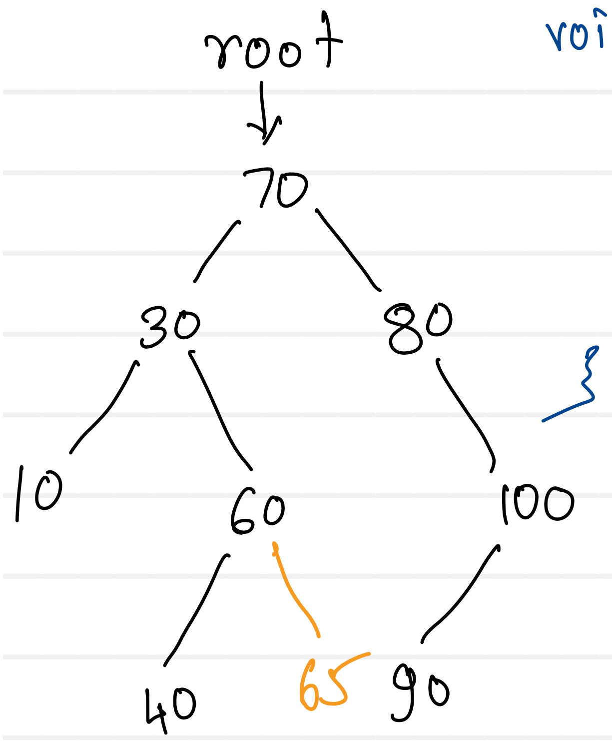
Sunbeam Institute of Information Technology
Pune and Karad

Module – Data Structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Add Node (Recursion)

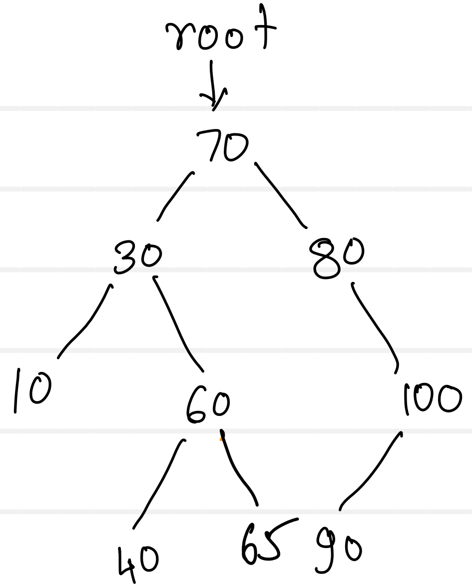



```
void addNode(int value) {
    if (root == null)
        root = new Node(value);
    else
        addNode( value , root);
}
```

value = 65
addNode(65)
addNode(65, &70)
addNode(65, &30)
addNode(65, &60)

```
void addNode(int value, Node trav) {
    if( value < trav. data) {
        if(trav. left == null) {
            trav. left = new Node(value);
            return;
        }
        else
            addNode(value, trav. left);
    }else {
        if(trav. right == null) {
            trav. right = new Node(value);
            return;
        }else
            addNode(value, trav. right);
    }
}
```

Binary Search (Recursive)

