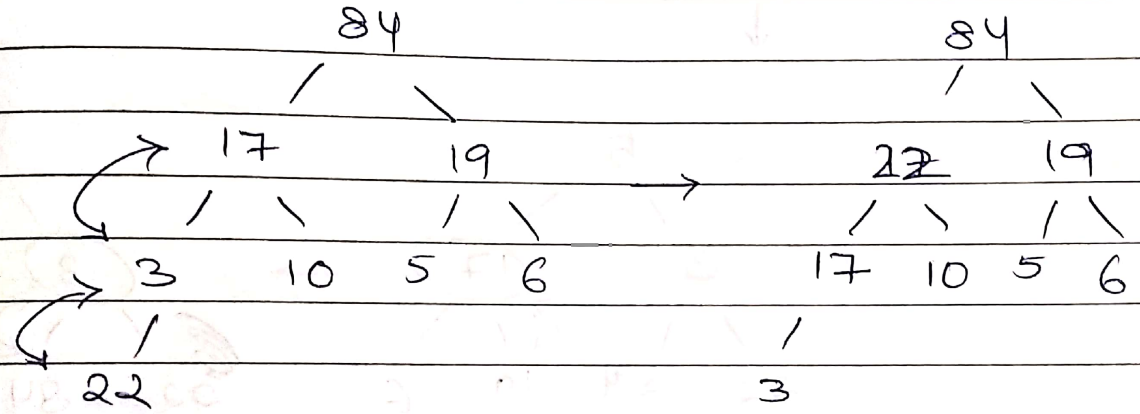
84	17	19	3	10	5

(vii) Insert 6.



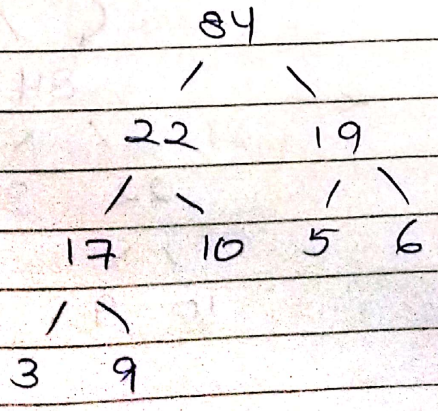
| 84 | 17 | 19 | 3 | 10 | 5 | 6 |

(viii) Insert 22



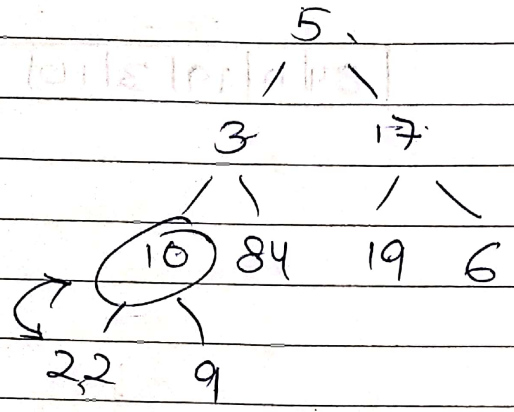
| 84 | 22 | 19 | 17 | 10 | 5 | 6 | 3 |

(ix) Insert 9.



| 84 | 22 | 19 | 17 | 10 | 5 | 6 | 3 | 9 |

Now using build max heap:

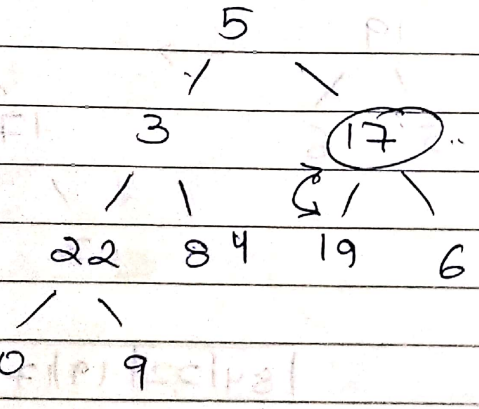


$\lfloor 9/2 \rfloor = 4$

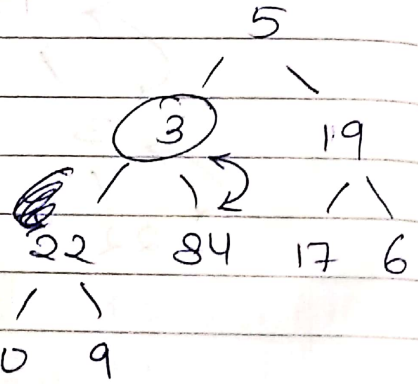
↓

start from 4

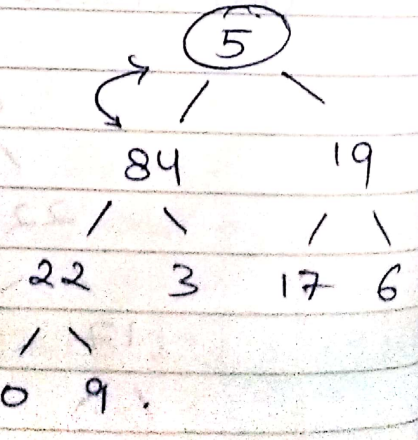
↓



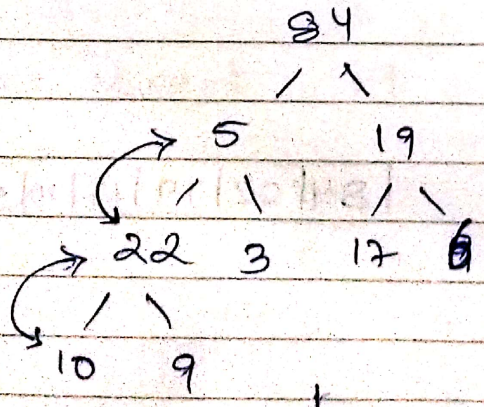
→



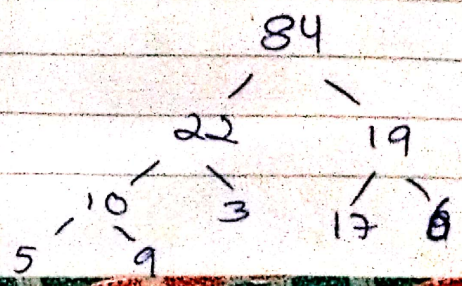
↓



←



↓



84	22	19	10	3	17	6	5

These are min priority heaps.

FIBONACCI HEAPS:-

Binomial heaps support the mergeable heap operations (INSERT, MINIMUM, EXTRACT_MIN, UNION plus, DECREASE_KEY & DELETE) in $O(\log n)$ ~~worst~~ in ~~worst~~ $O(\log n)$ worst case time.

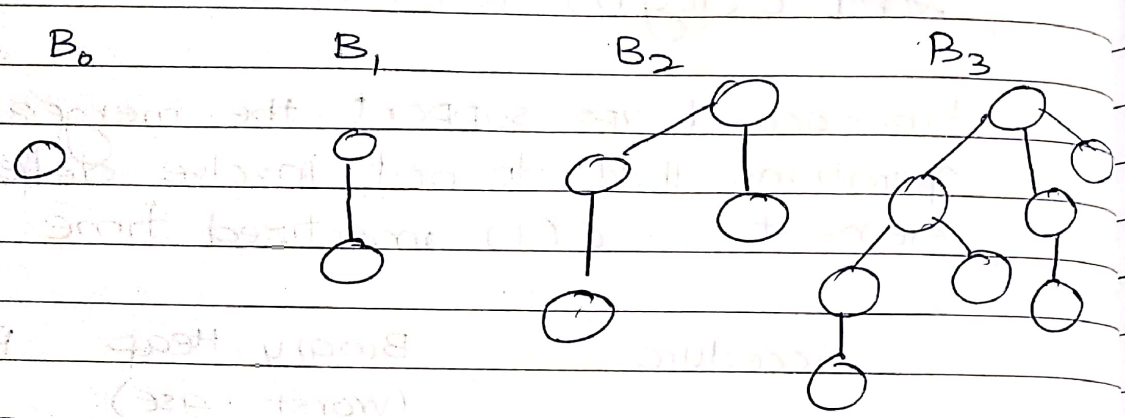
Fibonacci heaps support the mergeable heap operations that do not involve deleting an element in $O(1)$ amortized time.

Procedure	Binary Heap (worst case)	Fibonacci Heap (amortized)
make-heap	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(1)$
Minimum	$O(1)$	$O(1)$
Extract-min	$O(\log n)$	$O(\log n)$
Union	$O(n)$	$O(1)$
Decrease-key	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$

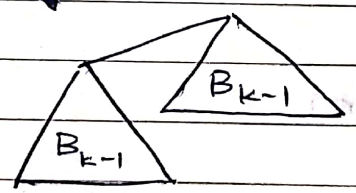
Same like use key in max heap, we have use key in min heap

Fibonacci heap is a loose version of Binomial Heap.

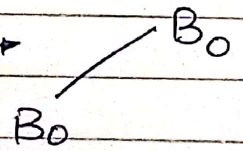
→ **Binomial Heaps:** → min priority heaps.



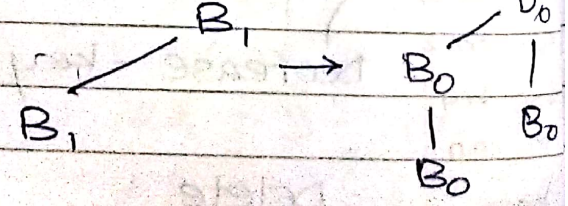
Binomial Heap B_k is a collection of



When $k=1$, B_1 →



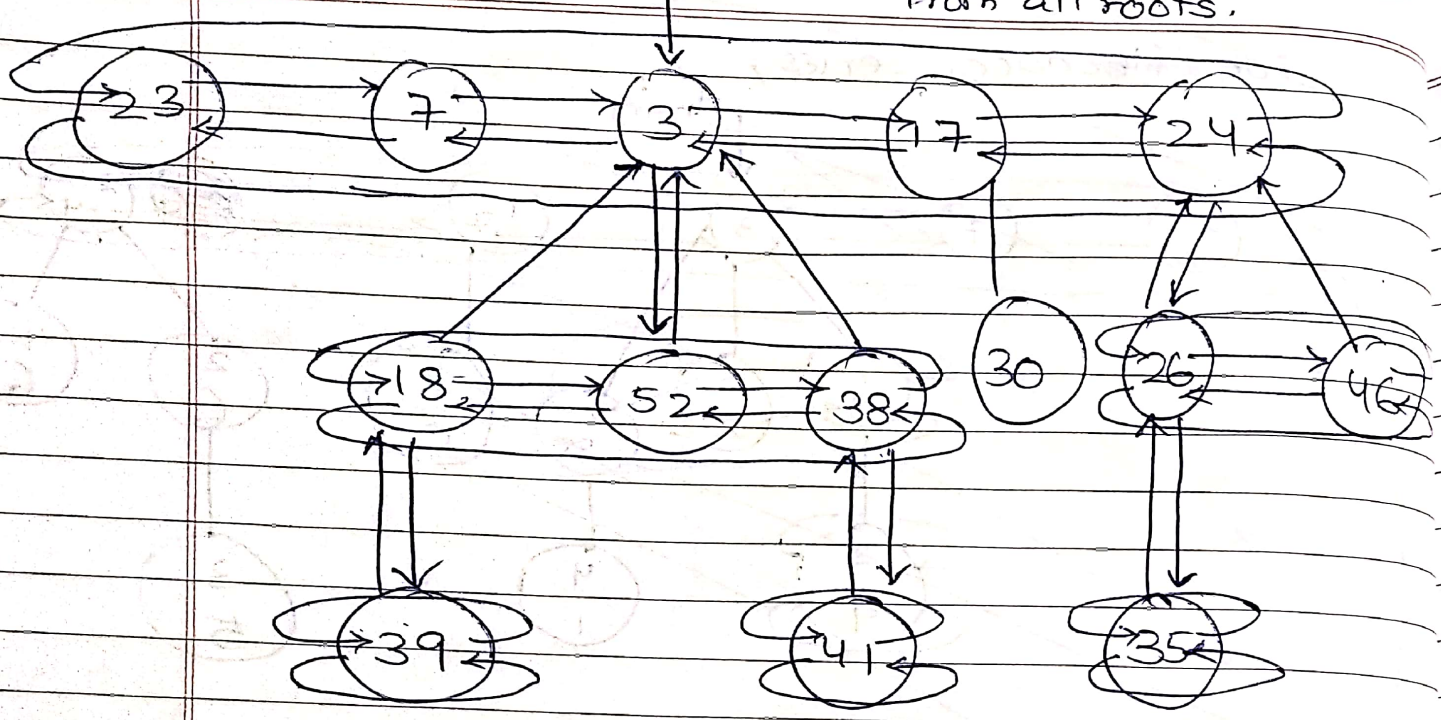
When $k=2$, B_2 →



& so on . . .

