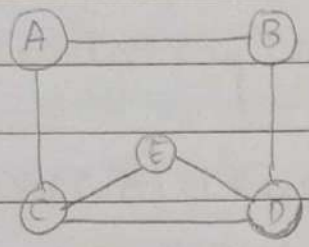
TREES ANG GRAPHS

Graph:- Set of vertices which are connected by edges forms a graph



$V = \{A, B, C, D, E\}$

$E = \{AB, AC \ldots\}$.

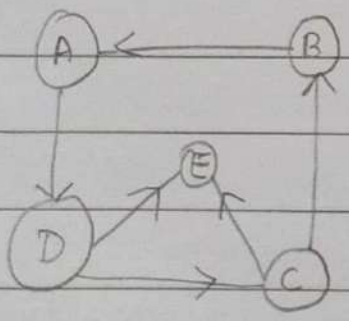
TYPES OF GRAPHS:

There are mainly two types:

i) Undirected graph: No direction is present on the edge

ii) directed graph: Direction is present on the edge

Ex

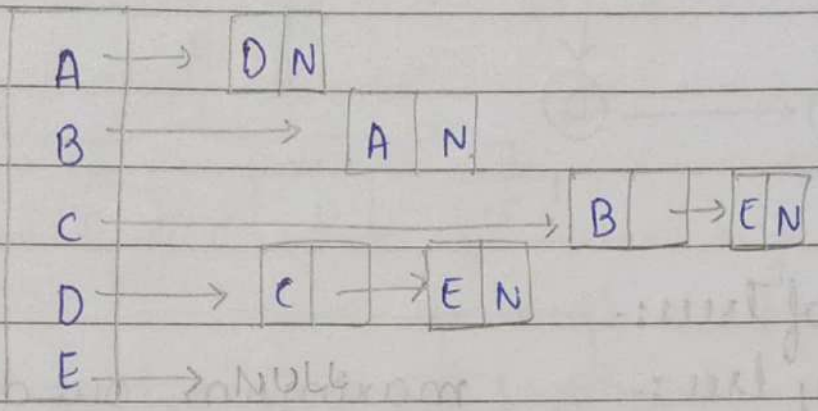


Two ways of representation:

* Adjacency Matrix

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

* Adjacency list representation:

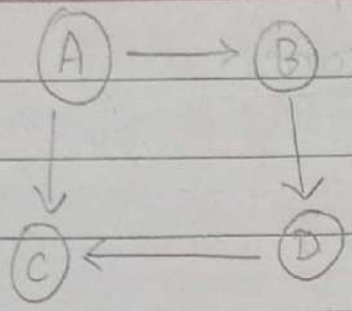


* Comparision of matrix representation with time and space complexity

Represnth	Space	Time to find edge b/w 2 vertices	Time to iterate over edges of a vertex
Matrix	$O(v^2)$	$O(1)$	$O(v)$
List	$O(v+E)$	$O(\text{outdegree}(v))$	$O(\text{outdegree}(v))$

* A tree is a directed acyclic graph. In which, there should be one node whose indegree is zero and all other nodes must have indegree greater than zero.
The node whose indegree is zero is called 'root'

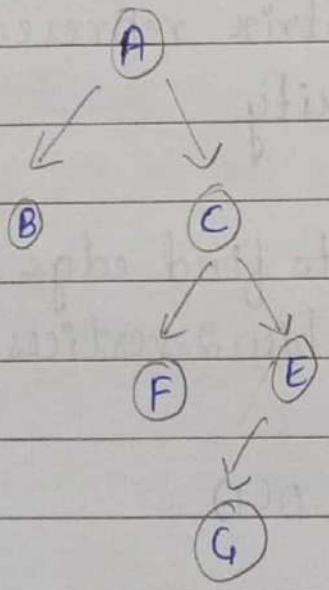
root

Ex. 

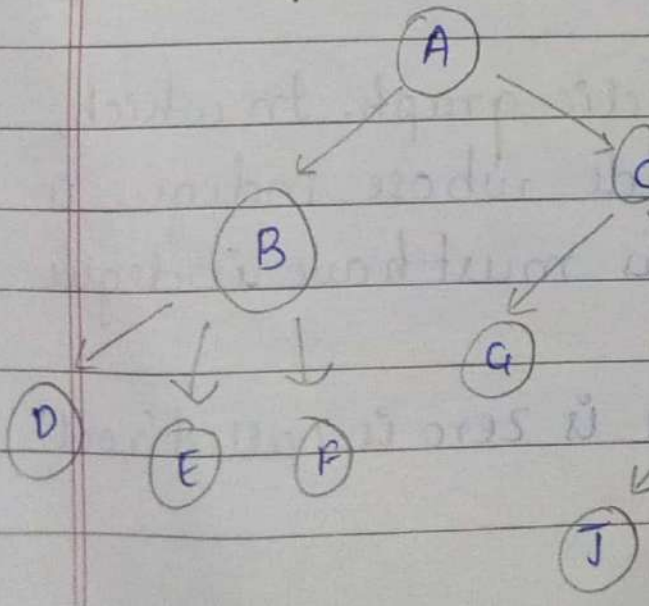
'A' is the root

Types of Trees:-

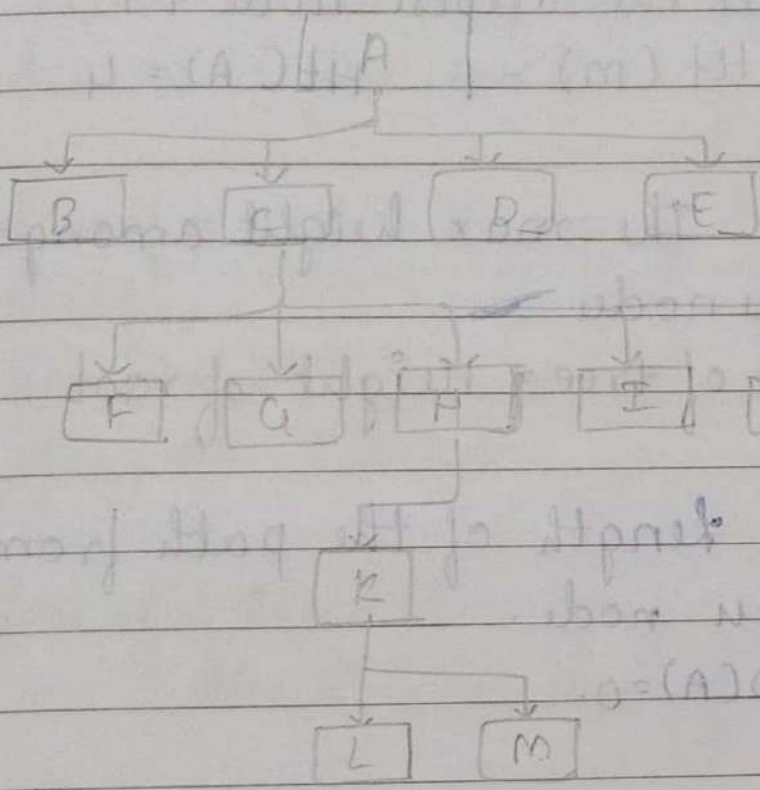
1) Binary tree:- maximum out degree of each node must be 2



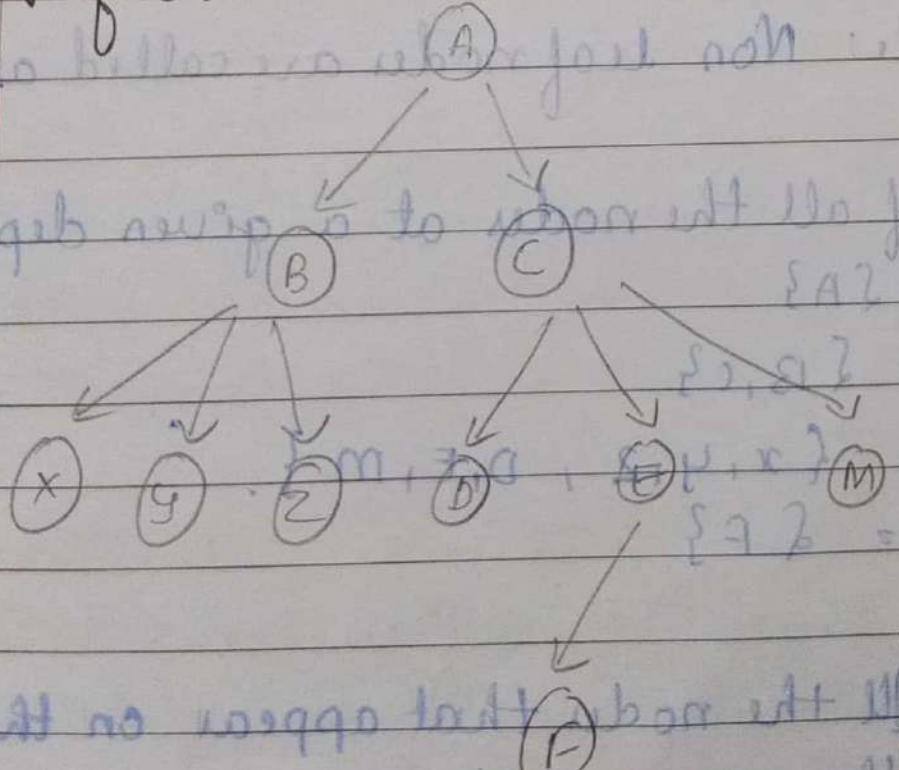
2) Ternary tree:- maximum outdegree of each node is three



Trie: Each node can have 'n' childsen. Used to represent dictionaries



Definitions:



Height of a node:- The length of the path from the given node to the deepest node + 1

H(I)(C) = 3 Ht(M) = 1 Ht(A) = 4

Height of a tree: The max height among the height of all the nodes

Height of tree = Height of root

Depth of a node: Length of the path from the root to a given node.

D(y) = 2 D(A) = 0.

Leaf node: The node whose outdegree is zero

x, y, z, D, F, M

Internal node: Non leaf nodes are called as internal node.

Level: Set of all the nodes at a given depth

Level (0) = {A}

level (1) = {B, C}

level (2) = {x, y, z, D, F, M}.

level (3) = {F}

Ancestors:- All the nodes that appear on the path from root the given node

Anc (D) = {C, A} Anc(A) = ϕ

Descendants:- The set of all the children and their childrens till leaf node
des (C) = { D, E, M, F }

Size of a node:- the no. of descendents including itself
des (C) + 1 = 4 + 1 = 5 = size (C)
size (A) = 9 + 1 = 10

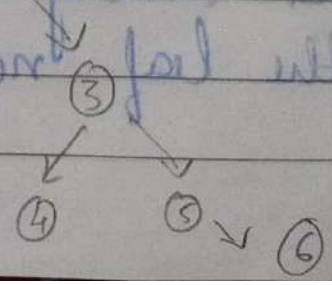
Size of a tree: The no. of nodes
Size of tree = Size of root

Least common Ancestor (LCA) of two nodes.
LCA (X, C) = A // Every node is ancestor of itself
LCA (D, F) = C.
LCA (B, A) = A

Parent: The immediate ancestor is called parent
π(y) = B.