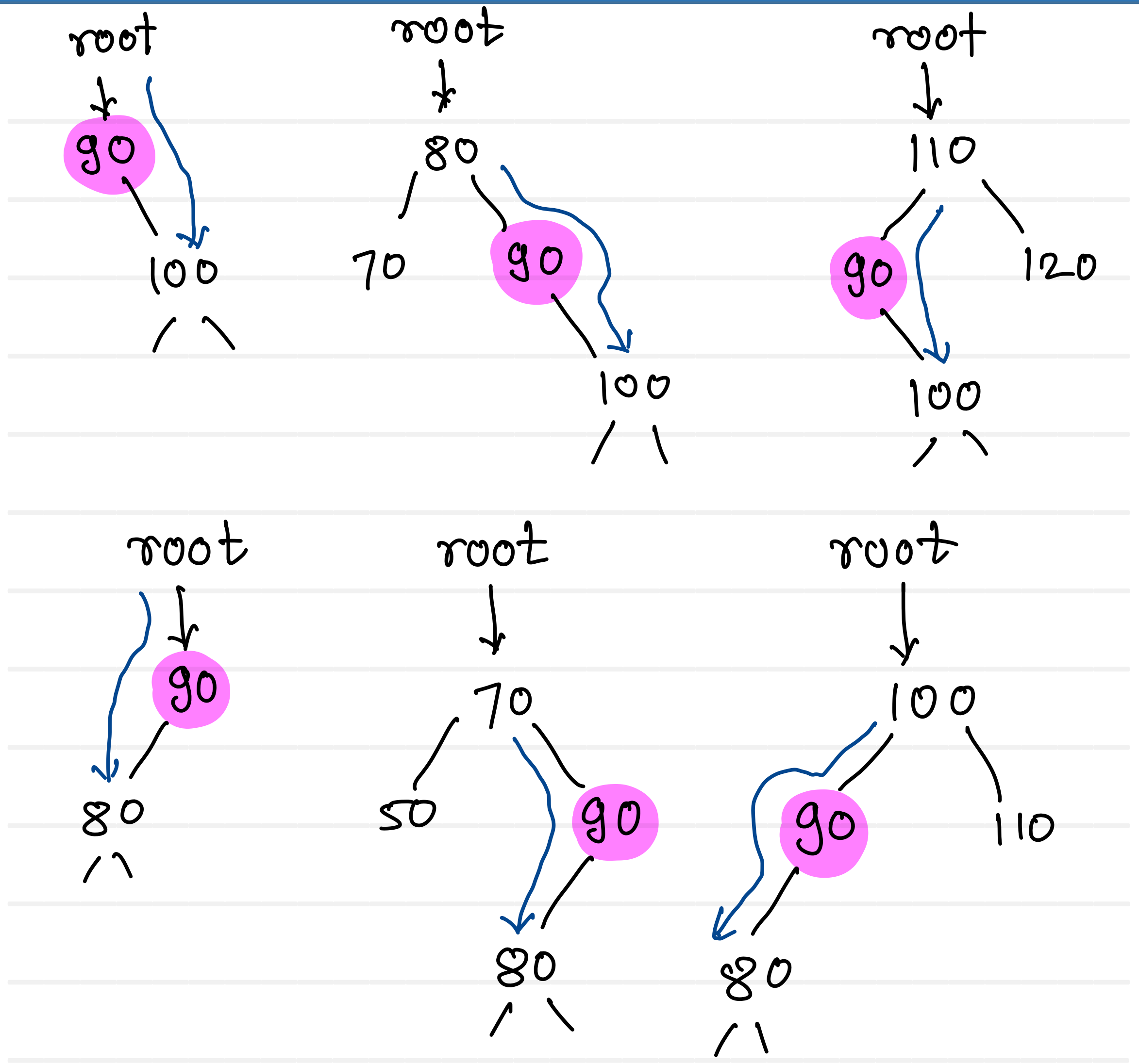


Key = 65

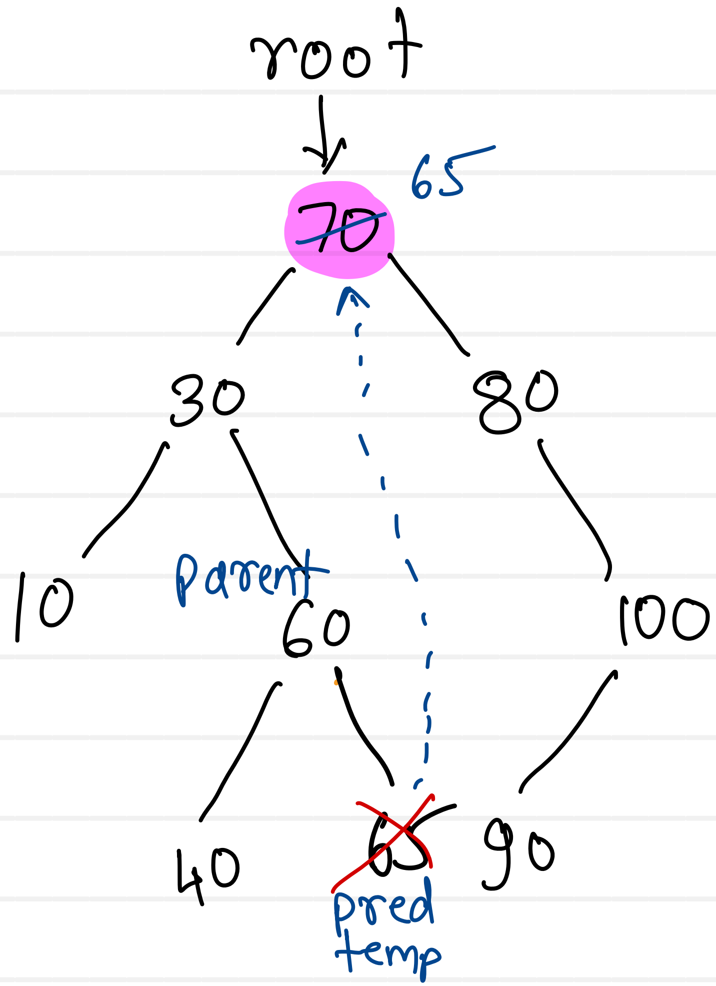
bSRec(65, &70)
bSRec(65, &30)
bSRec(65, &60)
bSRec(65, &65)

```
Node binarySearchRec(int key, Node trav) {
    if (trav == null)
        return null;
    if (key == trav.data)
        return trav;
    else if (key < trav.data)
        return binarySearchRec(key, trav.left);
    else
        return binarySearchRec(key, trav.right);
}
```