Root of final tree will be minimum value among roots of the 2 trees.

Fibonacci Heap



- All root nodes wrapped in same circular doubly L.L.
- All siblings (having same parent) are wrapped in same circular doubly L.L.
- All leaf nodes are wrapped in seperate circular doubly L.L.
- All nodes (except for root) have 4 pointers

(a) parent
(b) child
(c) right sibling
(d) left sibling.

Marked & Unmarked :-

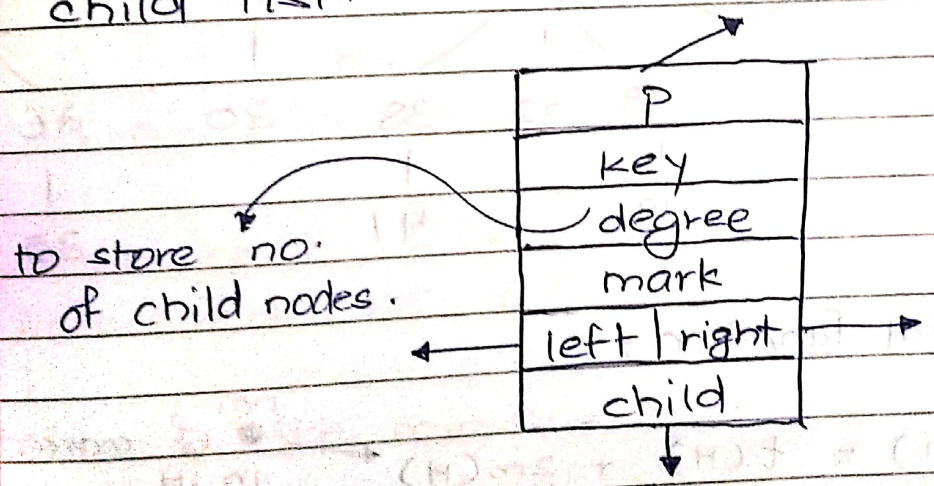
↳ if a node has a child and if the child gets deleted, then the node is marked.

STRUCTURE OF FIBONACCI HEAPS :-

Each node x contains:

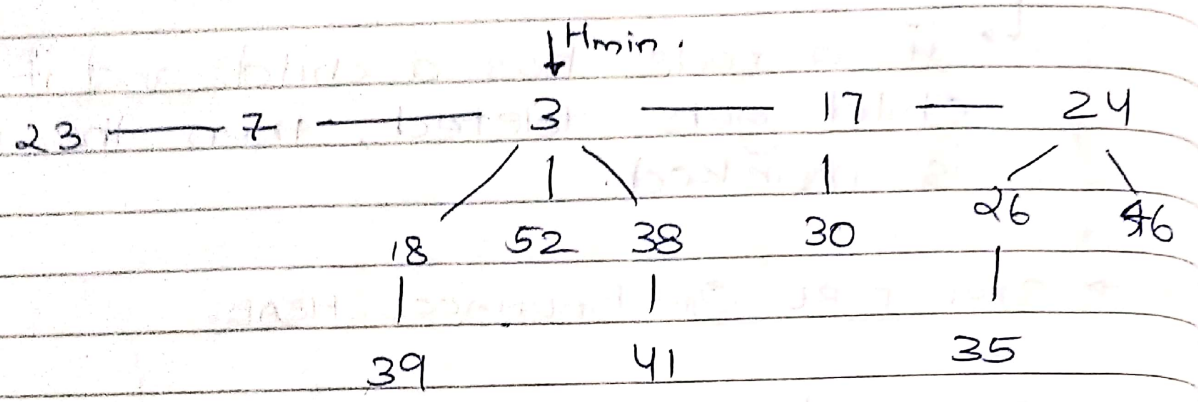
- A pointer $p[x]$ to its parent.
- A pointer $child[x]$ to one of its children

- The children of x are linked together in a circular, doubly L.L which is called the child list.



If y is an only child then, $left[y] = right[y] = y$

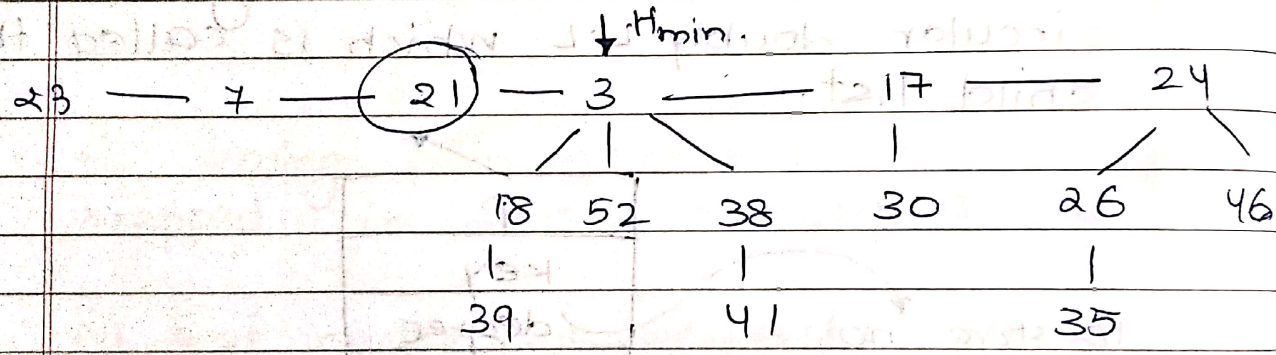
→ Insertion :-



Insert key 21

↓

new node is inserted in root list as left sibling of Hmin. & update the Hmin accordingly



Potential function :

$$\phi(H) = t(H) + 2m(H)$$

$\phi(H)$ ↓ Fibonacci Heap.

$t(H)$ ↓ no. of trees in the rooted list of H.

$m(H)$ ← no. of marked nodes in H.

D(n) = upperbound on the max degree of any node in an n-node Fibonacci Heap.

H.n = no. of nodes in Heap H.

→ Mergeable-heap operations:

Create a Fibonacci-Heap.

Algorithm:

Fib-Heap-Insert (H, x)

```
x.degree = 0
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == NIL
        create a root list for H containing just x
        H.min = x
else insert x into H's root list
        if x.key < H.min.key
                Hmin = x.
H.n = H.n + 1;
```

Actual cost : O(1)
Amortized cost : O(1) + 1.

Finding the minimum node:

H.min - O(1) Amortized cost O(1)
ϕ is not changed.

Uniting 2 Fibonacci Heaps :-

Algorithm:

$H.min = NIL$
$H.n = 0$

Fib-Heap-Union (H_1, H_2)

1. H = Make-Fib-Heap( )
2. $H.min = H_1.min$
3. concatenate the root list of H_2 with root list of H.
4. if $(H_1.min == NIL)$ or $(H_2.min \neq NIL$ & $H_2.min.key < H_1.min.key)$
5. $H.min = H_2.min$
6. $H.n = H_1.n + H_2.n$
7. return H.

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2))$$

$$= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2)))$$

$$= 0$$

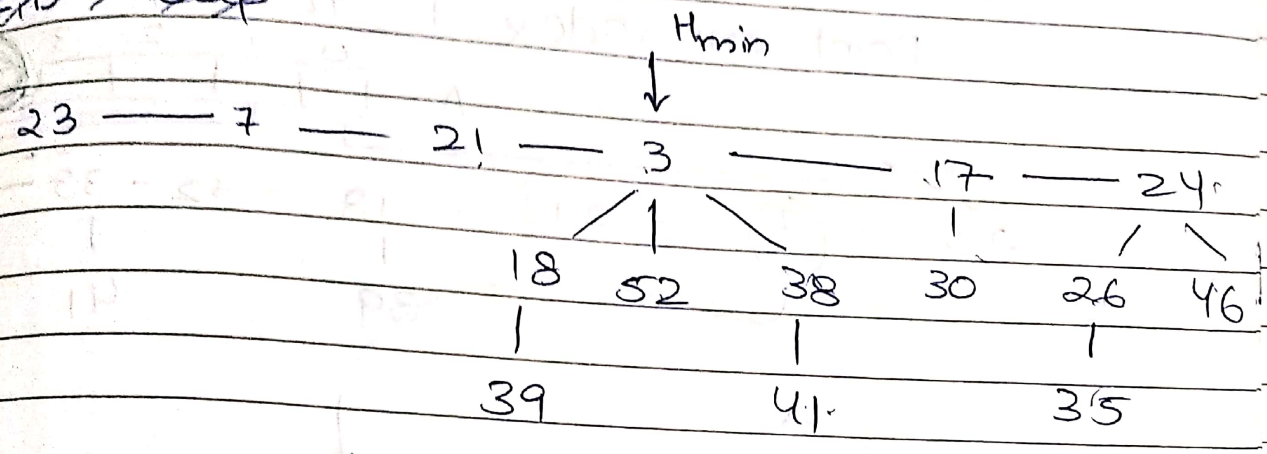
$\therefore$ $t(H) = t(H_1) + t(H_2)$, $m(H) = m(H_1) + m(H_2)$

~~Thus the amort~~

Extracting the minimum node:-

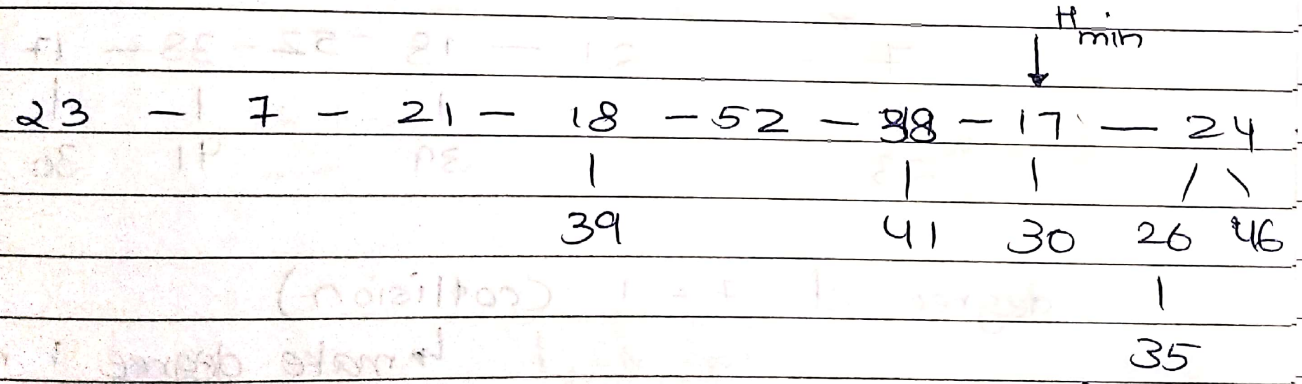
~~Fib - Heap~~

e.g.



Step i) delete (Hmin)

↳ children of Hmin get added to the root list. left sibling of Hmin = left sibling of children
right " " " = right " " "



Step (ii) Create an empty array of size D(n)

↳ degree of heap.