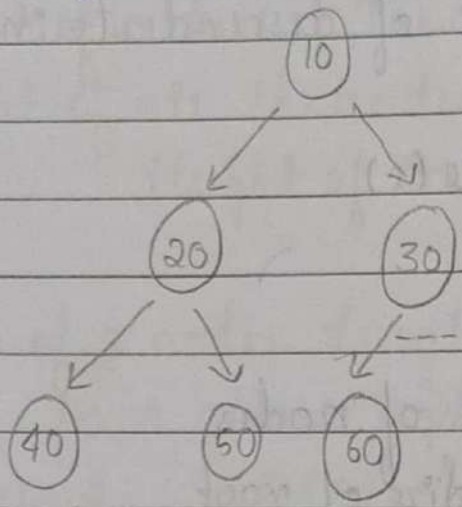
TYPES OF BINARY TREES:.

i) Binary tree:- Each node can have max of two children



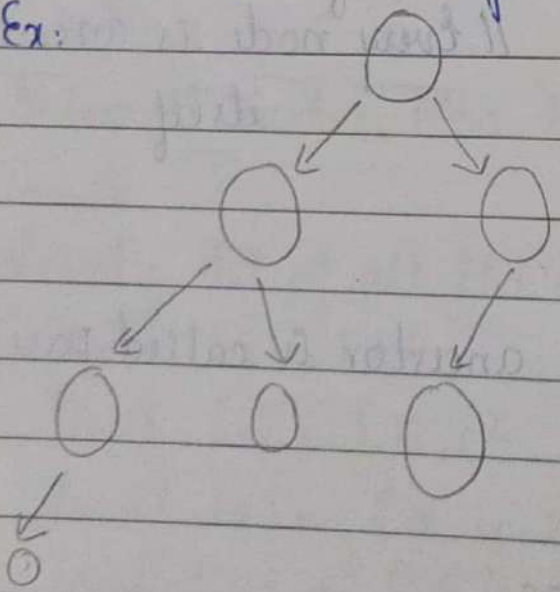
Complete Binary: All the levels should be completely filled except possibly the last level, if the last is partially filled then all the nodes should be as left as possible.

Ex:

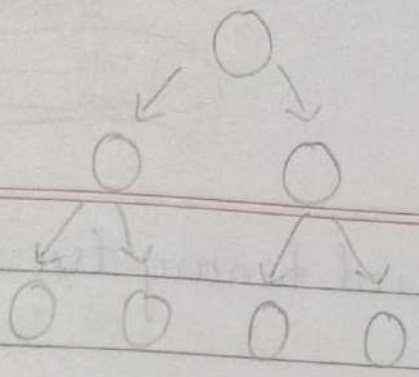


↳ Not complete binary binary tree

Ex:



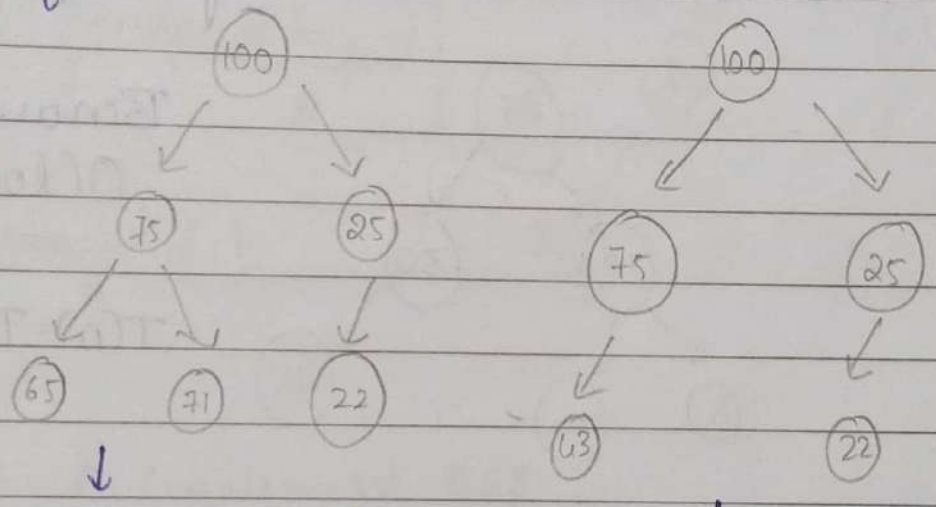
Full binary tree: It is called strict binary tree all nodes must exactly have 0 or 2 children and all the leaf nodes should be at same level



Heap:- It is a compldebinary tree. There are two types:

→ Max heap: Parent data greatu than child data for all nodes.

tx:-



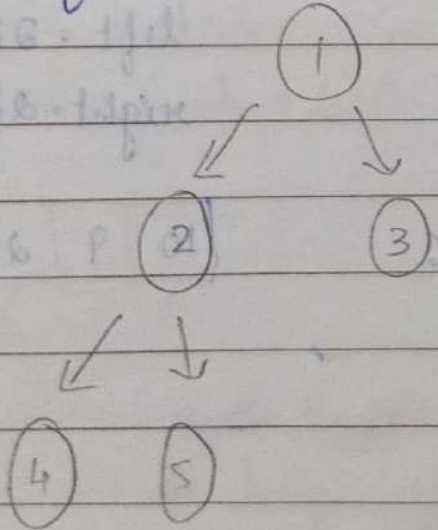
↓

It is max heap

↓

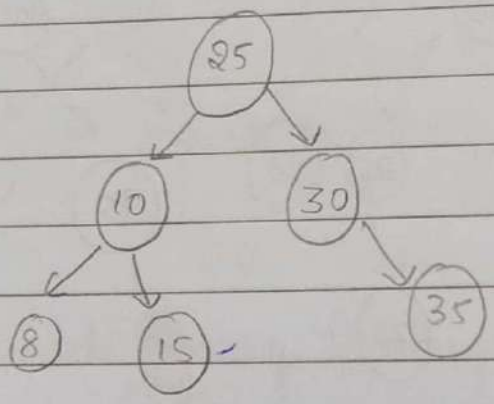
Not map heap
//not complete

→ Min heap: Parent data lesser than children data for all nods



Binary Search tree: Is node based binary tree which has following properties:

1. Rightsubtree has the data greater than its parent
2. Leftsubtree has the data less than its parent
3. Rightsubtree and left subtree should be binary search tree
4. There should not be any duplicate nodes

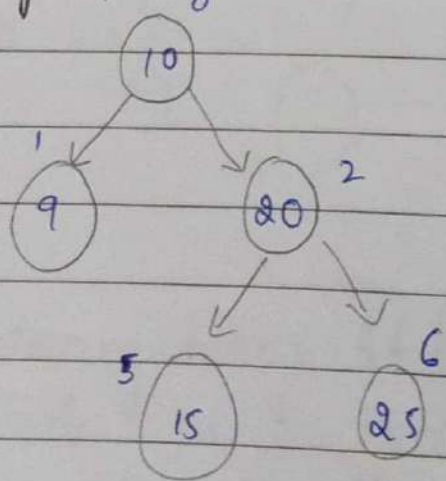


Binary search tree has $O(\log_2 n)$ to search

$$T(n) = T(n/2) + 1, \quad n > 1$$
$$1, \quad n = 1$$

Binary tree has time complexity $O(n)$ to search

* Array Representation of binary tree



root = 0 index.
left = 2i + 1
right = 2i + 2
} 'i' index of parent

10	9	20	N	N	15	25
0	1	2	3	4	5	6

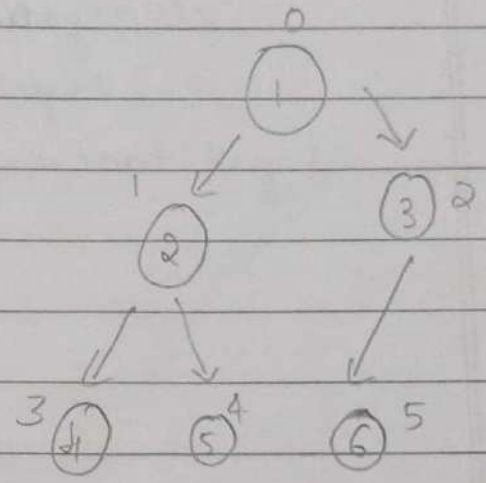
NOTE: It is mostly used for complete binary tree because all the NULL's are at the end.

We represent heap using arrays.

$\pi(25) = \frac{6-1}{2} = 2$

$\pi(key) = a\left[\frac{\text{index of key} - 1}{2}\right]$

$\pi(4) = \frac{3-1}{2} = a[1]$

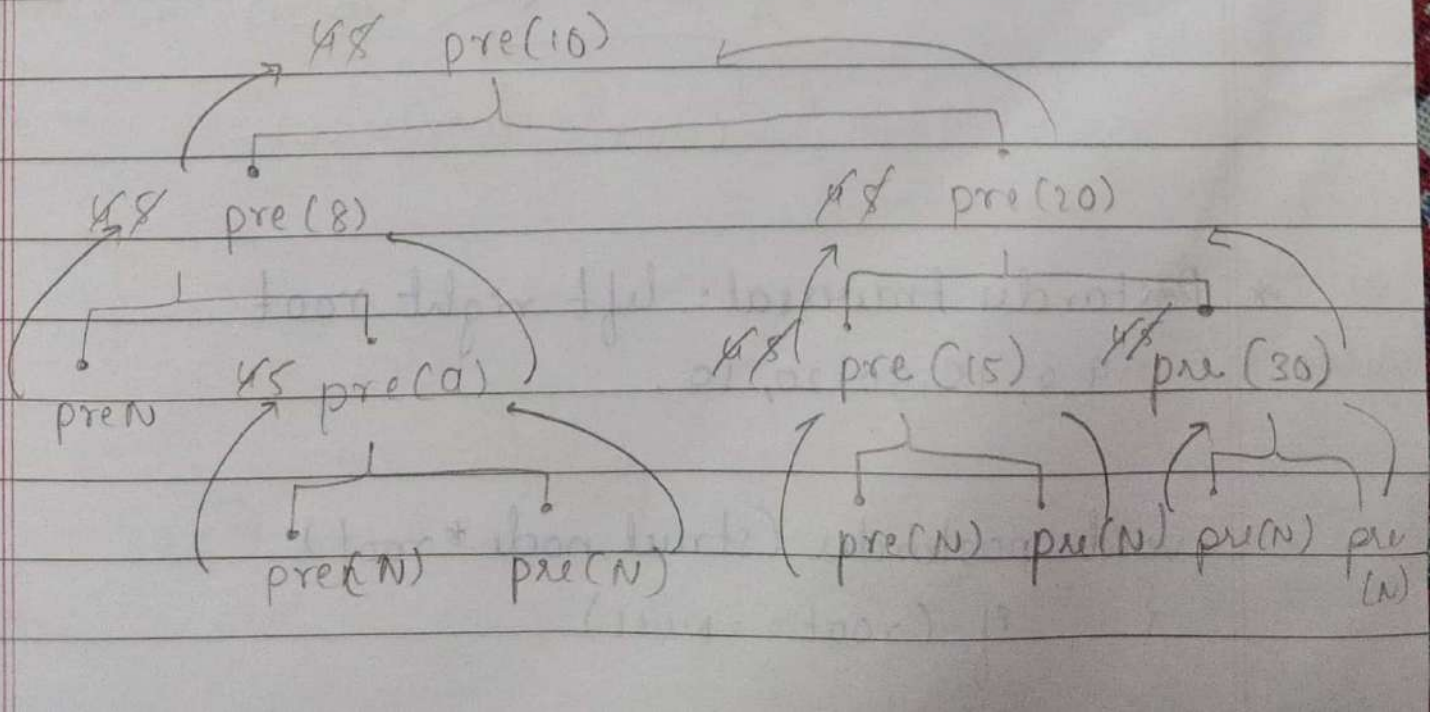


Tree traversal :-

preorder : root, left, right

10, 8, 9, 20, 15, 30

```
void preorder (struct node *root)
{   if (root == NULL)
        return;
    printf ("%d \n", root->data);
    preorder (root->left);
    preorder (root->right);
}
```