Delete Node



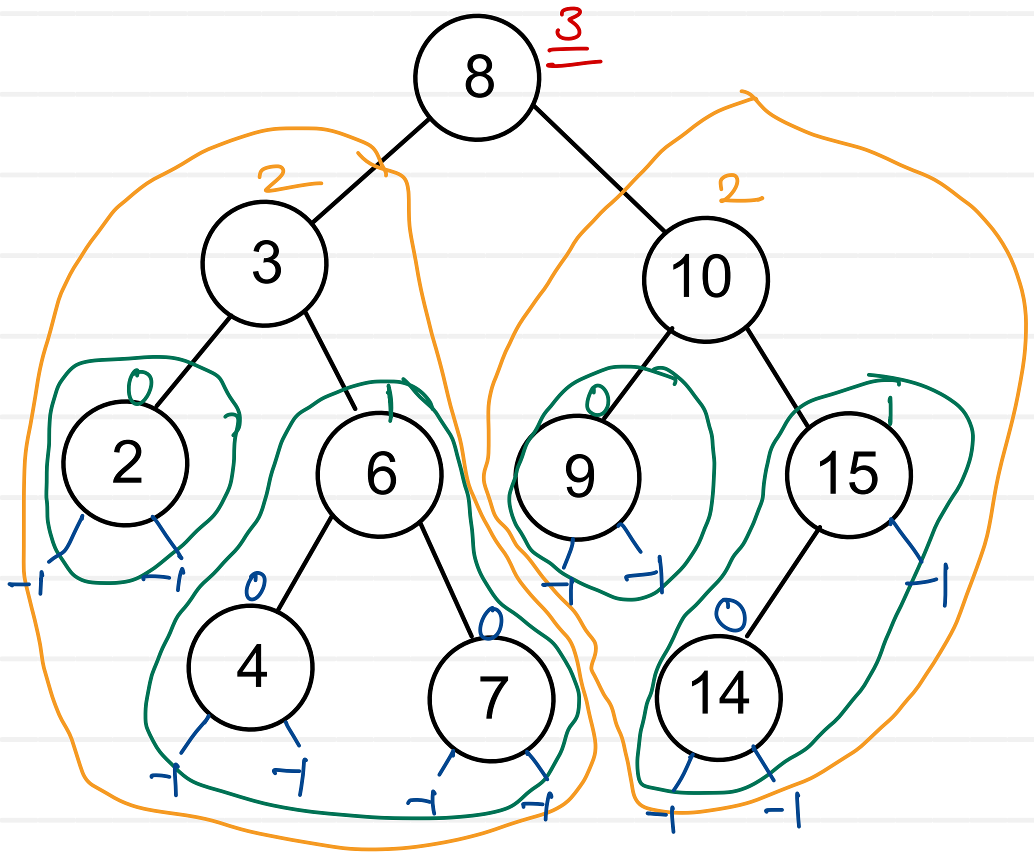
root
90
100
root
80
70
90
100
root
110
90
120
100
root
90
80
root
70
50
90
80
root
100
90
110
80



root
~~70~~
65
30
80
10
parent
60
100
40
~~65~~
90
pred
temp

Binary Search Tree - Height

Height of root = MAX (height (left sub tree), height (right sub tree)) + 1



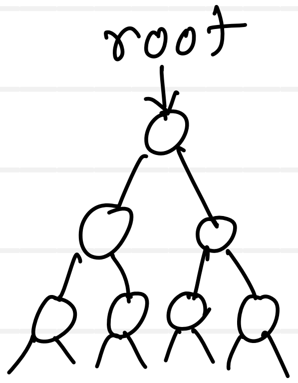
1. If left or right sub tree is absent then return -1
2. Find height of left sub tree
3. Find height of right sub tree
4. Find max height
5. Add one to max height and return

BST - Time complexity of operations

No. of elements = n
height of tree = h

$$\boxed{n = 2^{h+1} - 1}$$

h	n
-1	0
0	1
1	3
2	7
3	15
⋮	⋮



Add	:	O(h)	O(log n)
Search	:	O(h)	O(log n)
Delete	:	O(h)	O(log n)
Traverse	:		O(n)