$D(n) = \lfloor \log n \rfloor$
$= \lfloor \log_2 15 \rfloor$
$= \lfloor 3.9\ldots \rfloor$
$D(n) = 3$

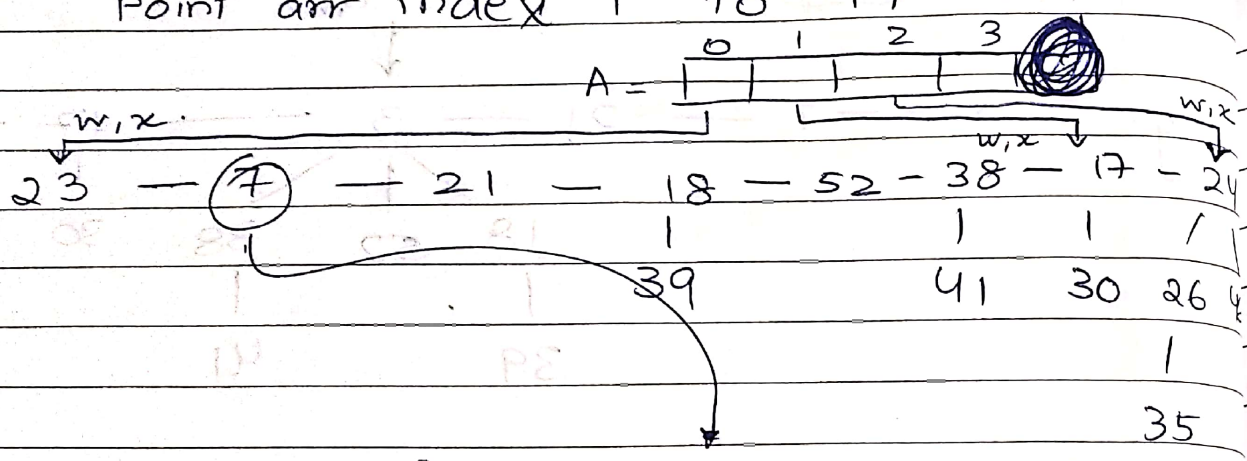
A = | 0 | 1 | 2 | 3 |

Now start with degree of Hmin & move towards right

Point index of A which is same as degree of Hmin to Hmin.

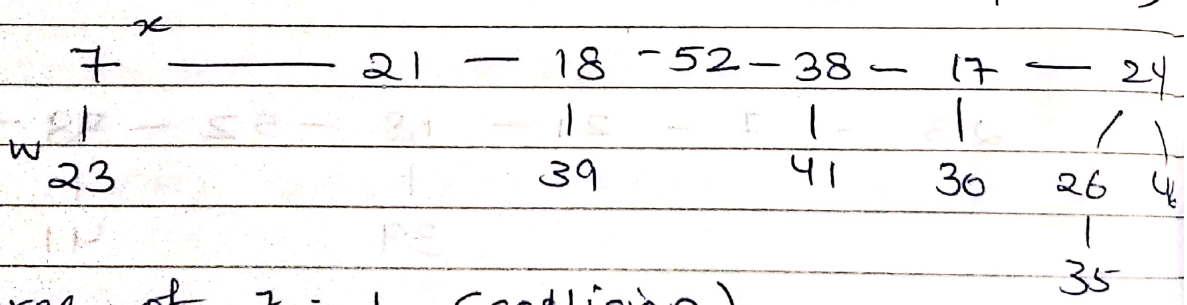
eg. Degree of 17 = 1

Point arr index 1 to 17.



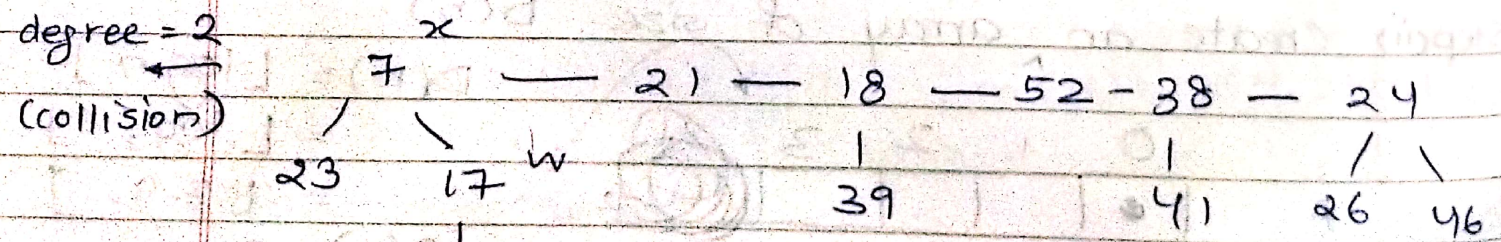
Now collision for degree 0 at node 7, since degree 0 already occupied by 23

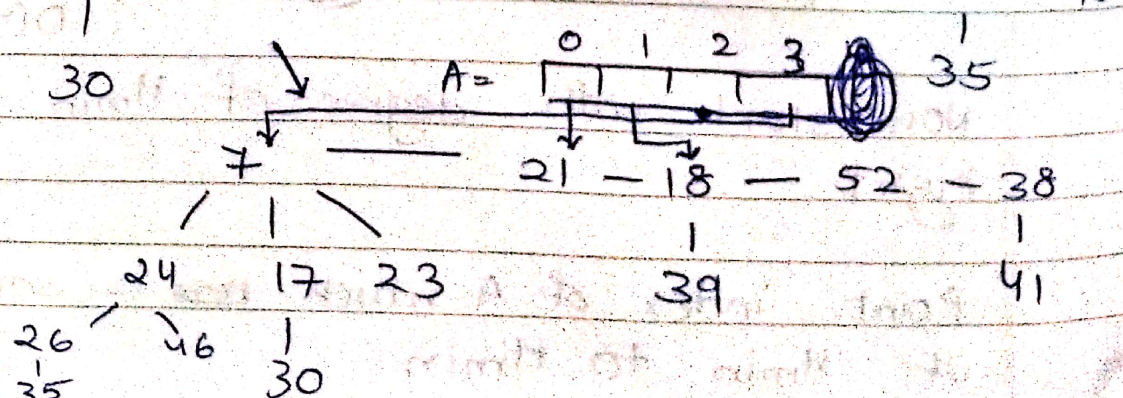
∴ Make 23 child of 7 & assign degree to 7 accordingly. ↳ ∵ 7<23 (ie make lower value as parent)



degree of 7 = 1 (collision)

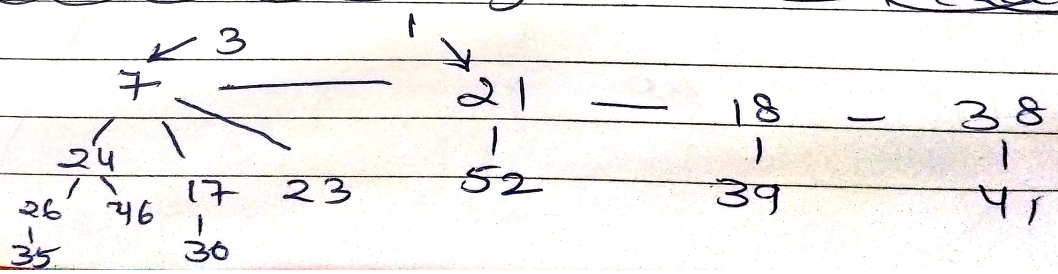
↳ make degree 1 node as child of 7.





after adding 21 collision → ∵ degree of 52 = 0 (occupied)

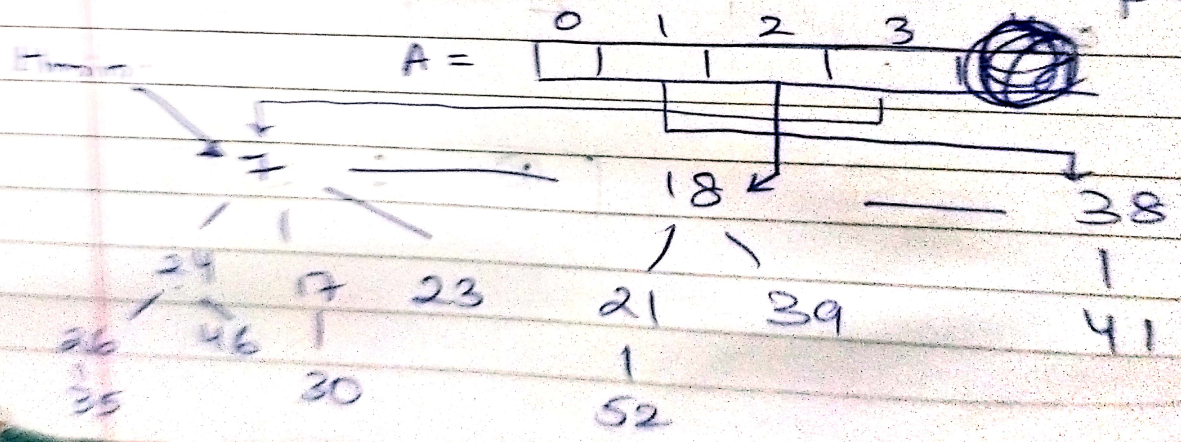
~~collision → degree of 21 = 1 (occupied)~~



P.T.O.

Degree of 18 = 1 (collision) & 18 < 21

parent



Algorithm:
Fib-Heap-Extract-Min (H)

```
z = H.min
if z ≠ NIL
    for each child x of z
        do { add x to the root list of H
             x.p = NIL }
        remove z from the root list of H.
if z == z.right
        H.min = NIL
else H.min = z.right
Consolidate (H)
H.n = H.n - 1
return z.
```

→ Decrease priority key:

(a) if priority key of leaf node is to be decreased:

↓

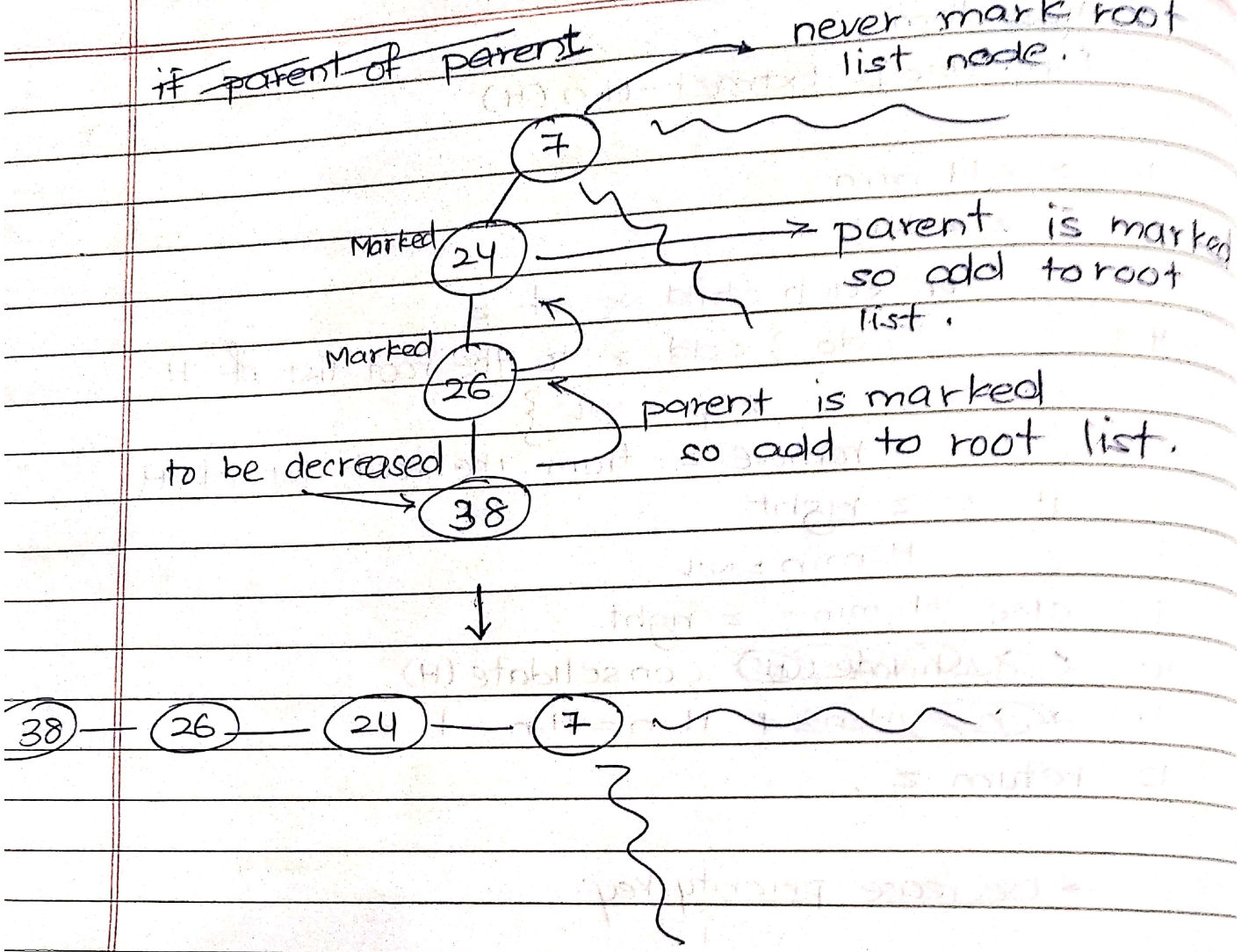
(i) remove the leaf node. → simple cut
(ii) mark the parent of the leaf node (if parent unmarked)
(iii) change key of leaf node
(iv) Add key in root list.

cascade cut.