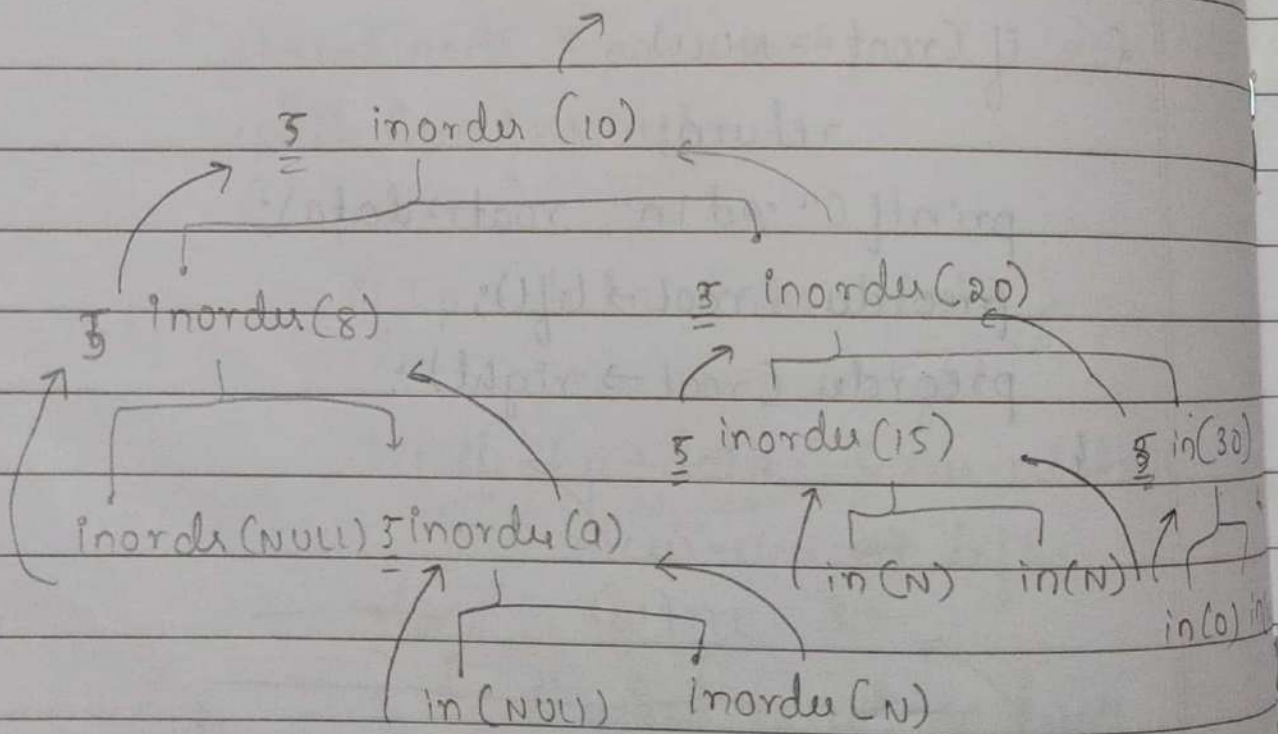


* Inorder traversal :- left, root, right

8, 9, 10, 15, 20, 30

```
void inorder (struct node * root)
{ 1 — if (root == NULL)
  2 —— return;
  3 —— inorder (root → left);
  4 —— printf ("%d \n", root → data);
  5 —— inorder (root → right);
}
```

8 9 10 15 20

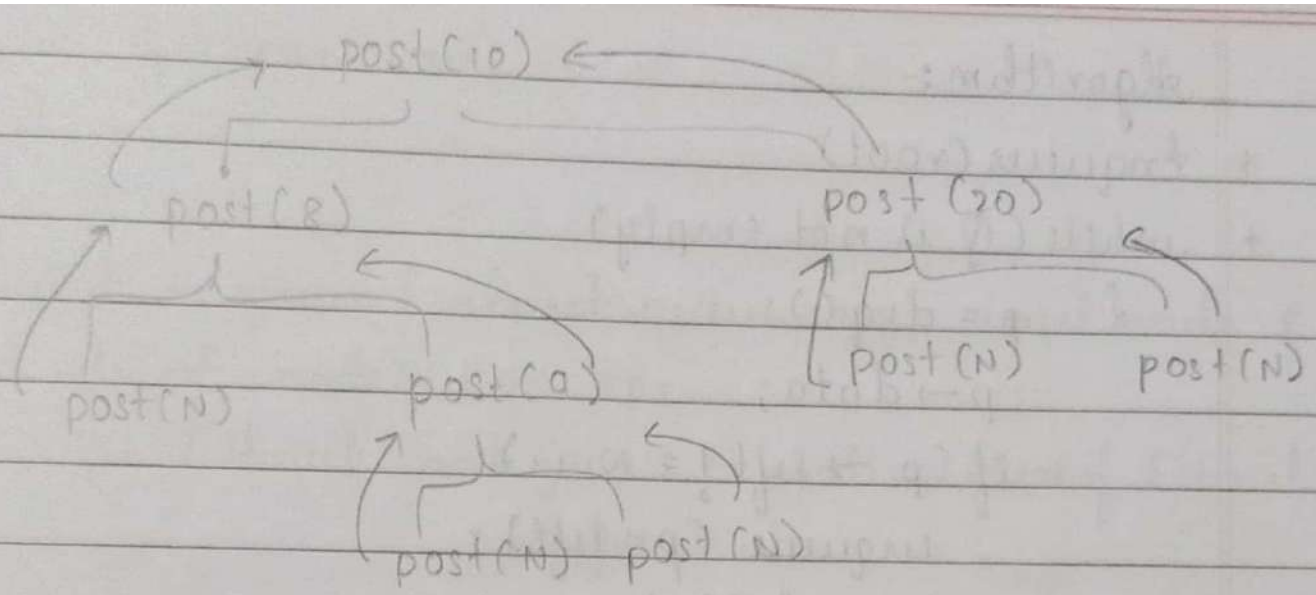


find
height
delete

* Postorder traversal: left right root

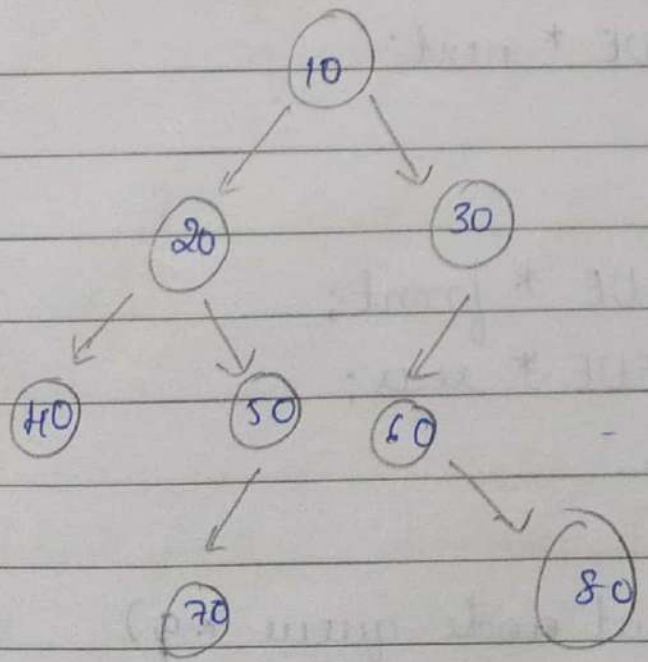
9, 8, 15, 30, 20, 10.

```
void postorder (struct node * root)
{ 1 — if (root == NULL)
```




```
2 —— return;
3 —— postorder (root → left);
4 —— postorder (root → right);
5 —— printf ("%d", root → data);
6 —}
```

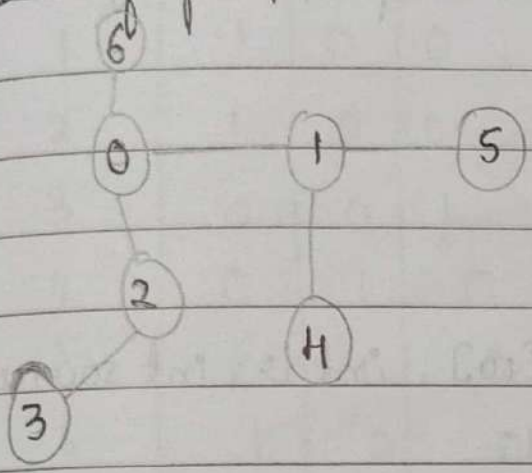
Level Order traversal:



10, 20, 30, 40, 50, 60, 70, 80

Breadth first search

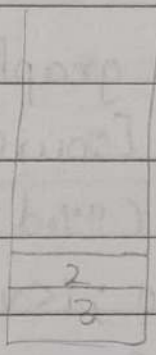
* DFS of graph:



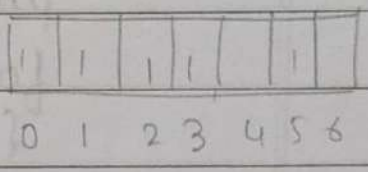
source = 0

o/p: 0, 1, 4, 5, 2, 3, 6.

source = 3



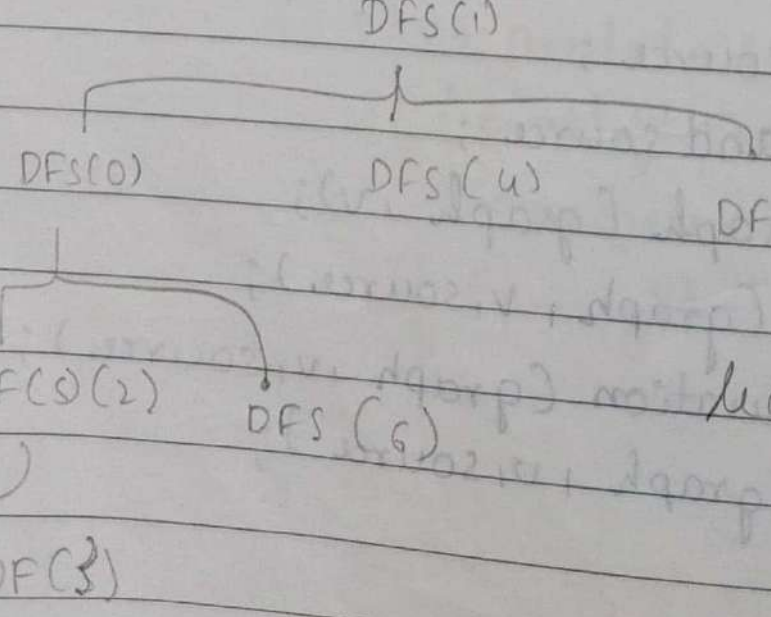
3, 2, 0, 6, 1, 4, 5.




```
// global declaration
int visited[10] = {0};
main()
{  int graph[10][10];
   int v, source;
   read v and source;
   readgraph (graph, v);
   DFS (graph, v, source);
   DFSituation (graph, v, source);
   BFS (graph, v, source);
}
```

```
void readgraph ( int graph[10][10], int v)
{   for( int i=0; i<v; i++)
    {  for(int j=0; j<v; j++)
       {  scanf ("%d", &graph [i][j]);
       }
    }
}
```

```
void DFS ( int graph [0][10], int v, int source)
{     visited [source ]=1;
      printf ("%d \t ", source);
      for(i=0 ; i<v ; i++)
      {  if (graph [source][i] ==1 &&
                visited [i] ==0)
1 —        DFS (graph, v, i);
2 —  }
3 — }.
```



leaf nodes are dead ends

visited

0	1	2	3	4	5	6

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	1
1	1	0	0	0	1	1	0
2	1	0	0	1	0	0	0
3	0	0	1	0	0	0	0
4	0	1	0	0	0	0	0
5	0	1	0	0	0	0	0
6	1	0	0	0	0	0	0

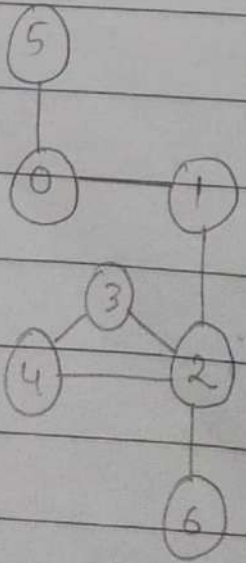
DFS (3)

BFS

```
main()
{ int graph [10][10];
  int v;
  read v;
  readgraph (graph, v);
  int source;
  read source;
  BFS (graph, v, source);
}
```

```
void BFS (int graph [10][10], int v, int source)
{ struct queue q;
  initqueue (&q);
  enqueue (&q, source);
  visited [source ] = 1;
  while ( ! isempty (&q) )
  {   u = dequeue (&q) ;
      printf ("%od", u);
      for(int i=0 ; i< v; i++)
      { if (graph [u] [i] == 1 &&
              visited [i] == 0)
        { visited [i] = 1;
          enqueue (&q , i);
        }
  }
}
}.
```

visted



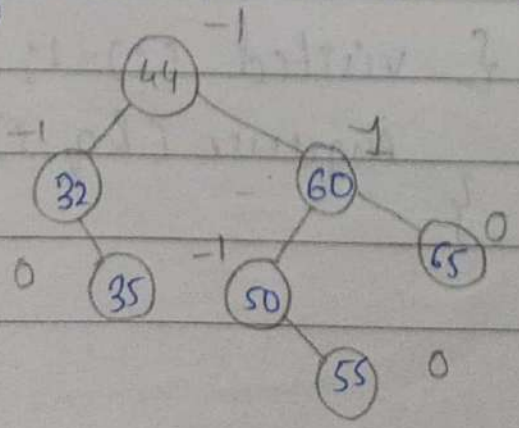
Queue

AVL Tree

An AVL tree is a BST in which the height difference of leftsubtree and rightsub-tree should be atmost 1