$$n = 2^h$$

$$\log n = \log 2^h$$

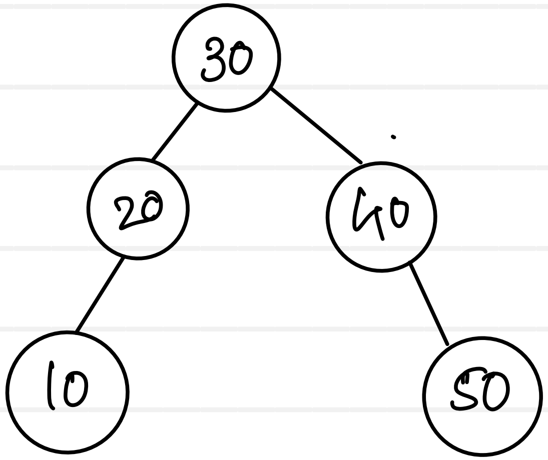
$$h = \frac{\log n}{\log 2}$$

$$\text{Time} \propto h$$

$$\text{Time} \propto \frac{\log n}{\log 2}$$

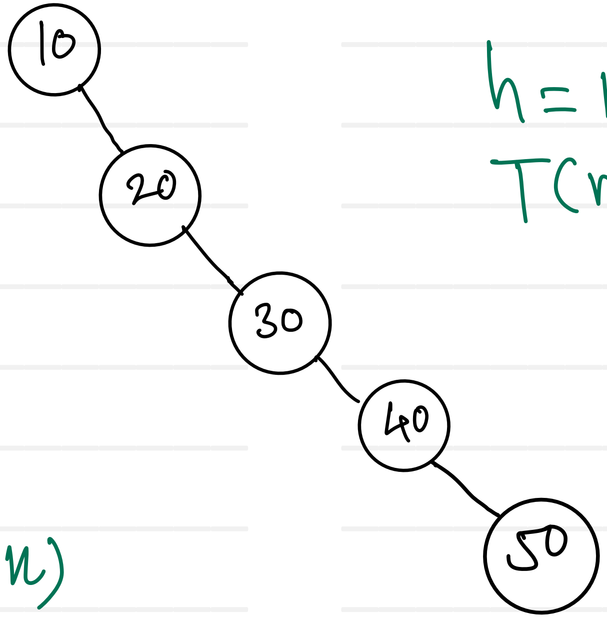
Skewed Binary Search Tree

Keys : 30, 40, 20, 50, 10



$h = \log n$ $\quad T(n) = O(\log n)$

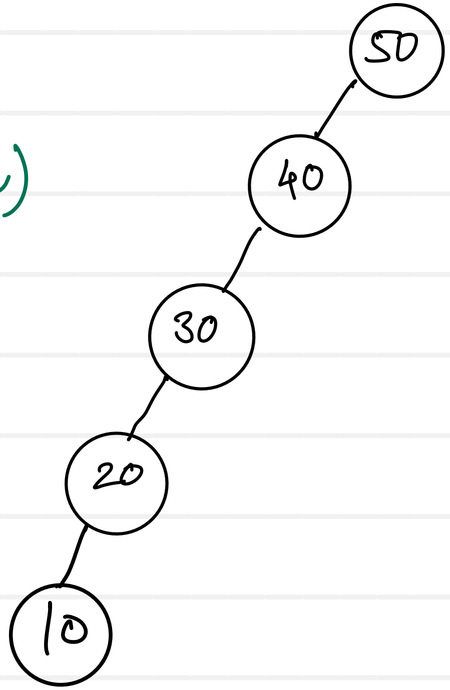
Keys : 10, 20, 30, 40, 50



$h = n$

$T(n) = O(n)$

Keys : 50, 40, 30, 20, 10



- In binary tree if only left or right links are used, tree grows only in one direction such tree is called as skewed binary tree
  - Left skewed binary tree
  - Right skewed binary tree
- Time complexity of any BST is O(h)
- Skewed BST have maximum height ie same as number of elements.
- Time complexity of searching is skewed BST is O(n)

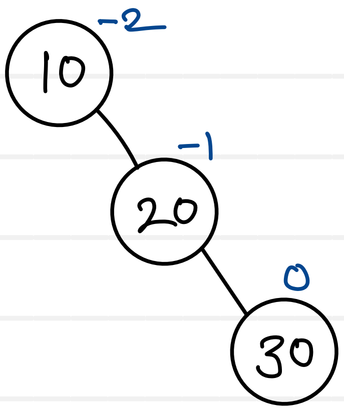
Balanced BST

- To speed up searching, height of BST should be minimum as possible
- If nodes in BST are arranged, so that its height is kept as less as possible, is called as Balanced BST

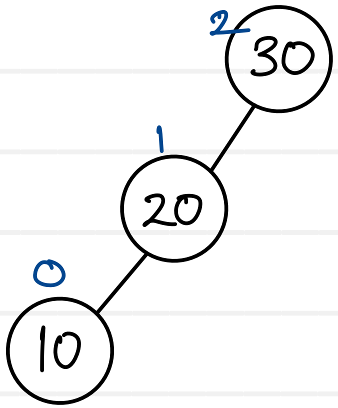
Balance factor (nodes) = Height (left sub tree) - Height (right sub tree)

- tree is balanced if balance factors of all the nodes is either -1, 0 or +1
- balance factors = {-1, 0, +1}
- A tree can be balanced by applying series of left or right rotations on imbalance nodes (node having balance factor other than -1, 0 or +1)