If parent of leaf node is already marked:

(i) Remove the marked parent node & place it in the root list.
(ii) Unmark that node (since, all nodes in root list are unmarked)

~~if parent of parent~~




```
Fib-Heap-Decrease-key(H, x, k)
if k > x.key
    error "new key is greater than current key"
x.key = k
y ← x.p
if y ≠ NIL & x.key < y.key
    {CUT(H, x, y)
    CASCADING-CUT(H, y)}
if x.key < H.min.key
H.min = x
```

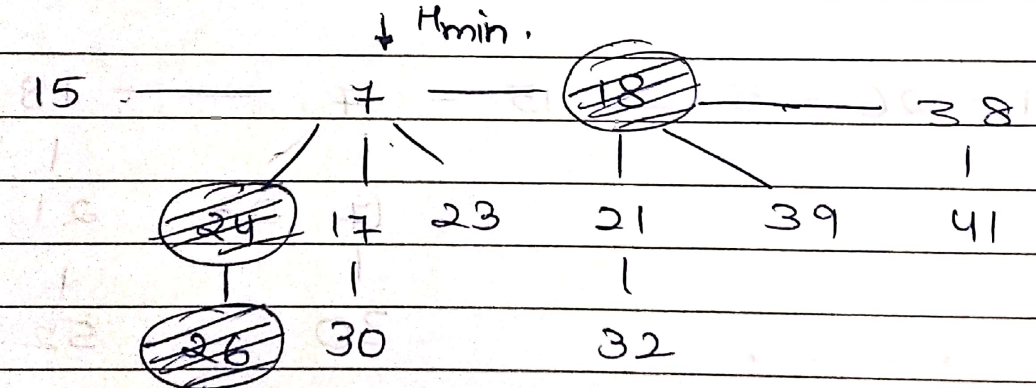
CUT (H, x, y)

1. remove x from child list of y, decrease y.degree
2. add x to root list of H.
3. x.p = NIL
4. x.mark = FALSE

CASCADING - CUT (H, y)

1. z ← y.p
2. if z ≠ NIL
   if y.mark == FALSE
      y.mark = TRUE
   else CUT (H, y, z)
      CASCADING-CUT (H, z)

e.g.



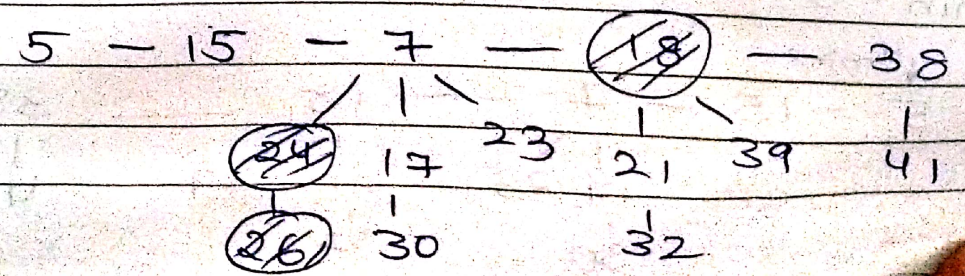
→ marked nodes.

Decrease 35 to 5.

→ Step (i) remove 35. CUT (35)

35 to 5.

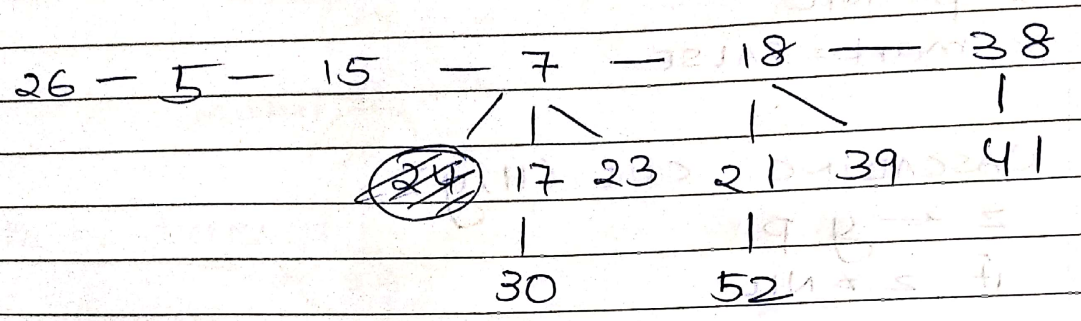
add to root list.



Step (ii) ~~parent~~ cascading-cut(26)

↓

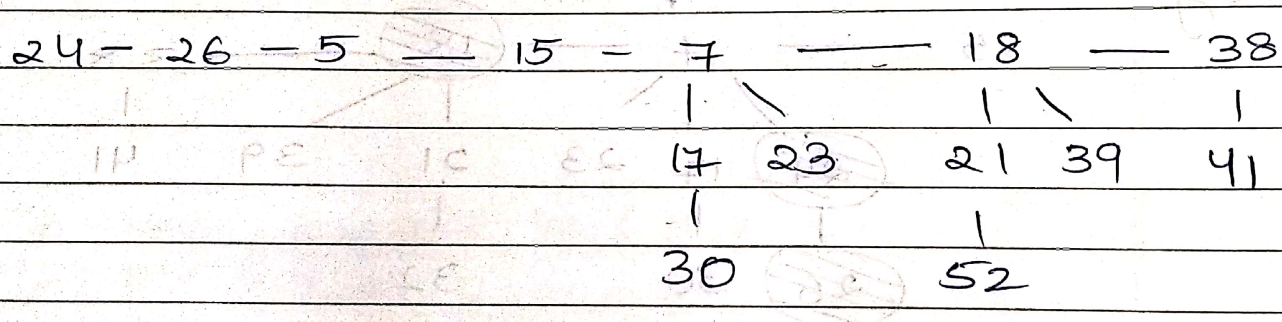
26 → marked → ∴ CUT (26)



Step (iii) cascading cut (24)

↓

24 → marked → ∴ CUT (24)

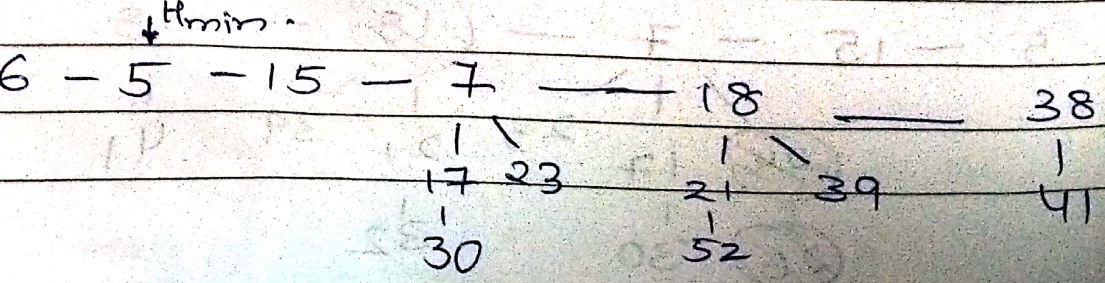


Step (iv) y = 7 & z ← 7.p ∴ z = NIL

∴ ~~Function~~ CUT & CASCADING-CUT stack called back,

& since 5 < 7

∴ Hmin = 5



2

→ For decreasing the key of an internal node,

Case (i): If the ~~parent of that~~ parent of the internal node is unmarked:

Remove

~~Take~~ the entire subtree (including the internal node) & add it to the root.

Case (ii): If the parent of the internal node is marked.

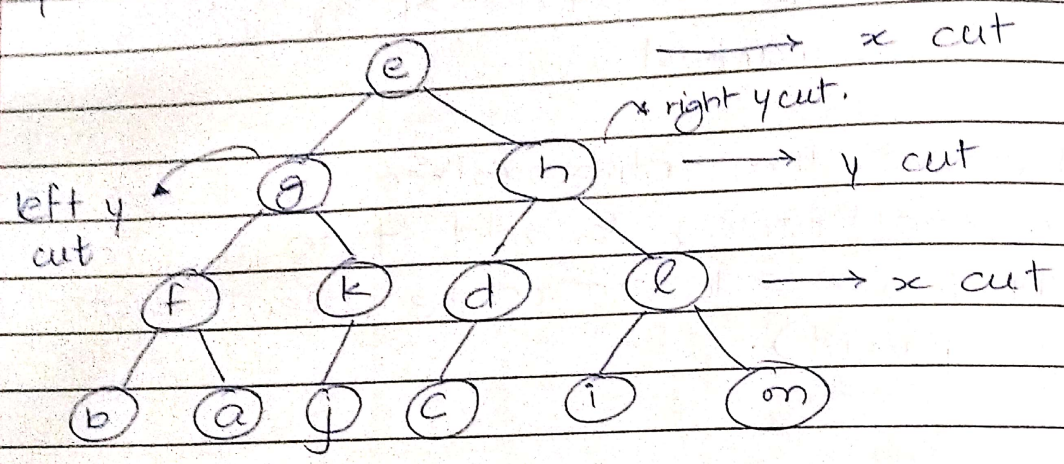
Remove the entire subtree (including the internal node) & add it to the root. The apply cascade cut on the parent subtree (includin the parent).

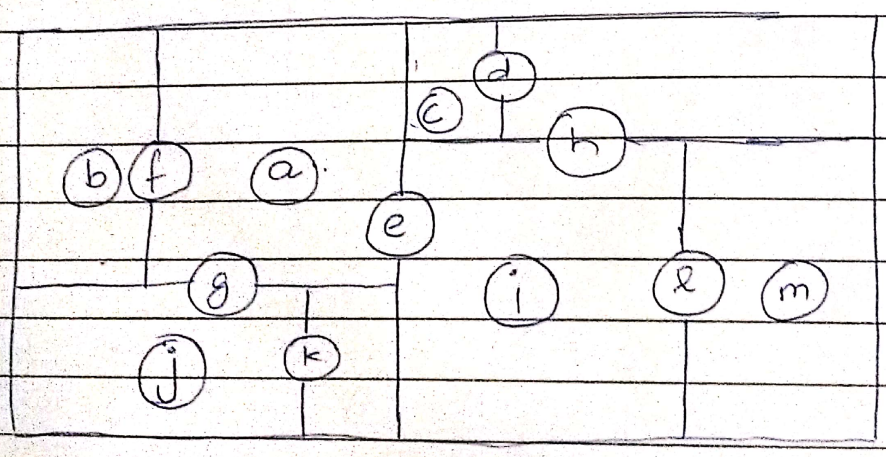
K-D TREES

→ K - dimensional trees.

KD Tree is a binary search tree where every node is a k-dimensional point.

Each level in KD Tree represents either x-cut or y-cut.



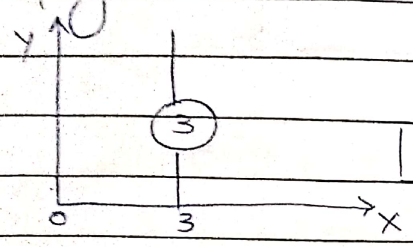


→ Creation of a 2-D Tree:

k = 2.