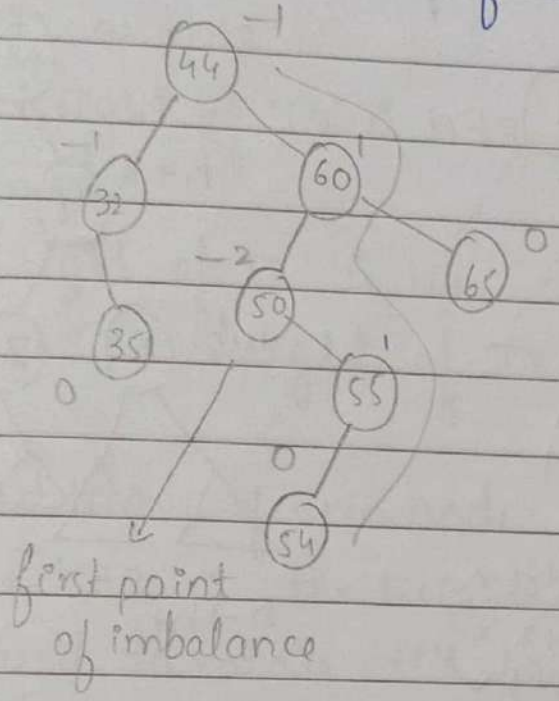
Balance factor = ht(L.S.T) − ht(R.S.T)

Ex:



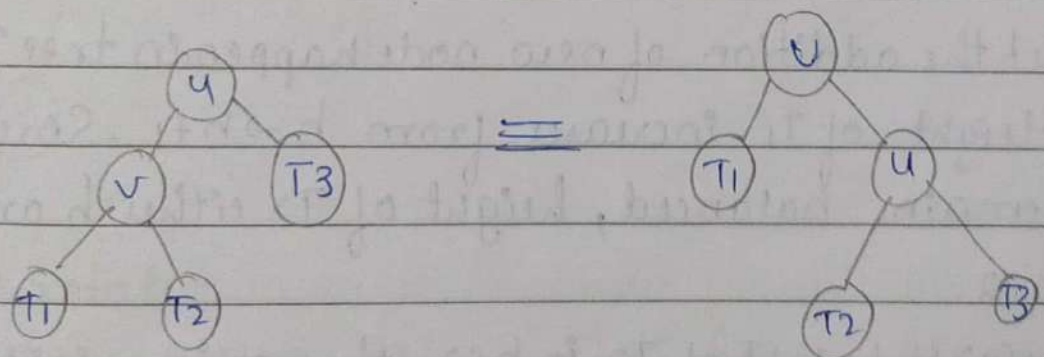
= {0, 1, -1}

If we add 54 the balance factor changes



The logical way of reorganising the tree - rotation

A tree can be reorganized so that it continues to remain BST

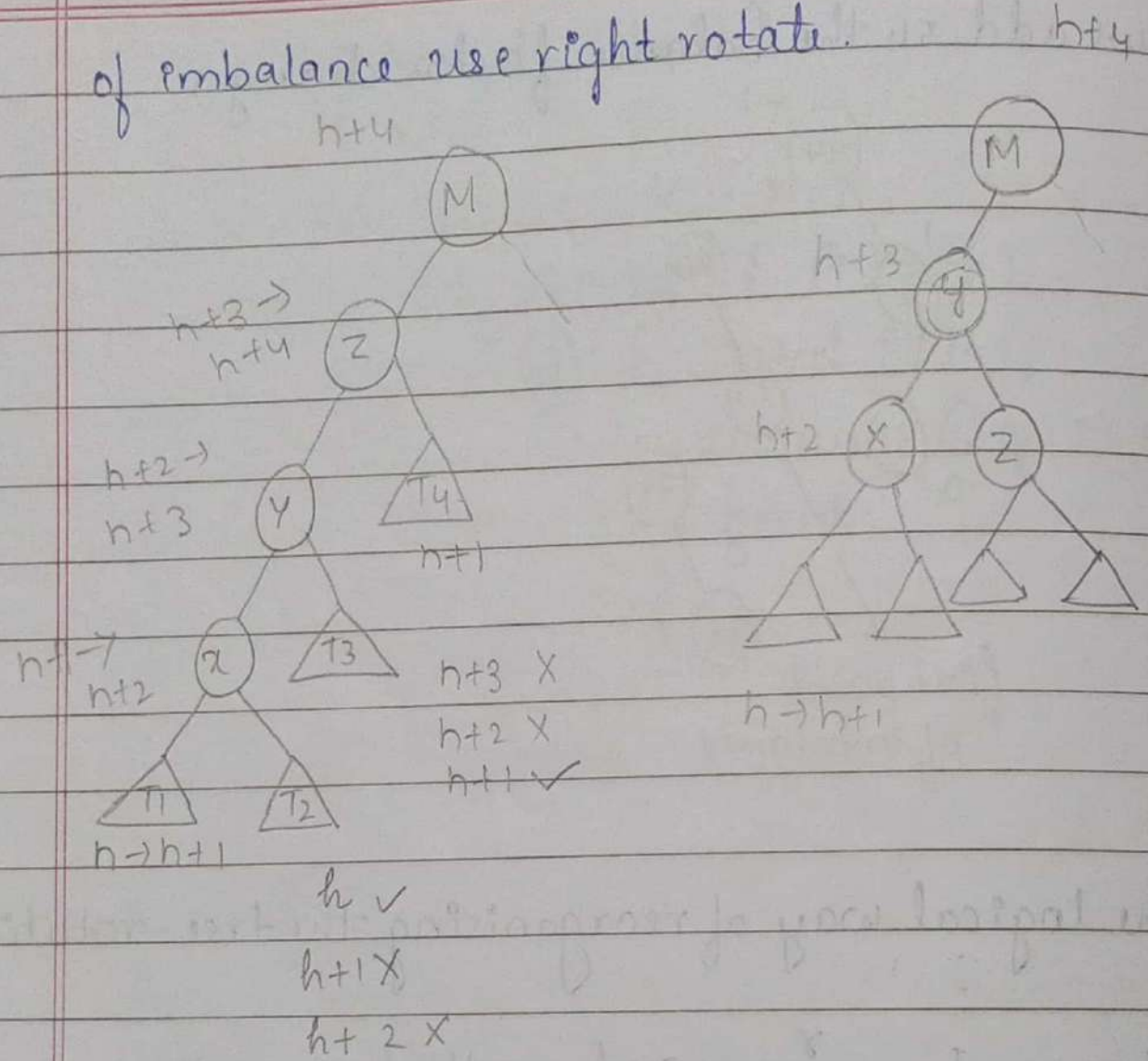


$T_1 < v < T_2 < u < T_3$

Types of Rotation

i) Right rotation :- If the insertion of new node happens towards complete left w.r.t. first point

of imbalance use right rotate.



Let the addition of new node happen in tree T_1
Height of T_1 increases from $h \to h+1$. Since x remains balanced, height of T_2 either h or h+1 or h+2.

→ (a) If height of T_2 is h+2 then x is originally imbalanced

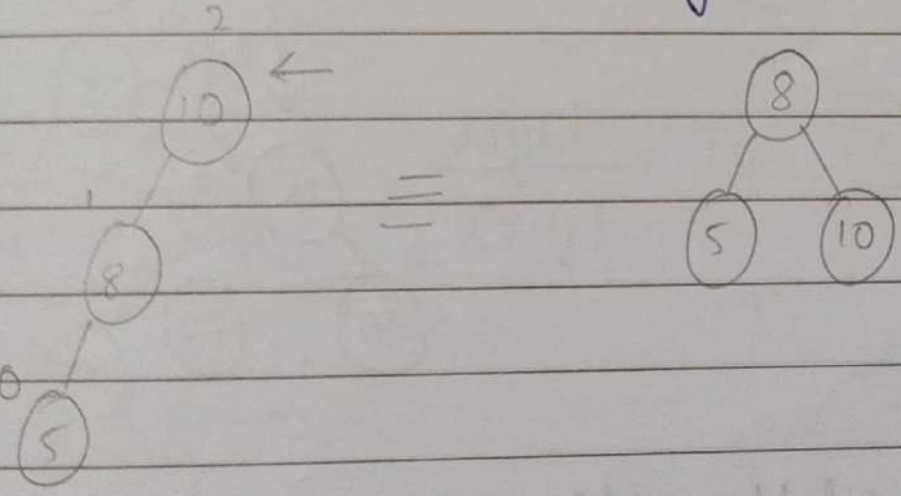
(b) If height of T_2 is h+1 then height of x wont change ∴
height of T_2 is h

Since y remains balanced height of T3 should be h+1 or h+2 or h+3

→(a) If height of T3 is h+3 then originally y is imbalanced

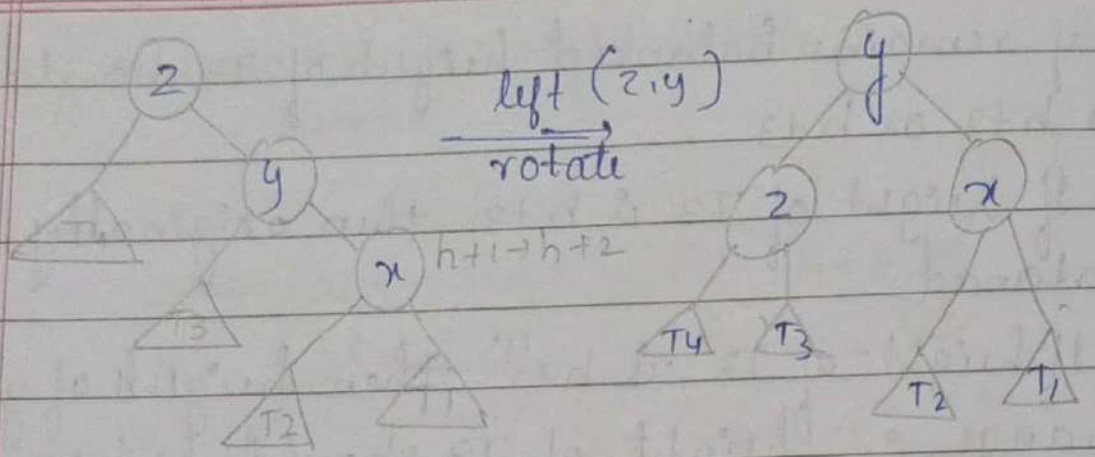
(b) If height of T3 is h+2 then height of y won't change ∴ height of T3 should be h+1

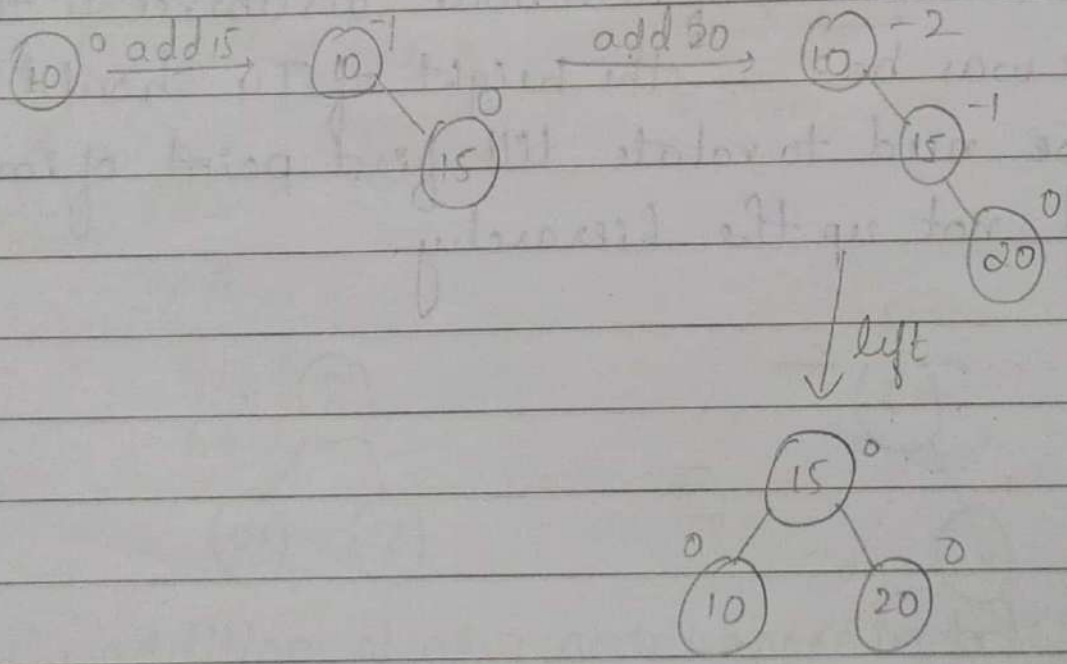
Before addition of new node in subtree T1 the height of z was h+3 ∴ the height of T4 should be h+1 ∴ we need to rotate till first point of imbalance and not up the hierarchy.