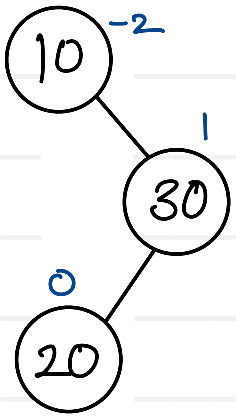
Keys : 10, 20, 30



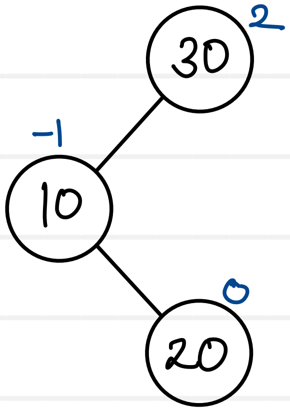
Keys : 30, 20, 10



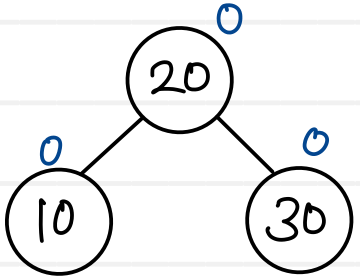
Keys : 10, 30, 20



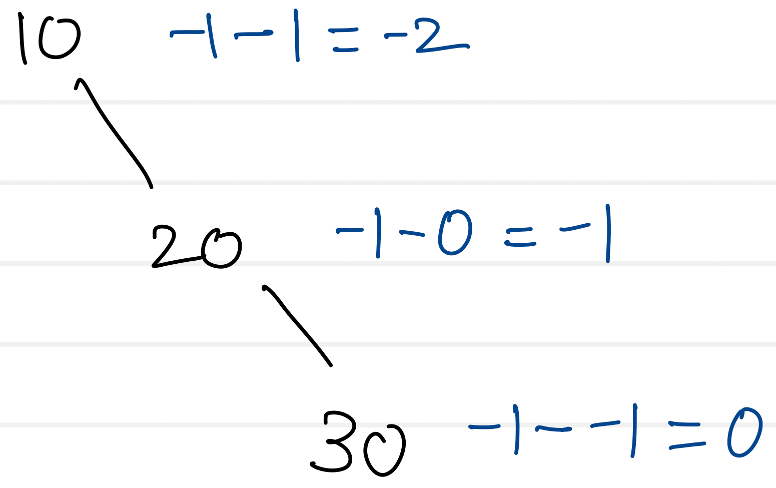
Keys : 30, 10, 20

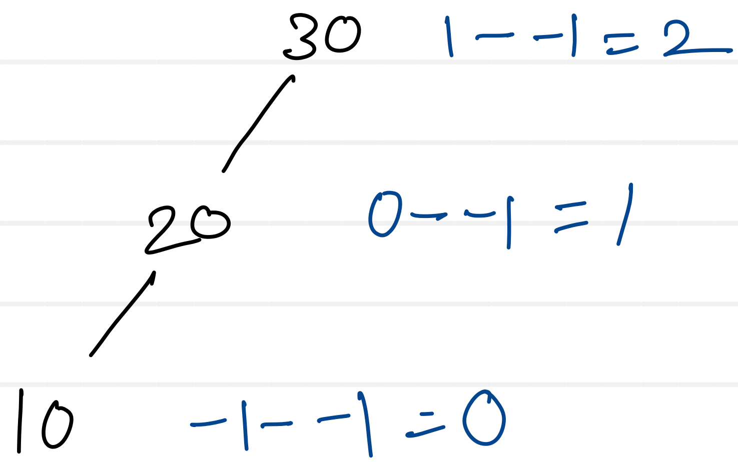


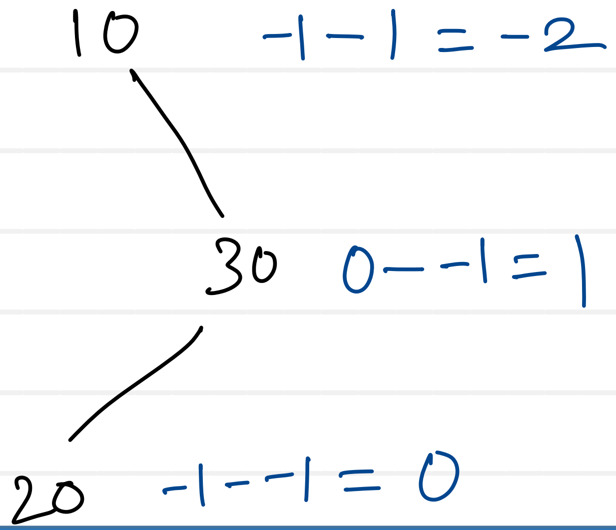
Keys : 20, 10, 30
Keys : 20, 30, 10

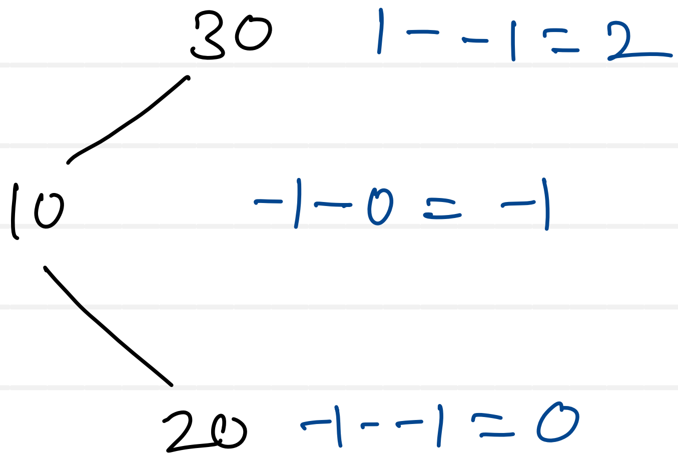


Balanced BST

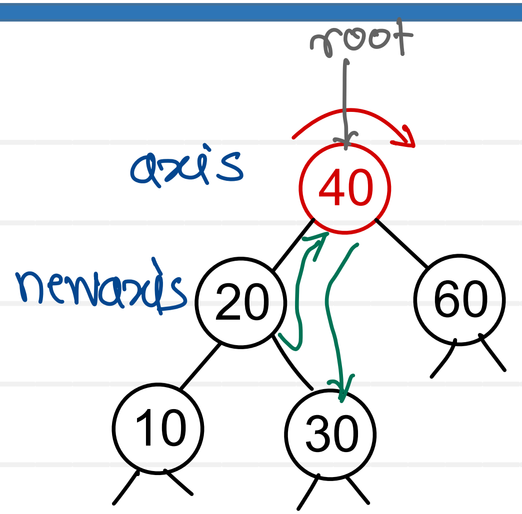

10
-1-1=-2
20
-1-0=-1
30
-1--1=0

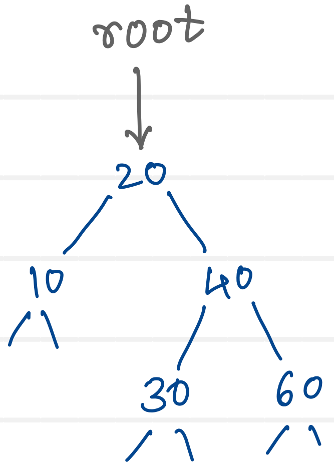

30
1--1=2
20
0--1=1
10
-1--1=0