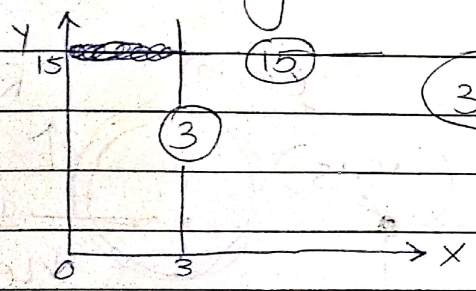
Insert (3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19).

Step (i) : Insert (3,6) as root node since tree is empty.



| 3, 6 | ⟶ x - cut.

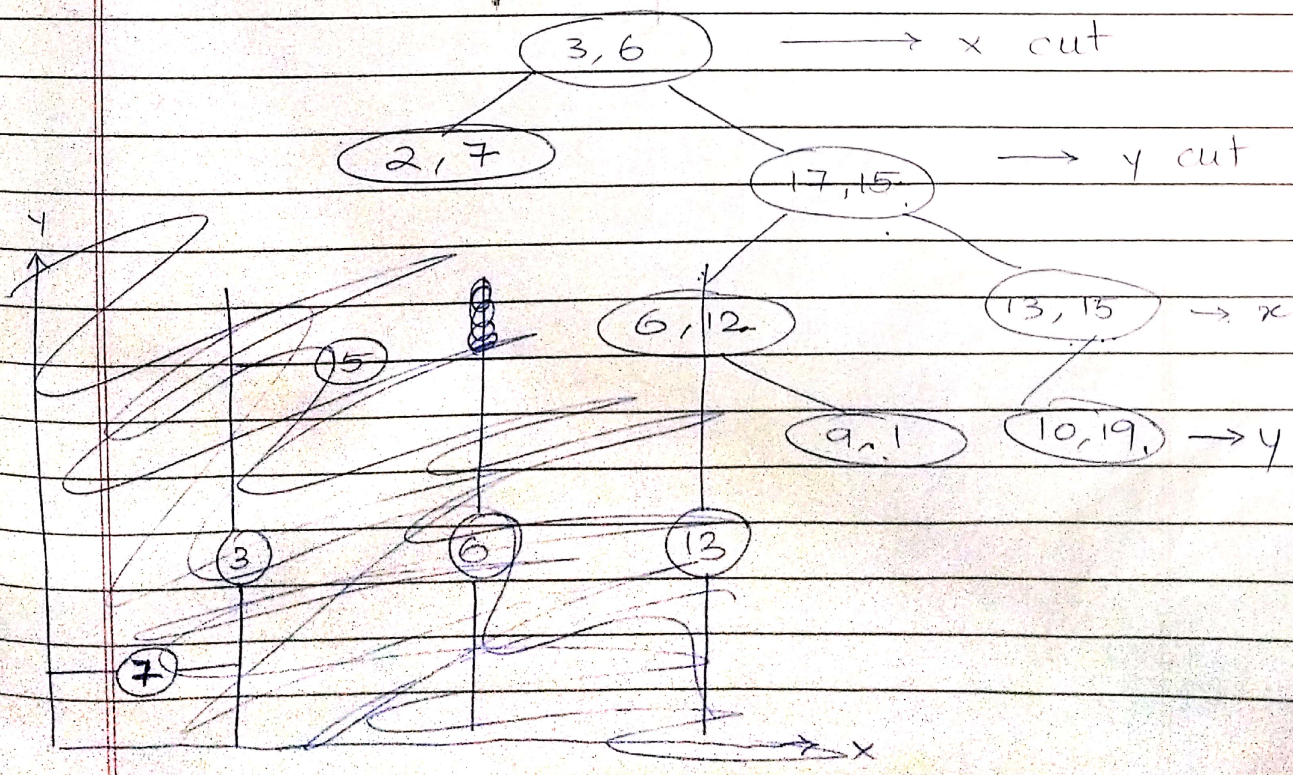
Step (ii) : Insert (17,15). Since parent node has x-cut, compare only x-coordinates. ∴ 17 > 3 ∴ (17,15) gets inserted in right subtree.

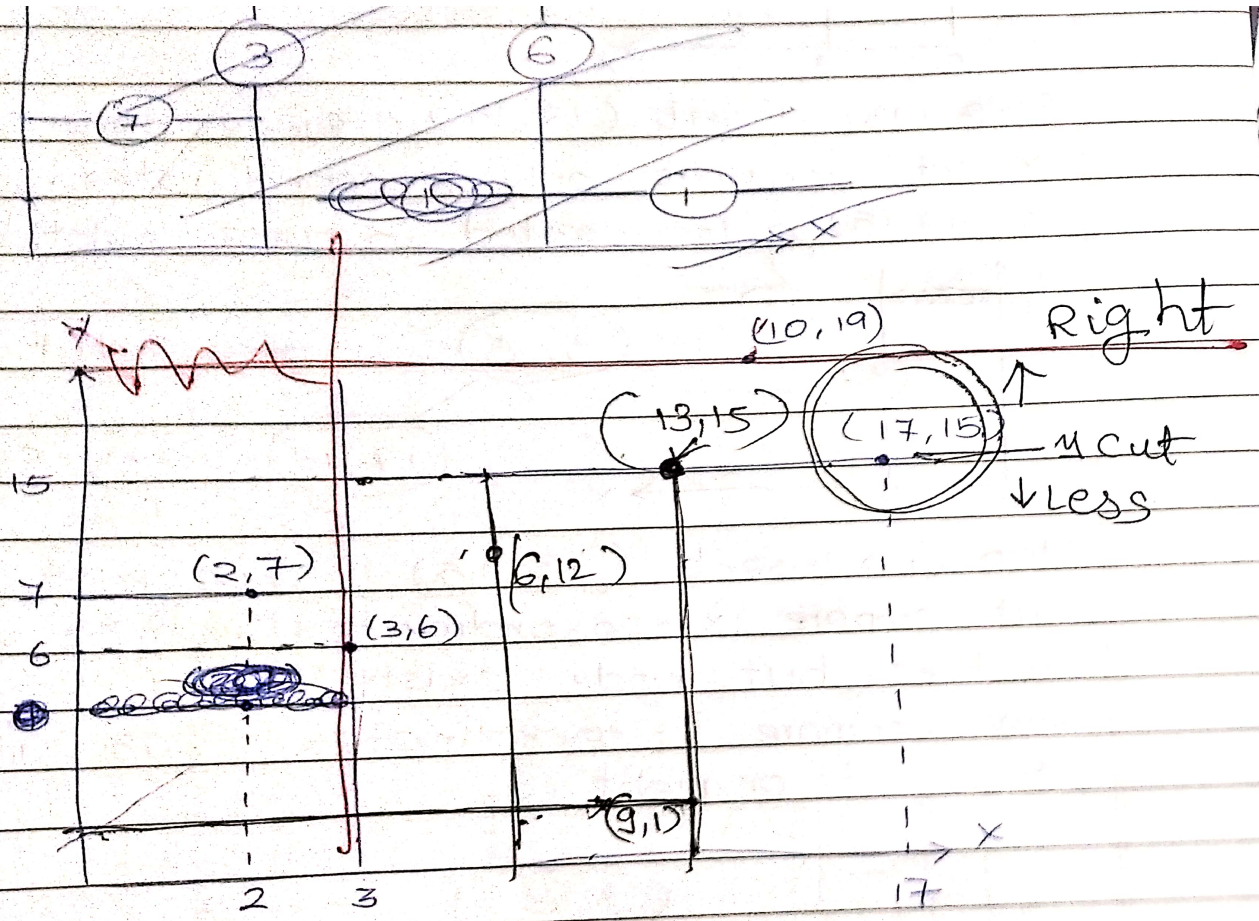


3, 6 ⟶ x-cut

17,15 ⟶ y cut.

Step (iii): Insert (13,15). Since root has x-cut compare x-co-ordinates ∴ 13 > 3 ∴ right subtree. but node exists & has y-cut. ∴ Now compare y-coordinates ∵ 15 = 15 ∴ add to either left or right.

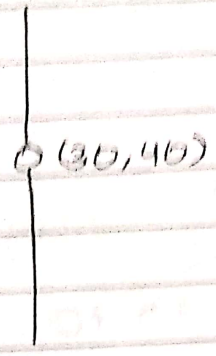



(10, 19)
Right
(13,15)
(17,15)
↑ cut
↓ Less
15
7
6
(2,7)
(3,6)
(6,12)
(9,1)
2
3
17
x
y

3

e.g. insert (30,40)

(30,40)

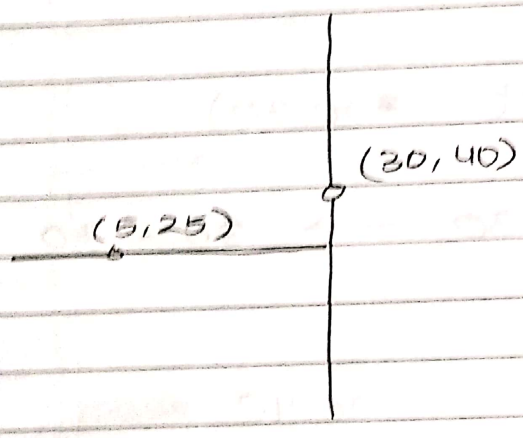


insert (5,25)

5 < 30

∴ 30,40

5,25



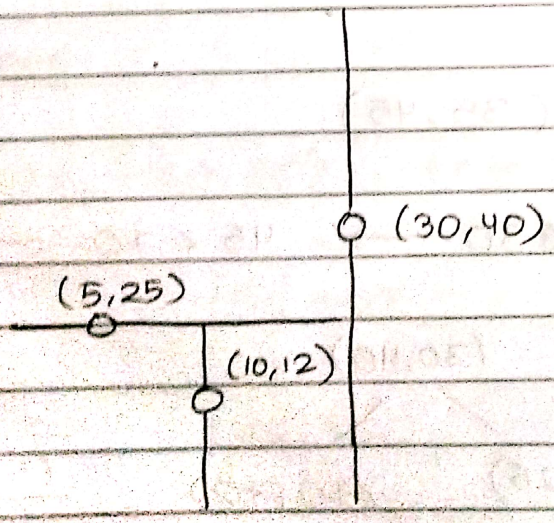
insert (10,12)

10 < 30 → 12 < 25

∴ 30,40

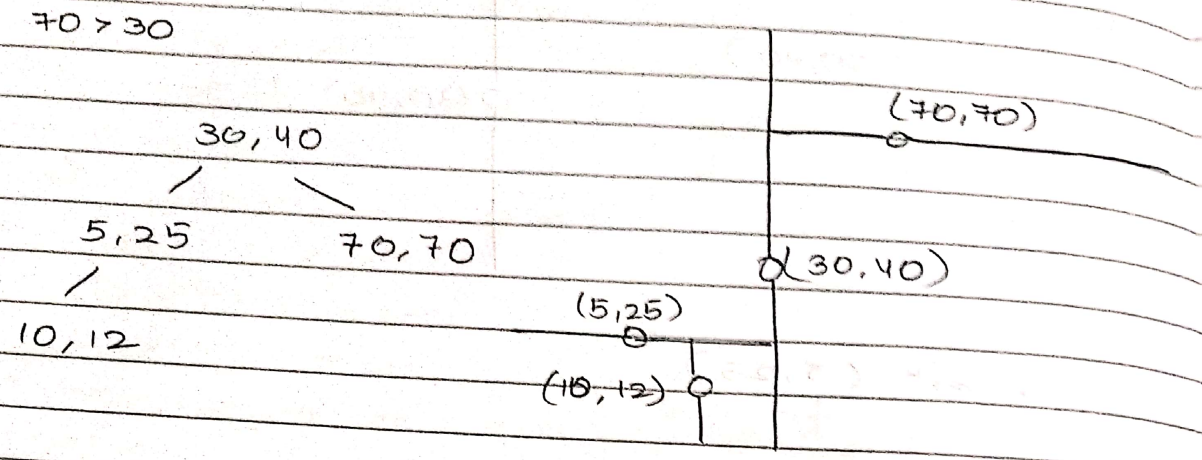
5,25

10,12



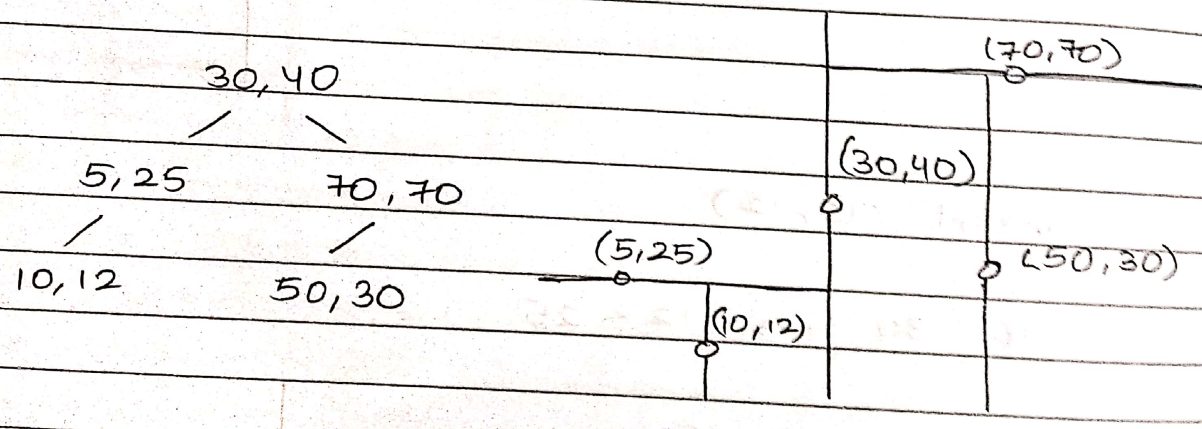
insert (70,70)