Left rotate :-

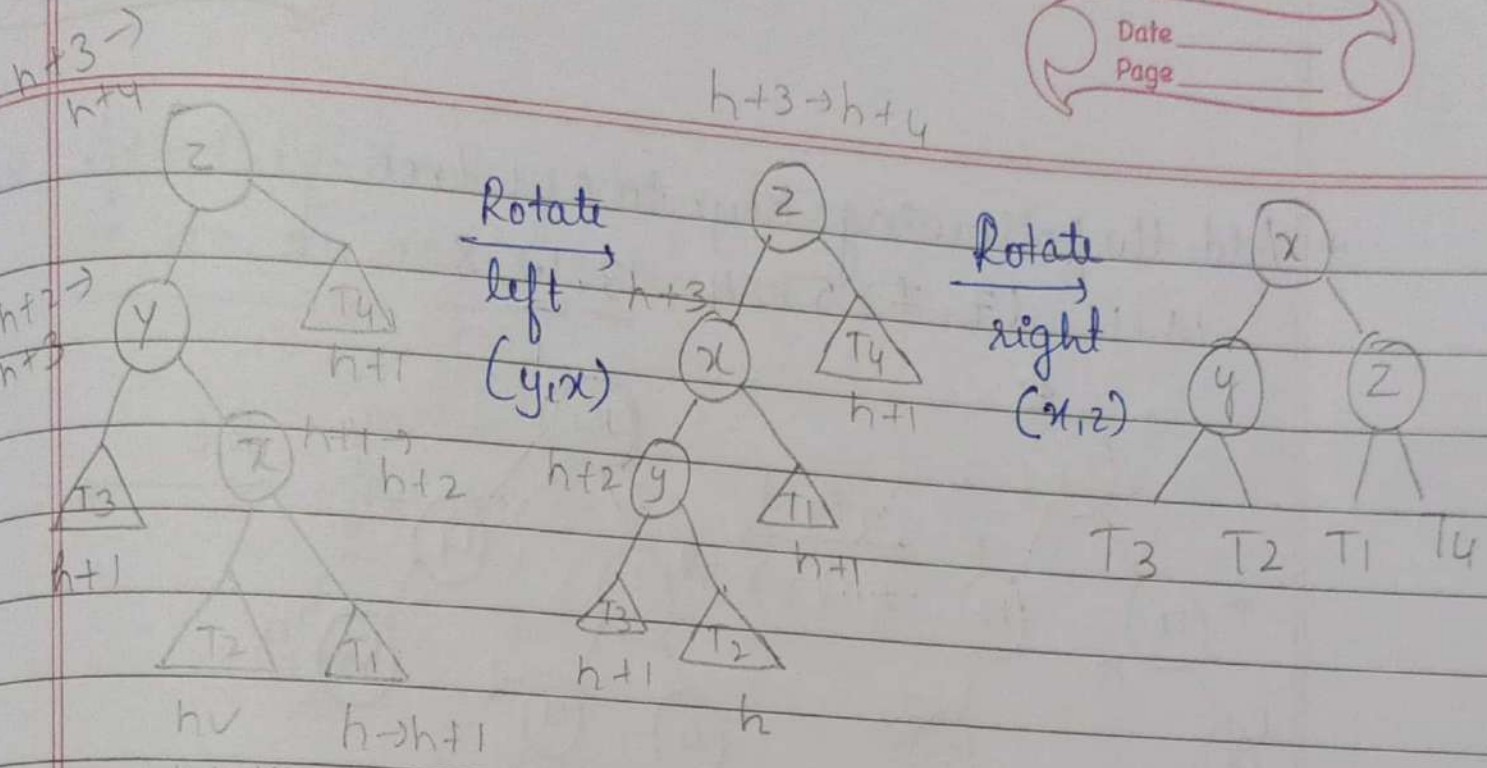
If the addition of new node happens towards complete right from the first point of imbalance use left rotate



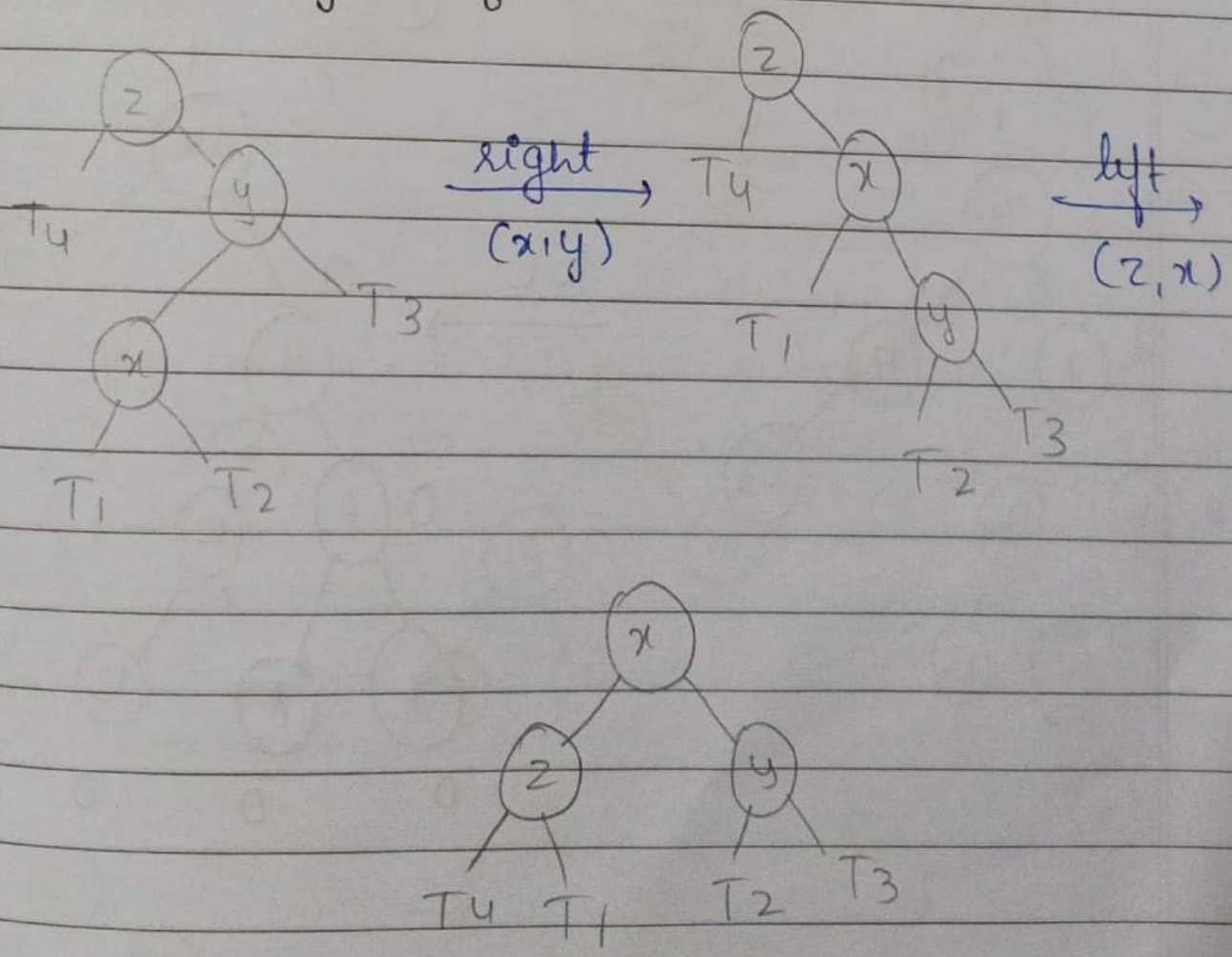


LR-rotate (left-right)

If addition of new node happens towards left and then right w.r.t first point of imbalance use LR rotate