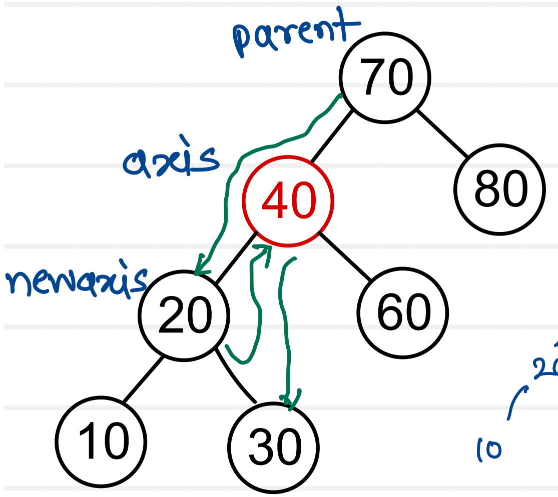

10
-1-1=-2
30
0--1=1
20
-1--1=0

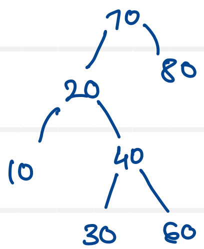

30
1--1=2
10
-1-0=-1
20
-1--1=0

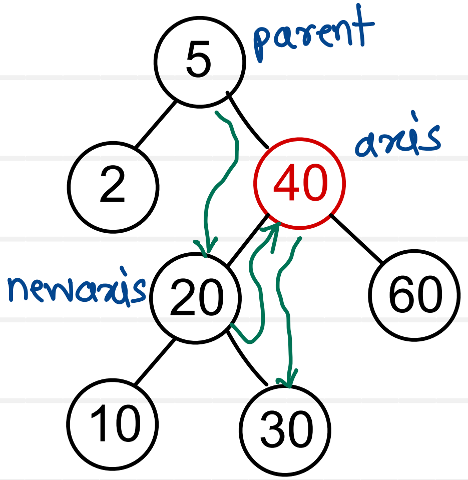
Right Rotation

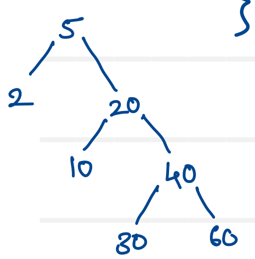






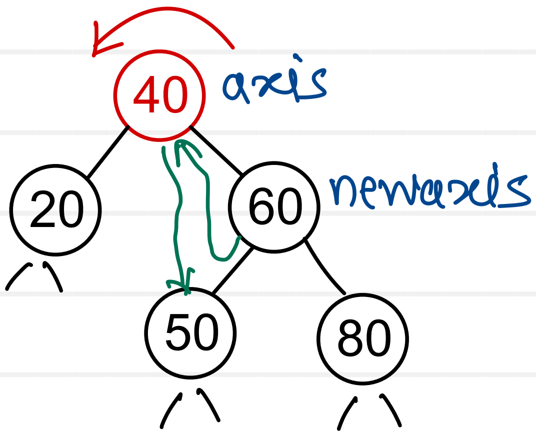


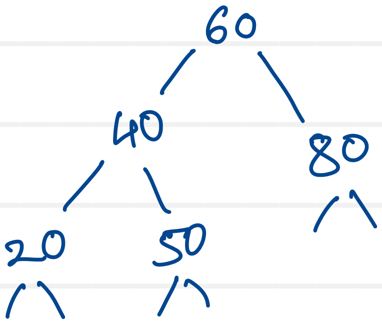




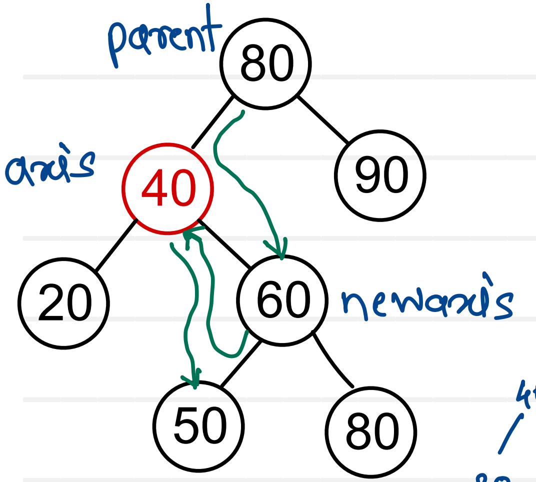

```
rightRotation(Node axis, Node parent){
    newaxis = axis.left;
    axis.left = newaxis.right;
    newaxis.right = axis;
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis
}
```