70 > 30



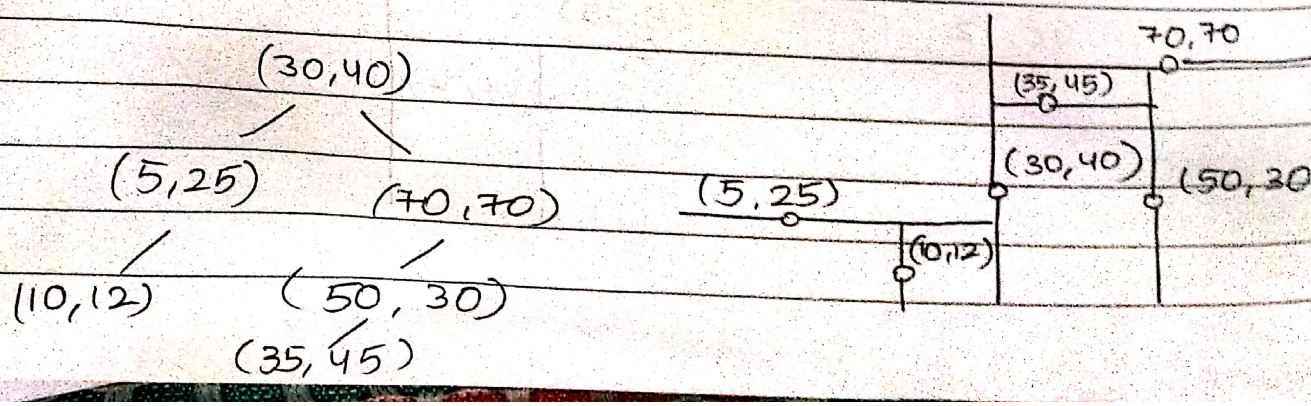
insert (50,30)

50 > 30 → 30 < 70

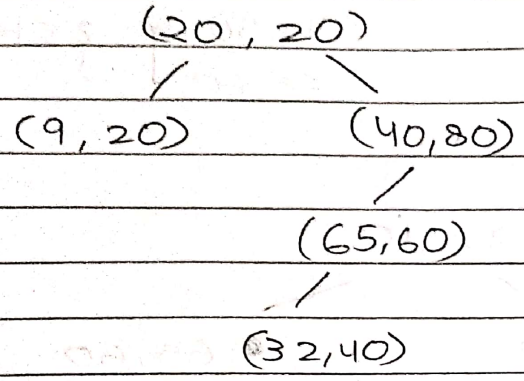


insert (35,45)

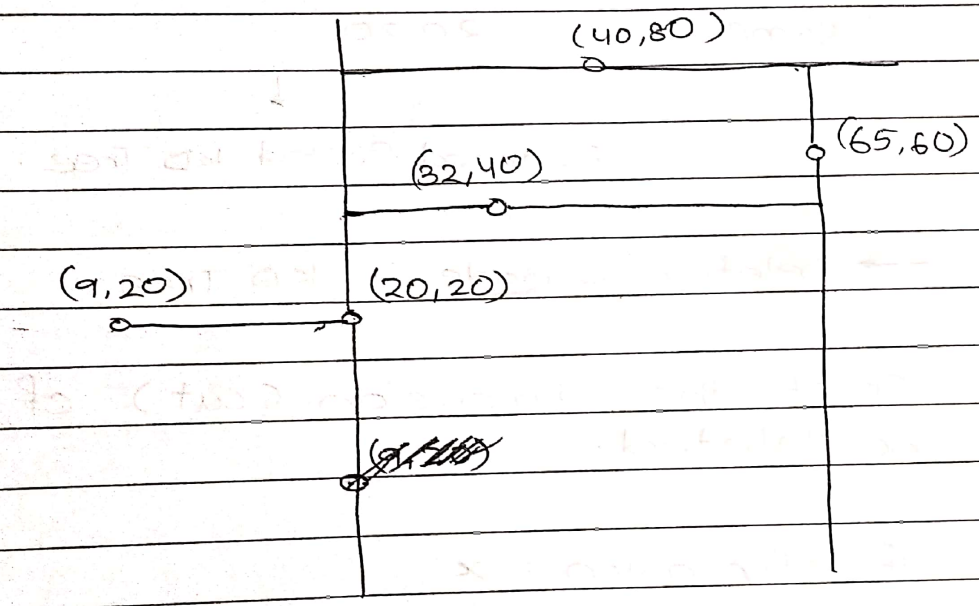
35 > 30 → 45 < 70 → 35 < 50.



e.g insert (20,20), (9,20), (40,80), (65,60), (32,40)



1) ste.

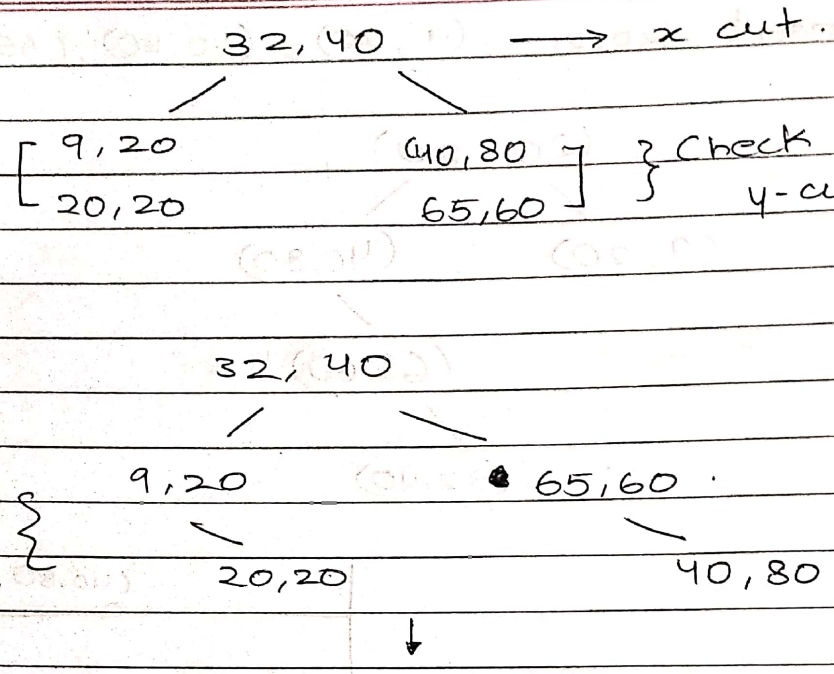


~~Above~~ all the above egs are of ~~unsorted~~ unbalanced KD Trees.

→ ~~sorted~~ Balanced KD Trees:

(20,20), (9,20), (40,80), (65,60), (32,40)

Sort according to x-coordinate → (9, 20)
(20,20)
(32, 40) → Take median.
(40,80)
(65, 60)



Balanced ~~Sorted~~ KD Tree.

→ Deleting a node in KD Tree.

Check the dimension (cut) of the node to be deleted.

If dimension = x
↓
then search for x minimum in right subtree of node to be deleted & replace the node to be deleted with x min wala node.

How to find x min → If x cut → then go into left subtree.

If y cut → then min(curr, left, right)

If dimension = y → then search for y min in right subtree of node to be deleted.
How to find y min → same as x min.

If node to be deleted is leaf node

↳ i.e. node ka left & right == NULL

then simply delete the node.

~~If internal node ka (which is to be deleted) ka right == NULL → directly delete the node~~

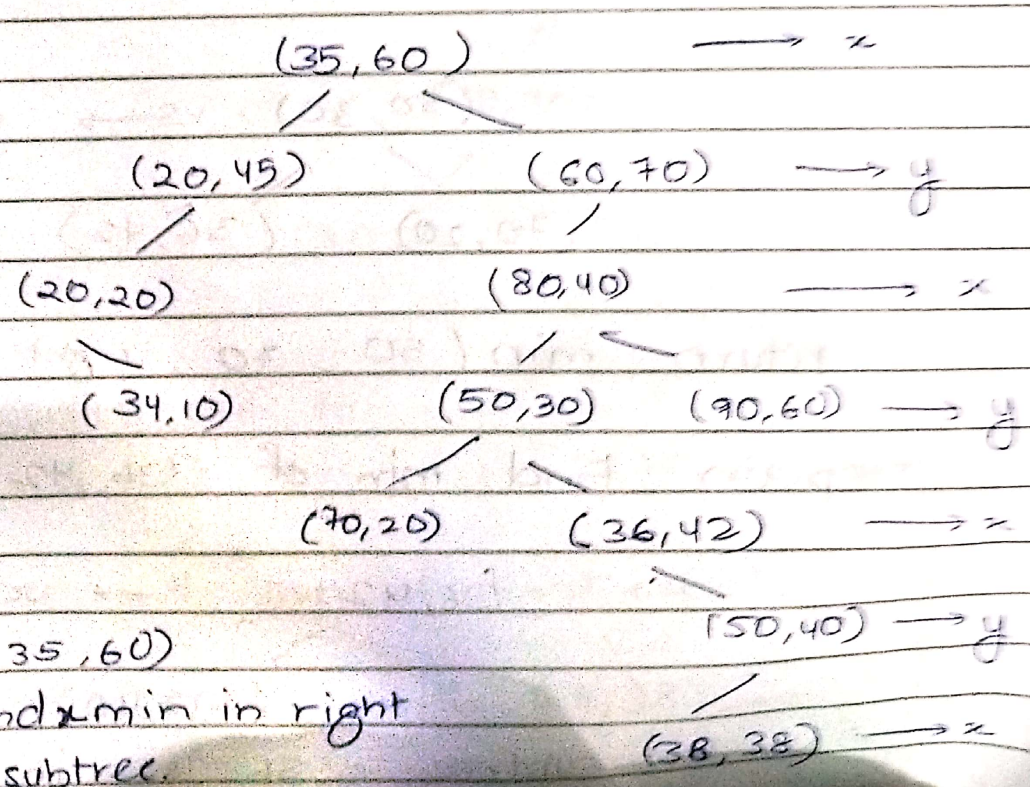
we always replace with the successor

<u>Method 1 (if no right subtree exists)</u>:-

If right subtree is NULL, then first check if left subtree exists & then replace node to be deleted with ~~minimum~~ maximum of left subtree.

~~e.g.~~ Recursive deletion call takes place for subsequent nodes.

e.g.



Delete (35,60)

↳ Find xmin in right subtree.

Step (i): Move to right subtree

(60, 70) → y - cut

(80, 40)

∴ min (80, left subtree, NULL)

Step (ii) Find min of (80, 40) ~~tree~~ subtree

(80, 40) → x cut.

(50, 30) (90, 60)

↓ return

∴ left subtree ka min.

Step (iii): Find min of (50, 30) subtree.