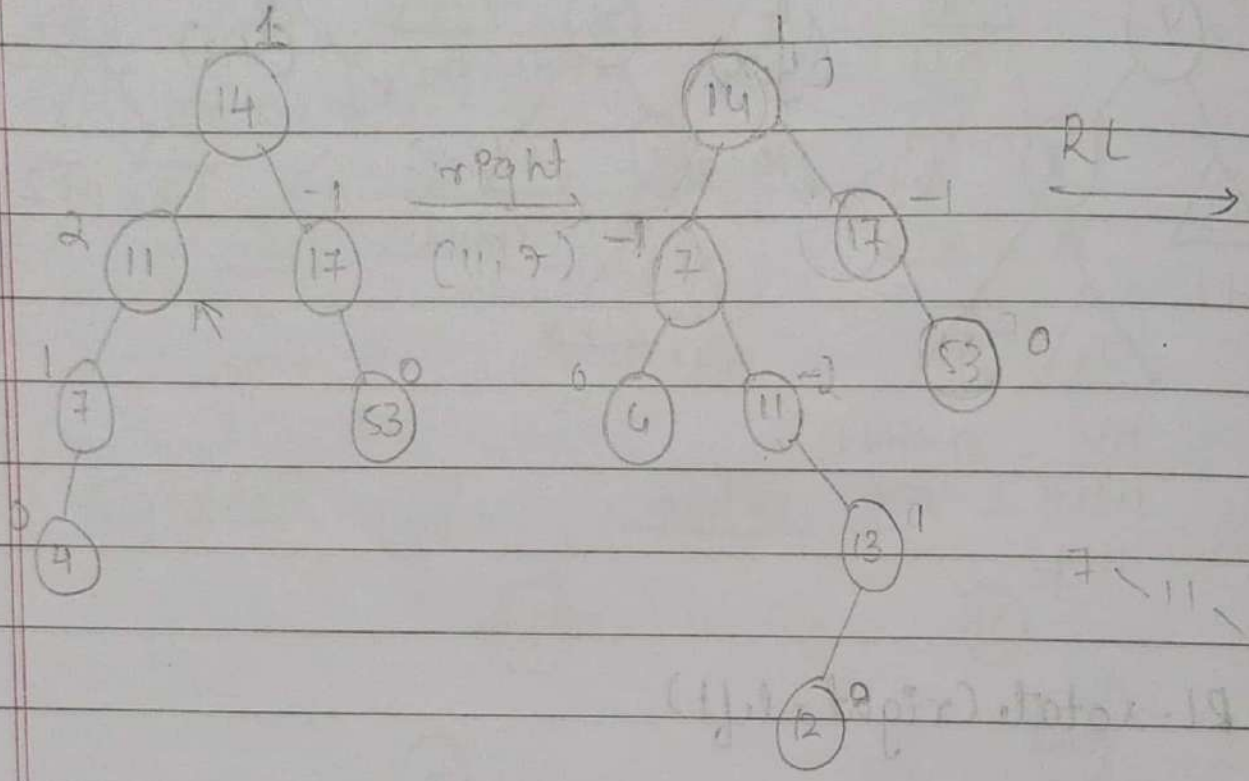


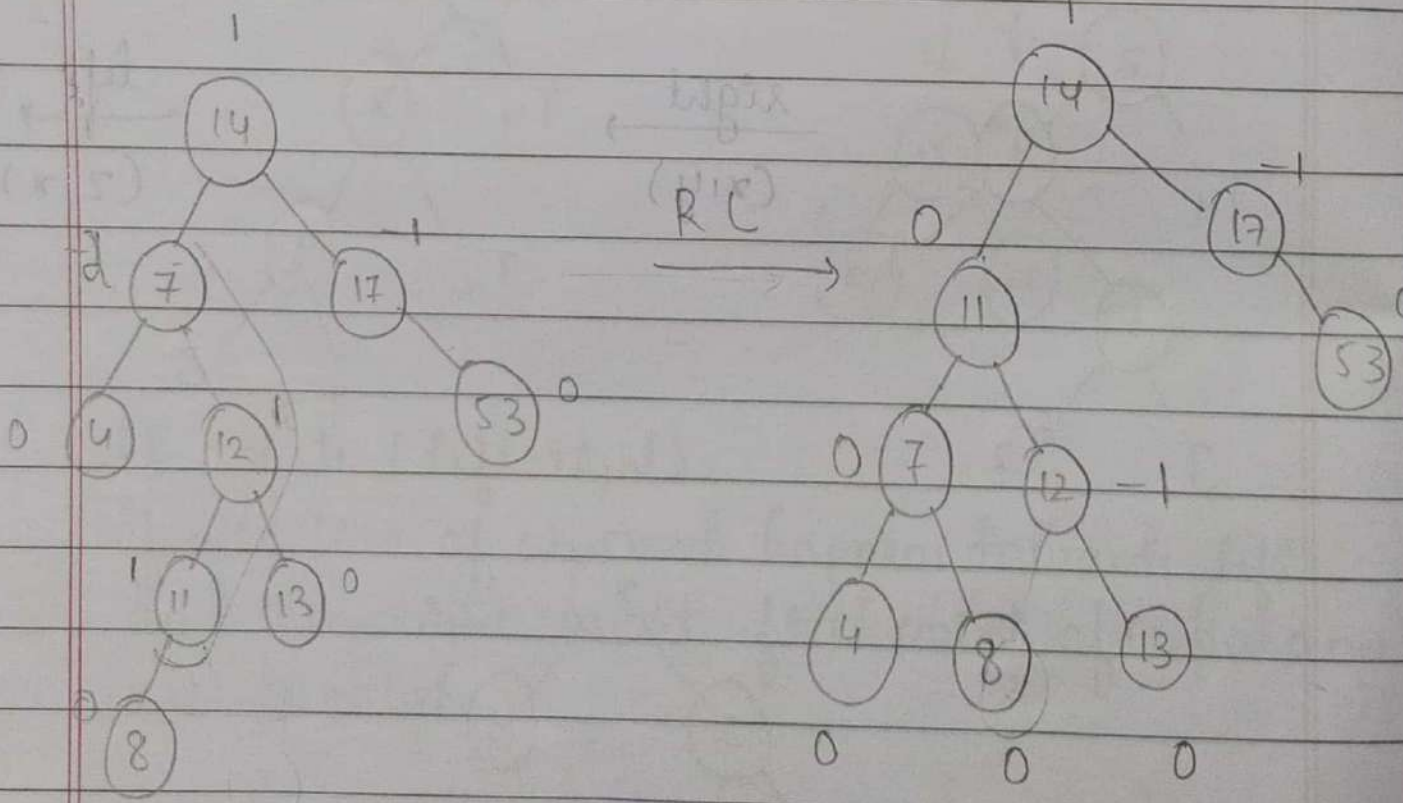
* RL-rotate (right-left)



* Add the following keys to AVL tree

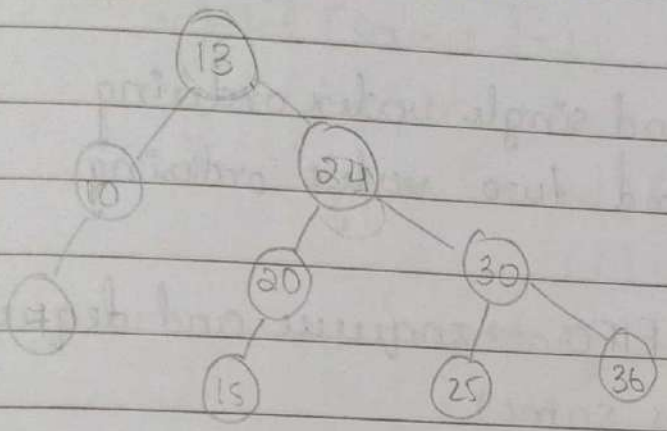
14, 11, 17, 7, 53, 4, 13, 12, 8

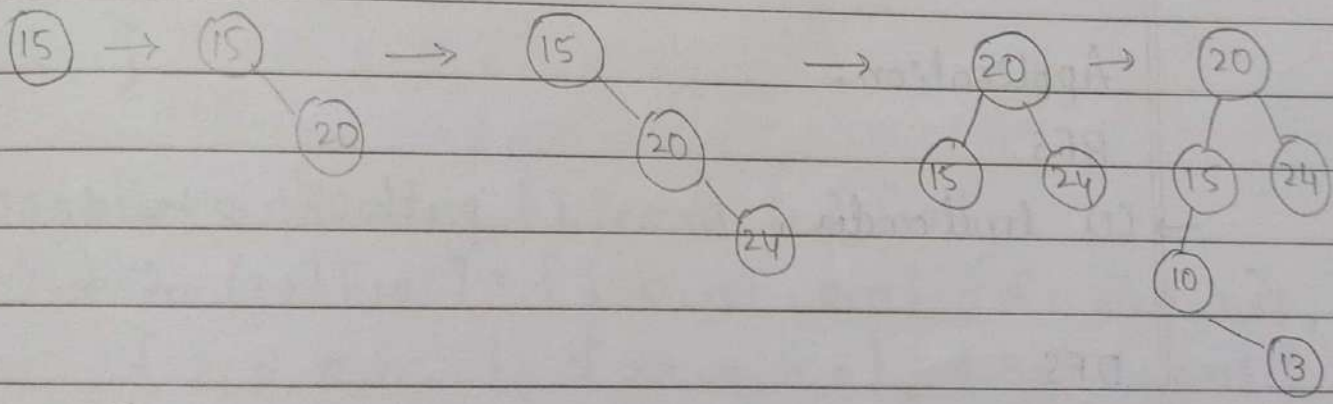


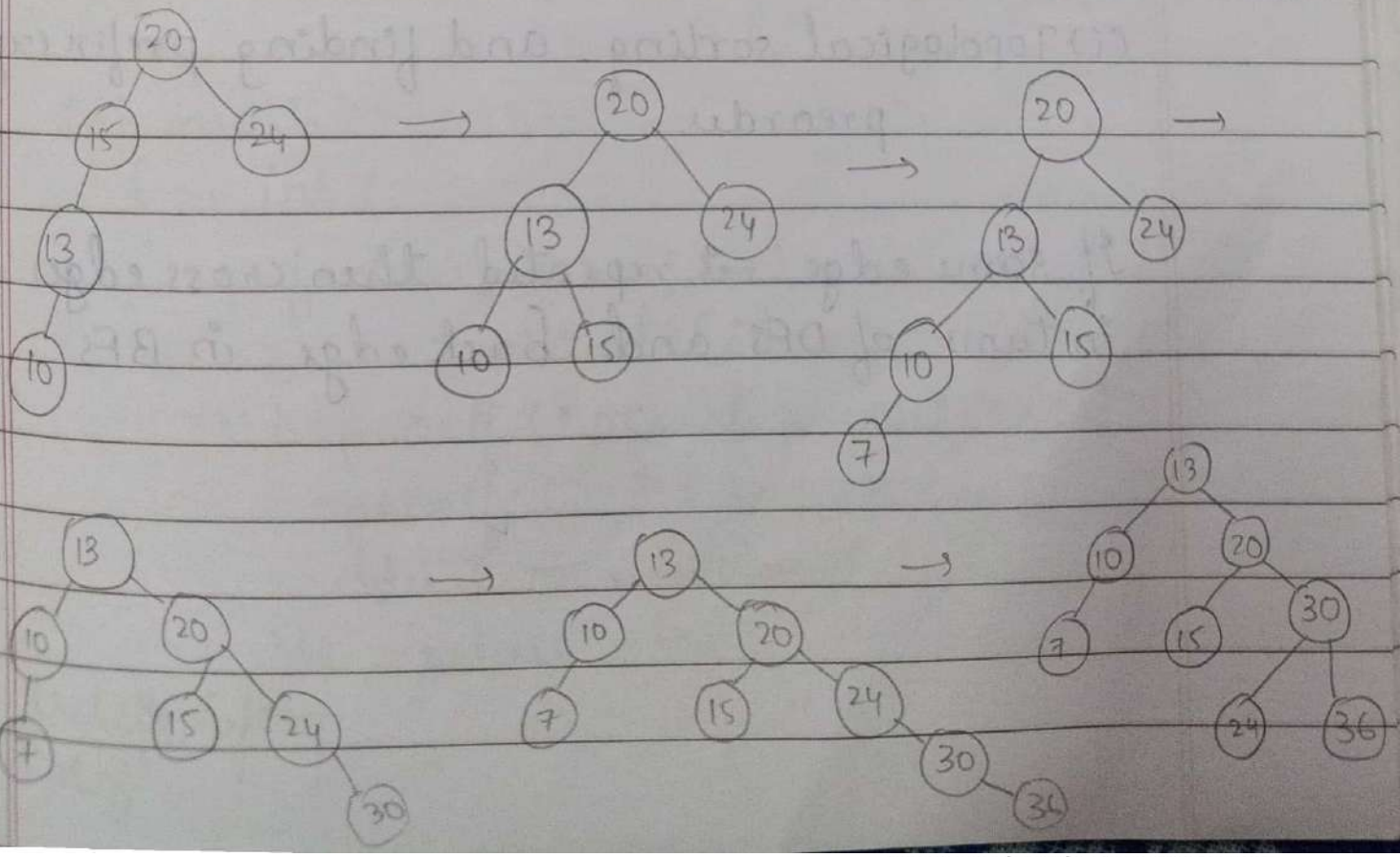


Date
Page

* Add the following keys to AVL tree
15, 20, 24, 10, 13, 7, 30, 36, 25



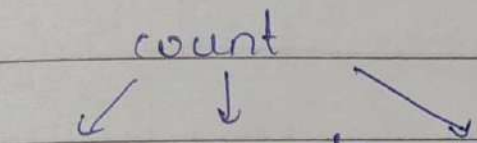




TRIE

It is used to store collection of strings. It is used for prefix based search

/* Prefix is only stored. So space is saved.



Only count is used or stored so space is reduced */

It is used to implement dictionaries

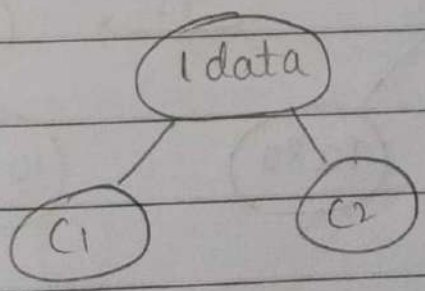
```
#include <stdbool.h>
struct trienode
{ struct trienode *h[26];
   bool is_end;
   struct node * head; //meaning
}

struct node
{ char str[25];
   struct node * next;
};
```

2-3 trees: /* It is a BST feature having tree. */

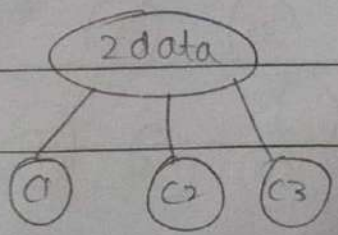
2-3 nodes have three nodes:

(i) leaf node

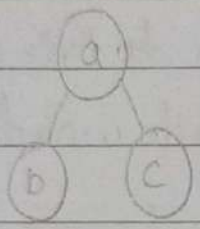
(ii) 2-nodes: It has one data item and two childern



(iii) 3-nodes: It has two data items and three childern

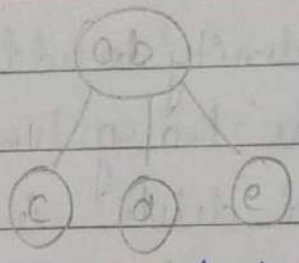


2node



b<a<c.