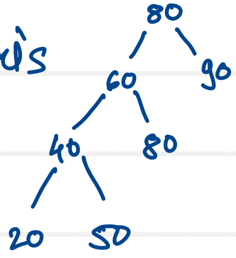
Left Rotation

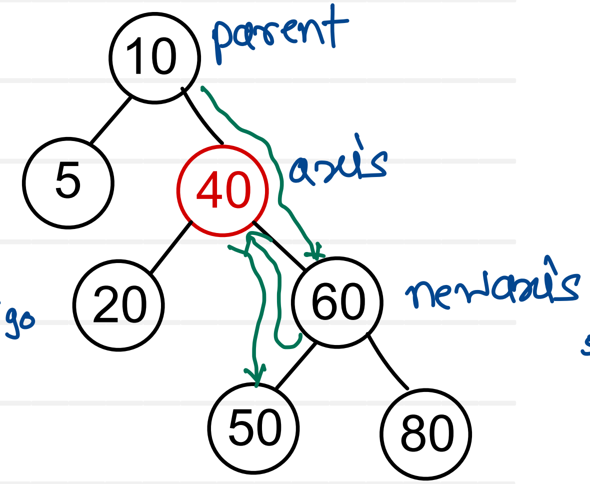


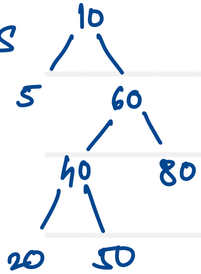



```
leftRotation(Node axis, Node parent){
    newaxis = axis.right;
    axis.right = newaxis.left;
    newaxis.left = axis;
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```







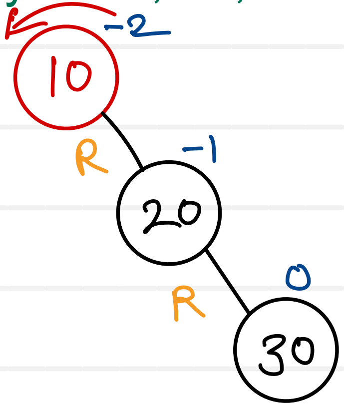


Rotation cases

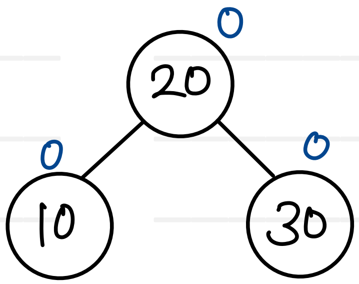
RR Imbalance

Single Rotation

Keys : 10, 20, 30

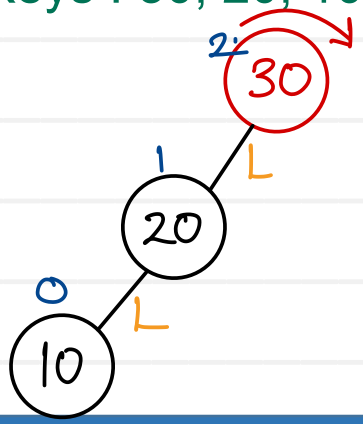


Left Rotation