(50, 30) → y cut

(70, 20) (36, 42)

return min (50, 70, right subtree)

step (iv): Find min of (36, 42)

(36, 42) → x cut

(50, 40)

∵ No left subtree ∴ 36 is min.

∴ Trace back to step (iii)

min (50 , 70 , 36) = 36

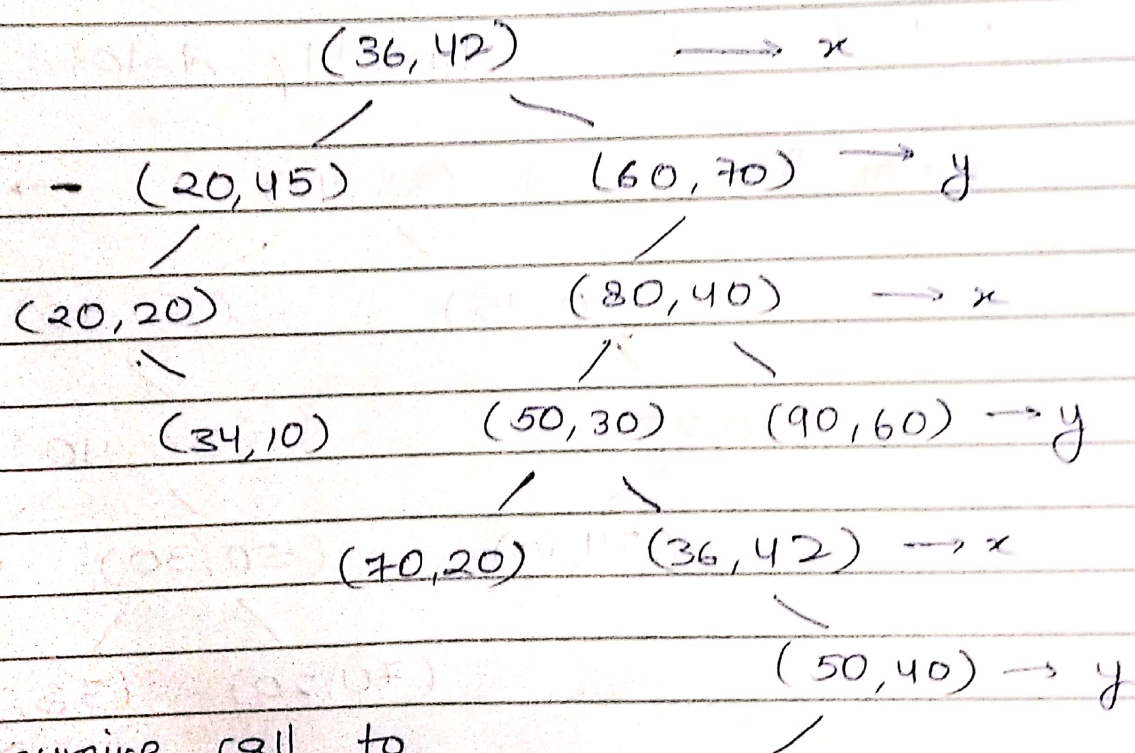
↓

Trace back to step (ii)

↓

min (60 , 36 , NULL) = 36

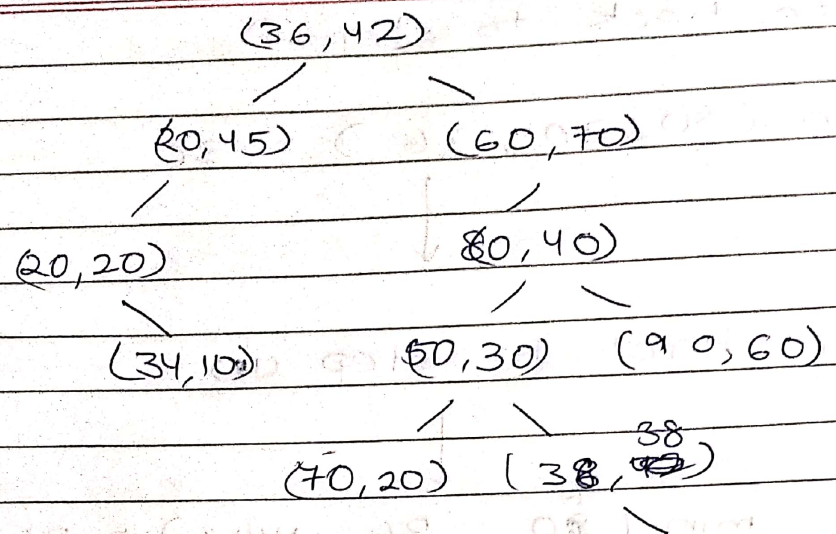
& ∵ 36 > 35 ∴ replace (35,60) with (36, 42)



Now recursive call to delete (36,42)

∴ Replace with min of right subtree.

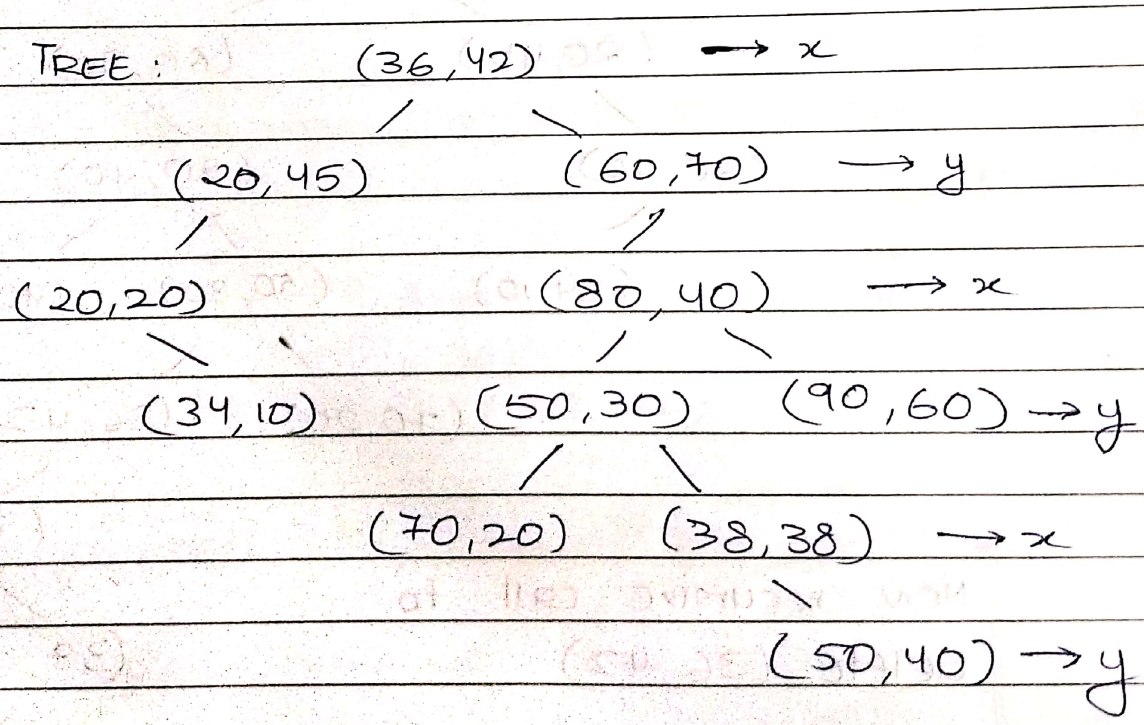
∴ Replace (36,42) with (38,38).



Recursive call to delete (38,38)

∵ leaf node ∴ directly delete

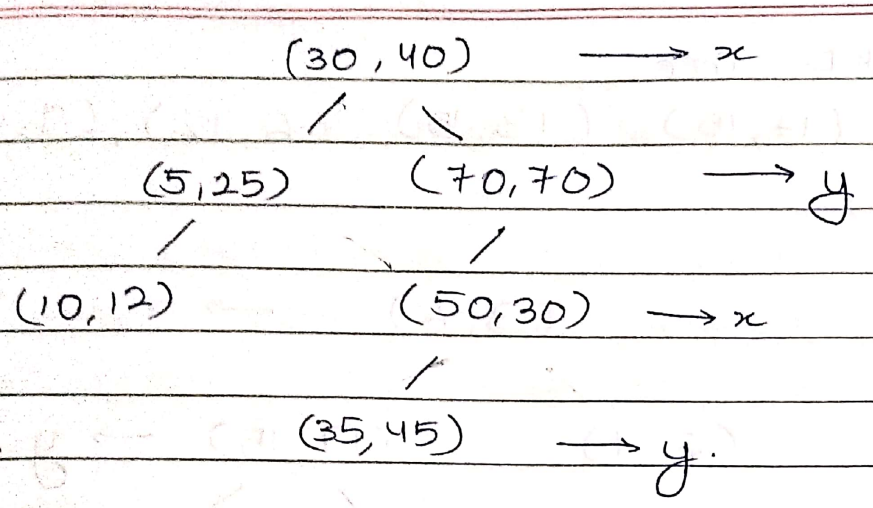
∴ FINAL TREE :



Method 2 (if no right subtree exists) :-

then we find minimum from left subtree & replace.

e.g.



Delete (70,70)

→ step (i) : Find y min of from left subtree.

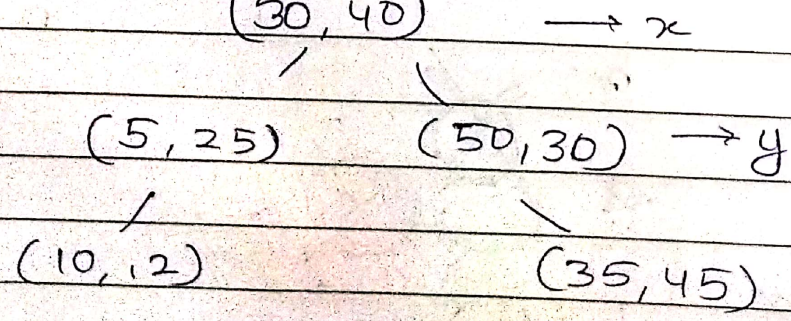
∴ replace with (50,30).

Now since, (50,30) goes in y dimension

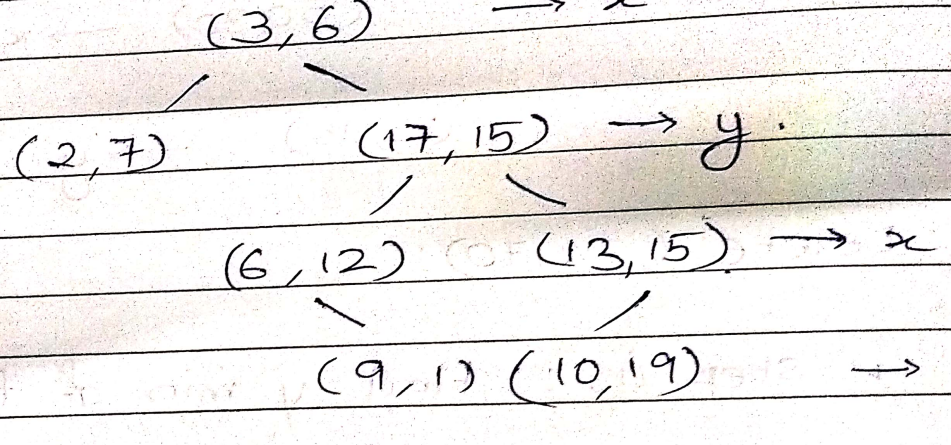
& child of (50,30) → i.e. (35,45) has y

60) co-ordinate greater than parent node.

∴ (35,45) becomes right subtree of (50,30) since 45 > 30



Q → Create KD Tree.
(3,6), (17,15), (13,15), (6,12), (9,1), (2,7), (10,19)



B+ TREES

B+ tree is a variation of B trees in which:

- internal nodes contain only search keys (no data)
- Leaf nodes contain pointers to data records.
- Data records are in sorted order by the search key.
- All leaves are at the same depth.