2,3node

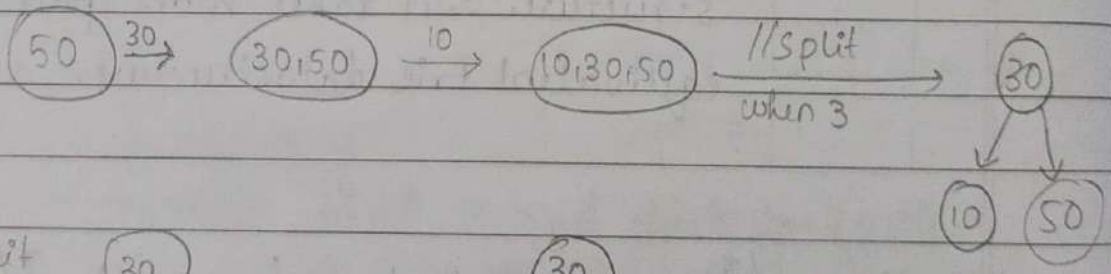


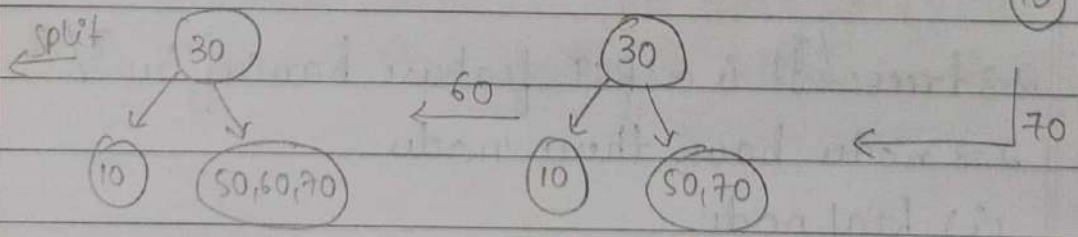
c<a<d<b<e

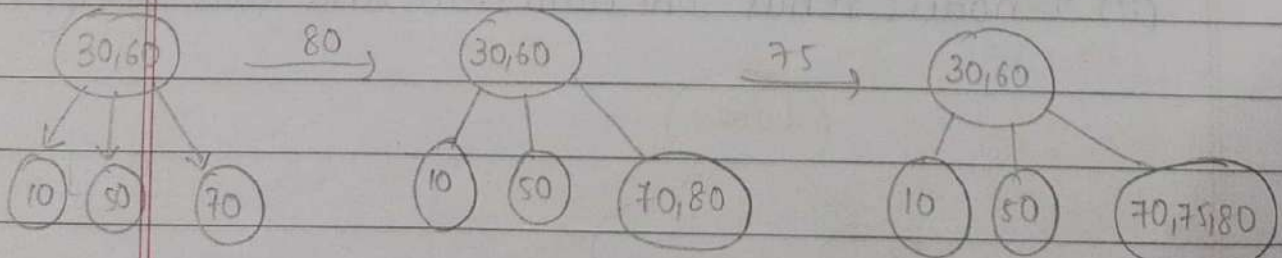
All leaf nodes are at same level. The time complexity for AVL tree is $\Theta(\log n)$. For 2,3 tree it is $O(\log n)$

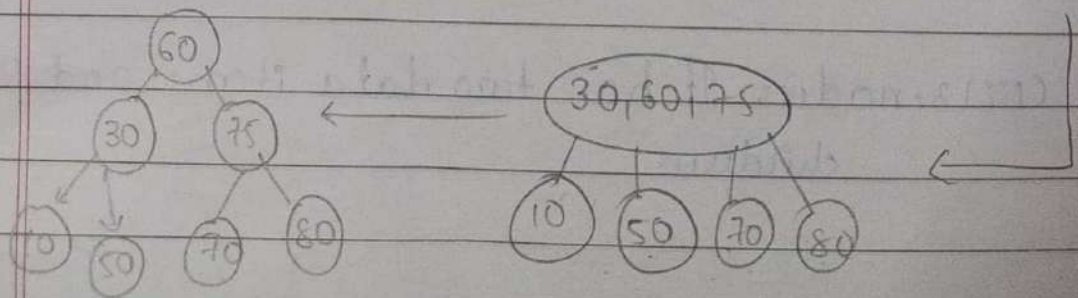
* Add the following to 2-3 tree

50, 30, 10, 70, 60 - - - //go on adding from leaf

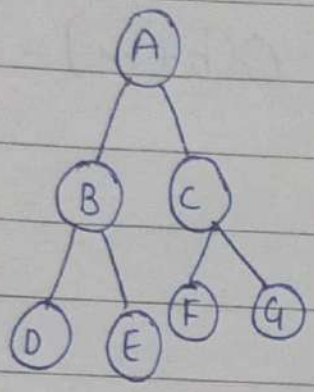






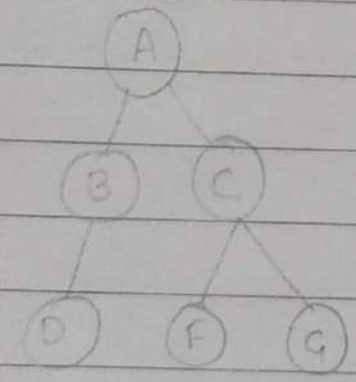


* Binary tree representation in array

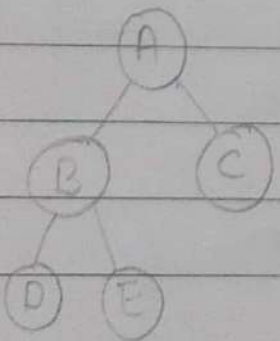


1	2	3	4	5	6	7
A	B	C	D	E	F	G

if node is at index i
its left child is @ 2 * i
its right child is @ 2 * i + 1
to find parent from i = $\left[\frac{i}{2}\right]$



A	B	C	D	-	F	G

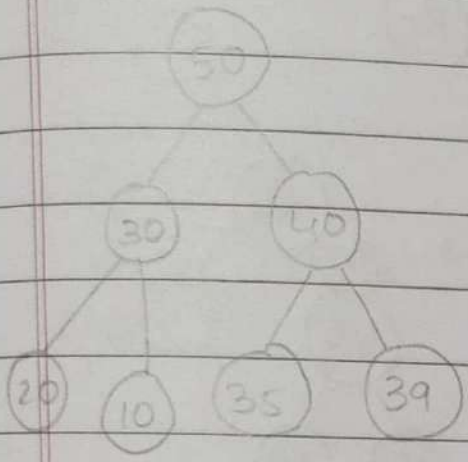


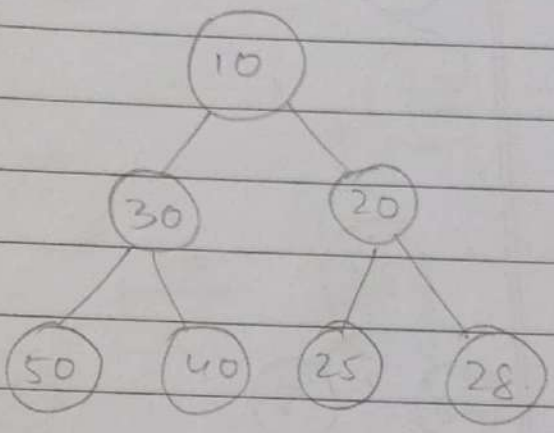
A	B	C	D	E	-	-

So complete binary tree is efficient

Heap:- Binary tree with keys assigned to node with following conditions.

1) tree's shape requirements (complete).
2) parental dominance.

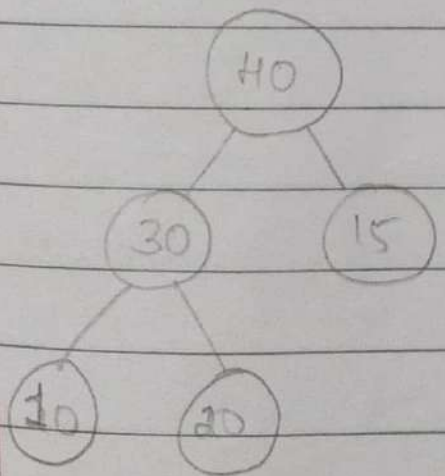




Max heap

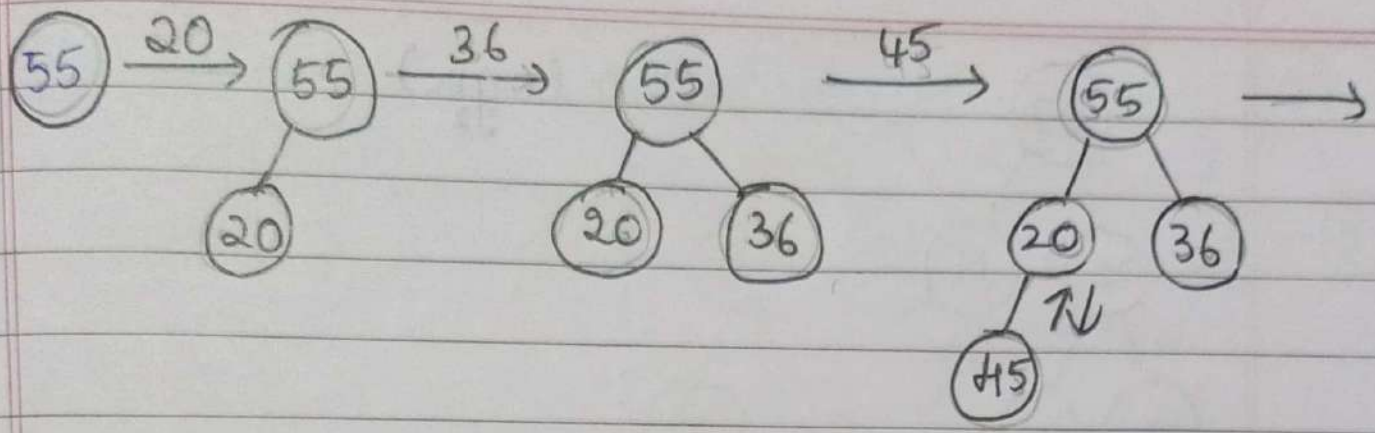
min heap

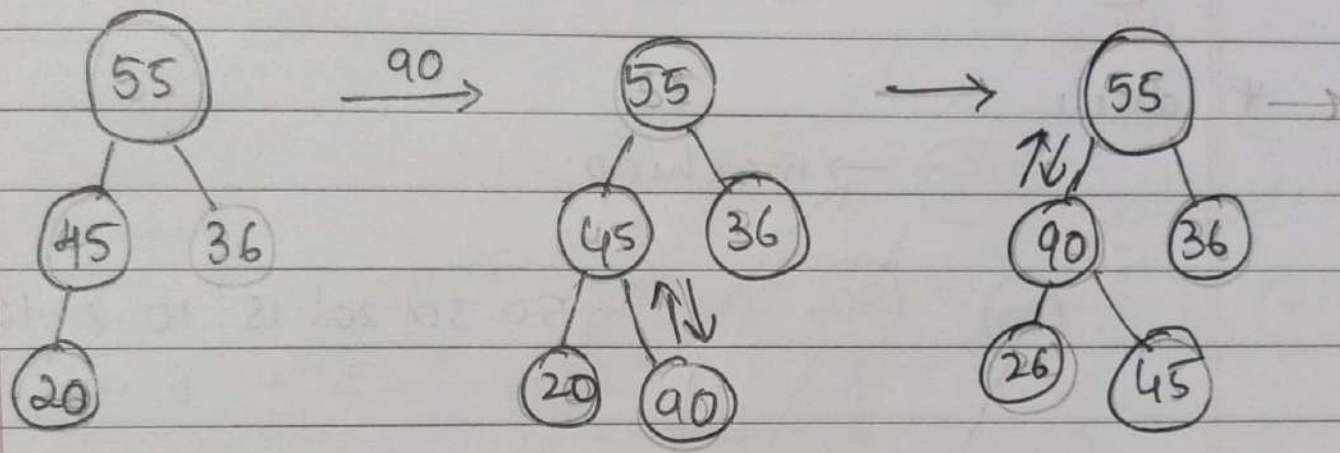
* 10, 20, 15, 30, 40 (max heap)