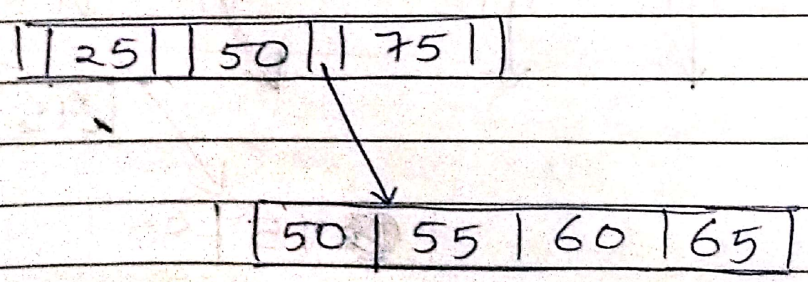
→ Internal Node:

(i) Pointer (Key, NodePointer) * M-1 in each node.
(ii).

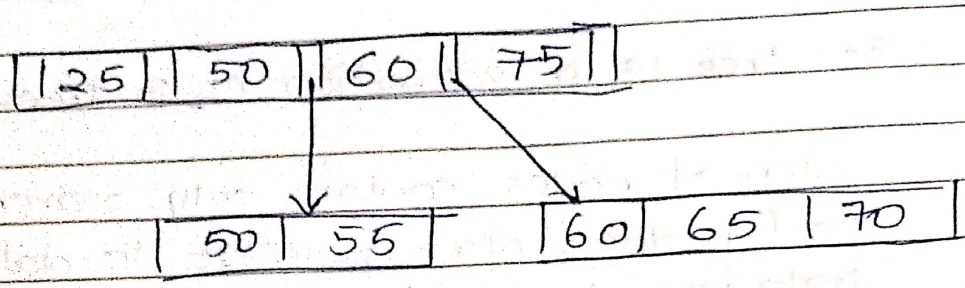
→ searching ~~insertion~~ in B+ trees is same as searching in B trees.

→ Insertion in B+ trees is same as insertion in B trees, but only difference is when leaf node overflows & we need to split the leaf node, then

↓

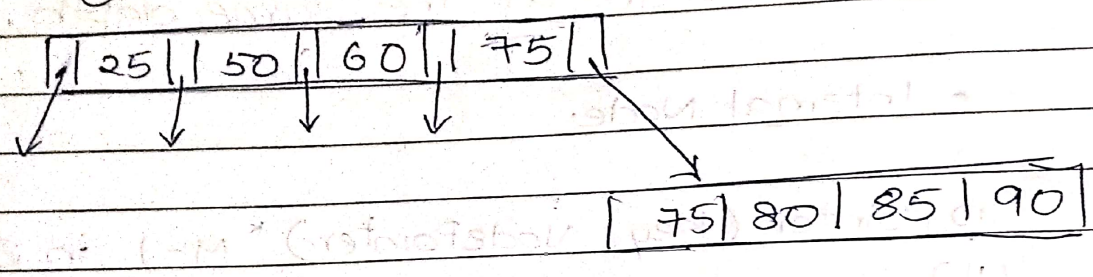


Now add 70 → then we split the leaf node at 60 → so we don't shift 60 in upper node, rather we create a copy of 60 & add to upper node.



~~→ Insert algorithm is~~

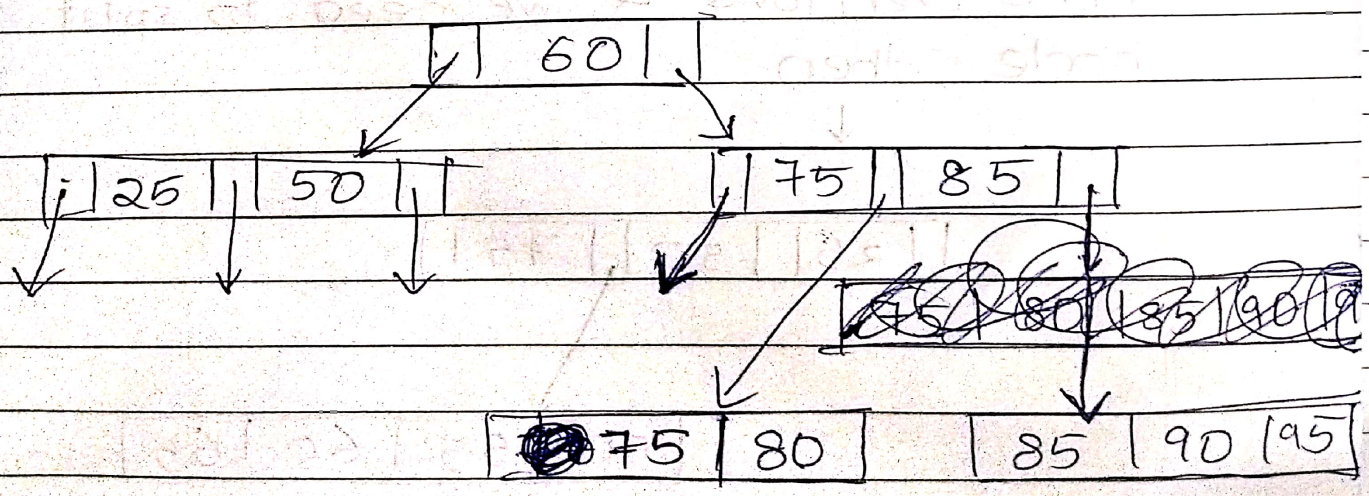
e.g.



add 95

↳ Then, since leaf node is split, **COPY** the median (i.e. 85) to the upper node.

↳ Then, internal node is filled, then split the internal node such that, median term (ie. is **MOVED** up.



Date
Page

(6)

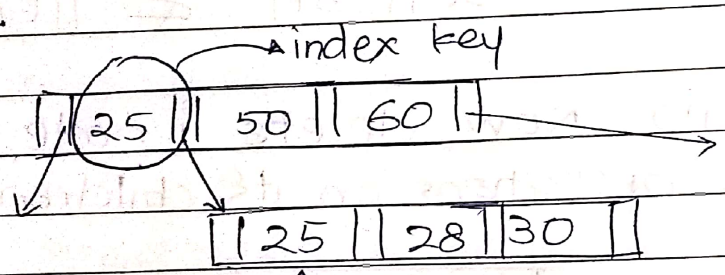
→ Deletion in B+ Trees: we can only delete data from leaf nodes, ∴ we cannot delete internal nodes. Internal nodes get deleted only when their data from leaf is deleted

Case (i) If the leaf node does not underflow, then no need to change anything.

~~(ii) If the leaf node underflows after key deletion, then apply merge operations.~~

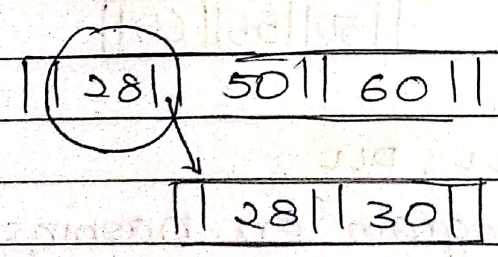
Case (ii) If the leaf node does not underflow, but the key to be deleted appears in the index page (i.e. appears in internal nodes) then, replace the index key with the key in the leaf node.

e.g.



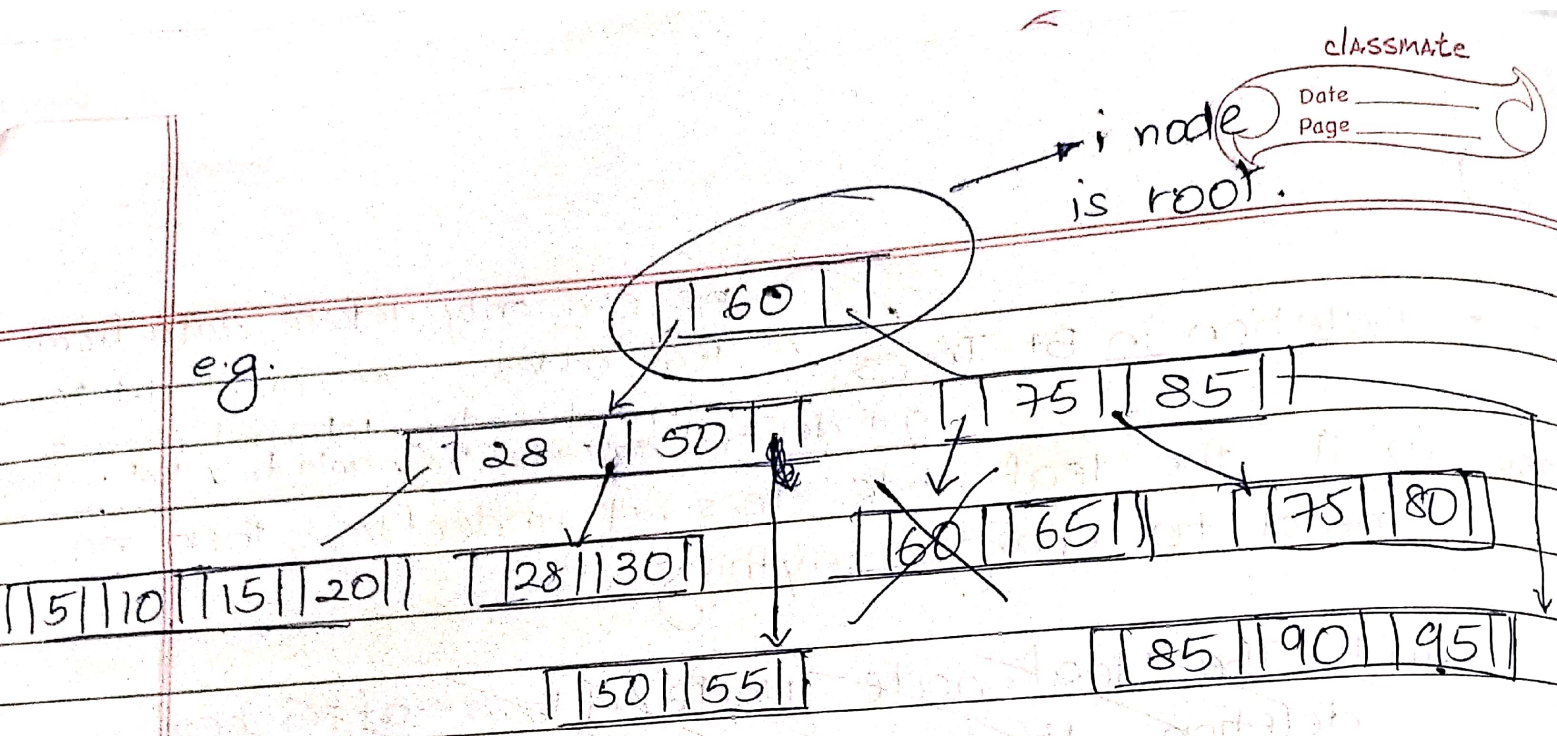
Delete 25

↳ 25 index appears, ∴ replace it with 28.



Case (iii) Data from leaf node to be deleted has index as root node, i.e. root node has to be deleted then perform merge operations.

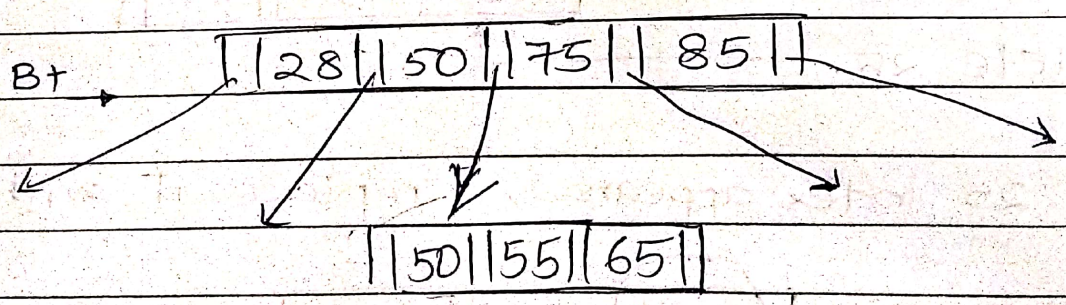
e.g.



Delete 60.

Step (i): Perform leaf node deletion. Leaf node underflows, ∴ merge.

Merge [50 55] & [65]

Step (ii): Now delete i node, & perform merge operations on its children.

Final B+ Tree