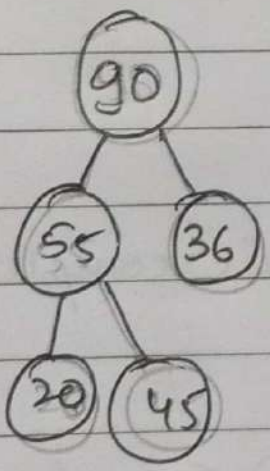


(actual value)

Eg: 55, 20, 36, 45, 90

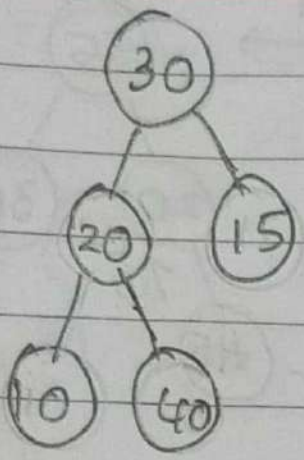






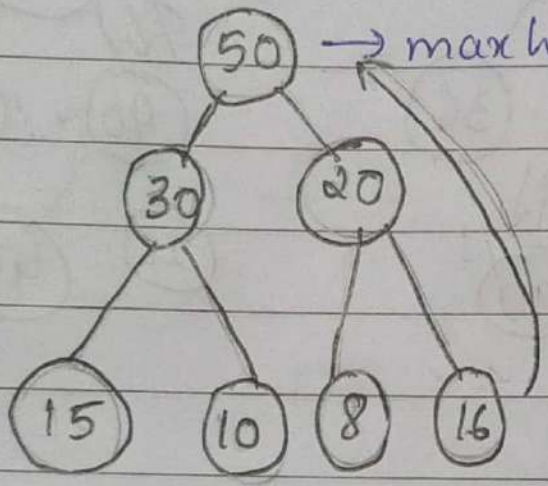
* 10, 20, 15, 30, 40

1	2	3	4	5
40	30	15	10	20
			10	



$O(n \log_2 n)$

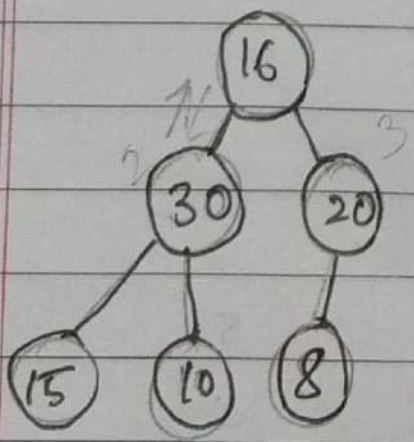
* delete



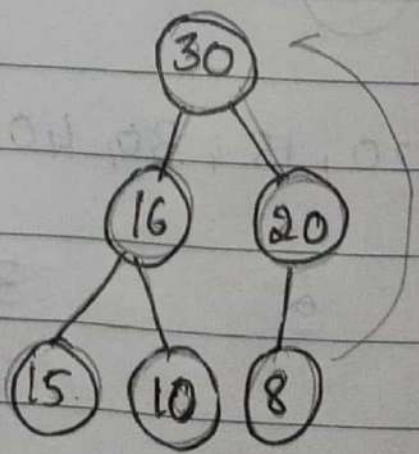
50	30	20	15	10	8	16
1	2	3	4	5	6	7

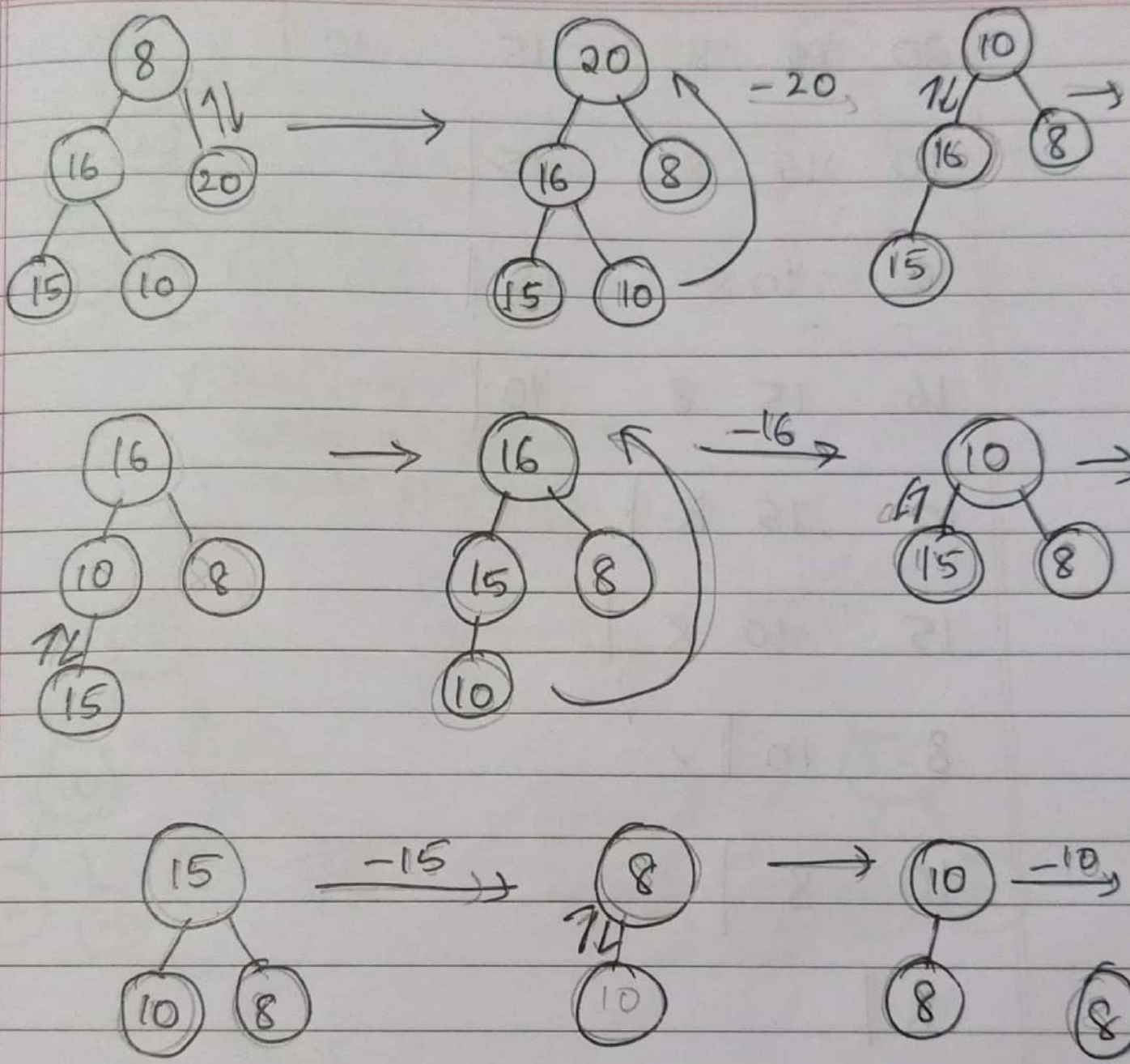
-50

We always delete root item
Replace rightmost to maintain complete binary tree



heapify





Array

50	30	20	15	10	8	16
16	30	20	15	10	8	
30	16	20	15	10	8	
8	16	20	15	10		

20 16 8 15 10 |

10 16 8 15 |

16 10 8 15 |

16 15 8 10 |

10 15 8 |

15 10 8 |

8 10 |

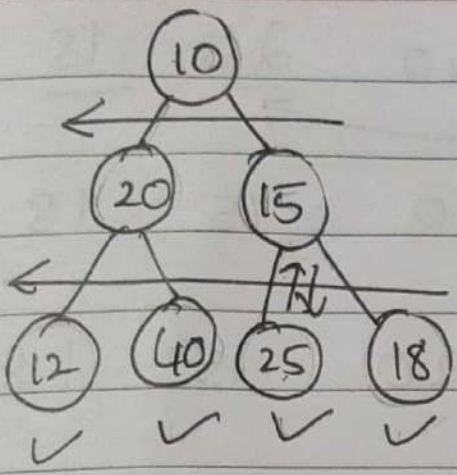
10 8 |

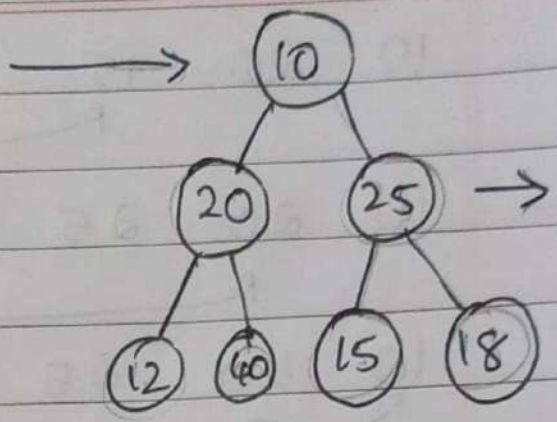
8 |

* Heapify method

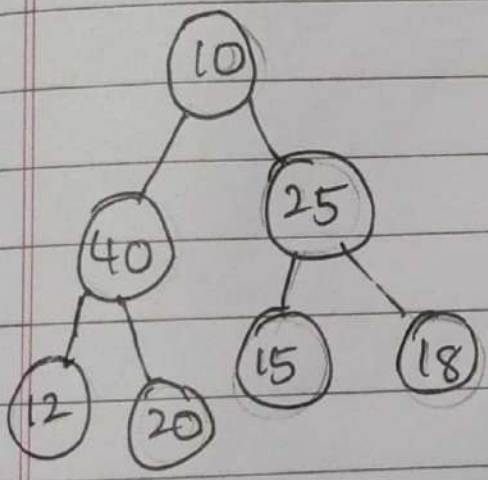
10, 20, 15, 12, 40, 25, 18

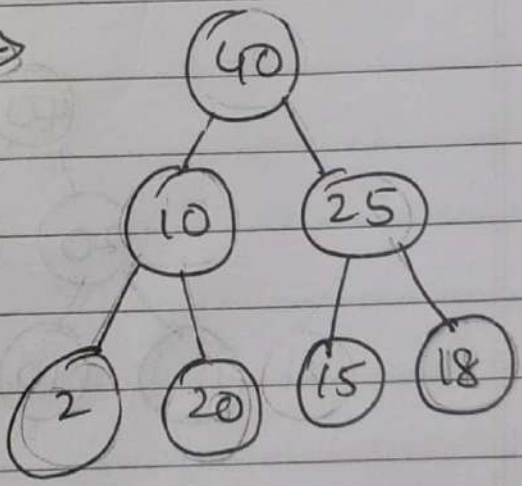
First fill all in complete binary tree

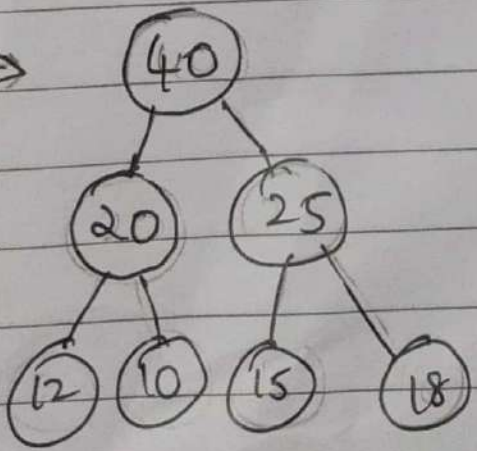




(leaf processed)
with no affect to
heap property







1	2	3	4	5	6	7
10	20	15	12	40	25	18
10	20	25	12	40	15	18
10	40	25	12	20	15	18
40	10	25	12	20	15	18
40	20	25	12	10	15	18