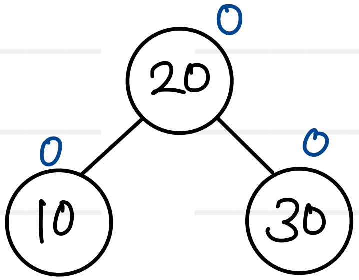


LL Imbalance

Keys : 30, 20, 10



Right Rotation

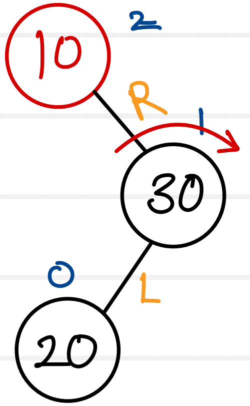


Rotation cases

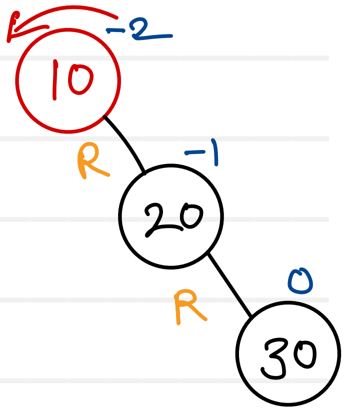
RL Imbalance

Double Rotation

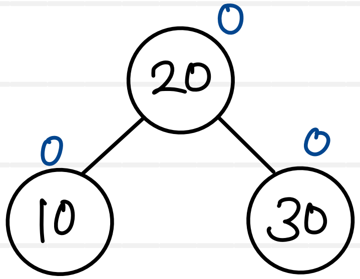
Keys : 10, 30, 20



Right Rotation

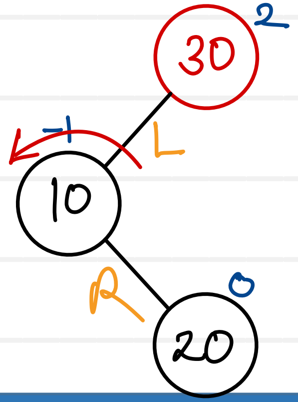


Left Rotation

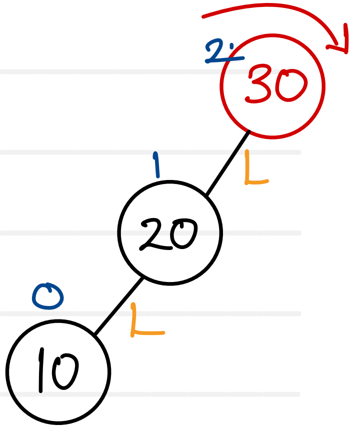


LR Imbalance

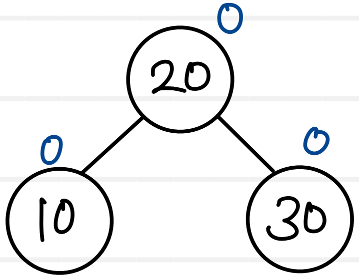
Keys : 30, 10, 20



Left Rotation



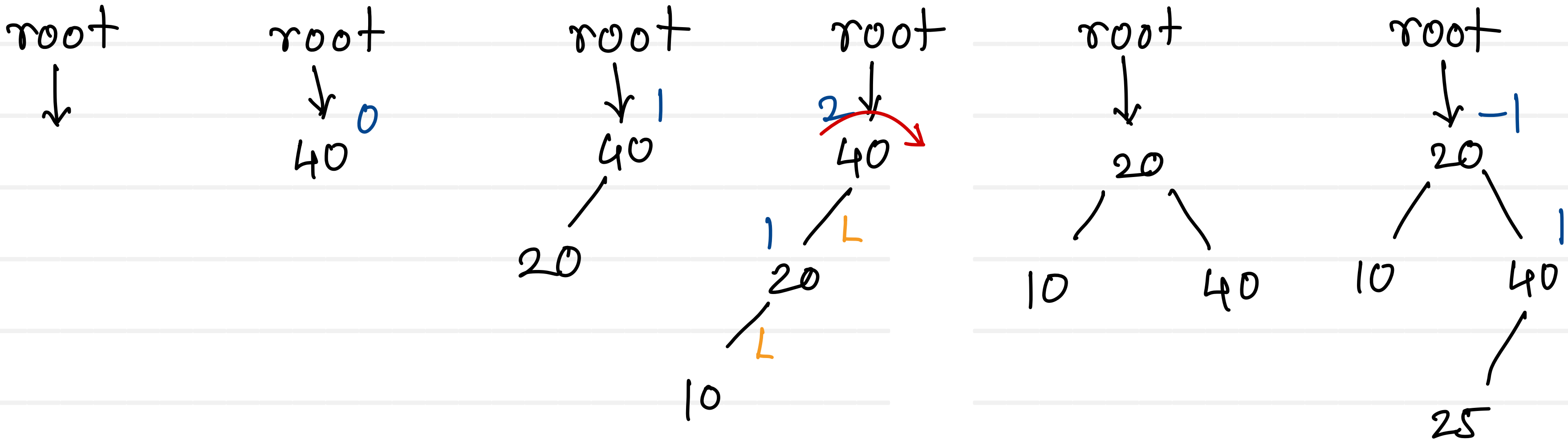
Right Rotation



AVL Tree

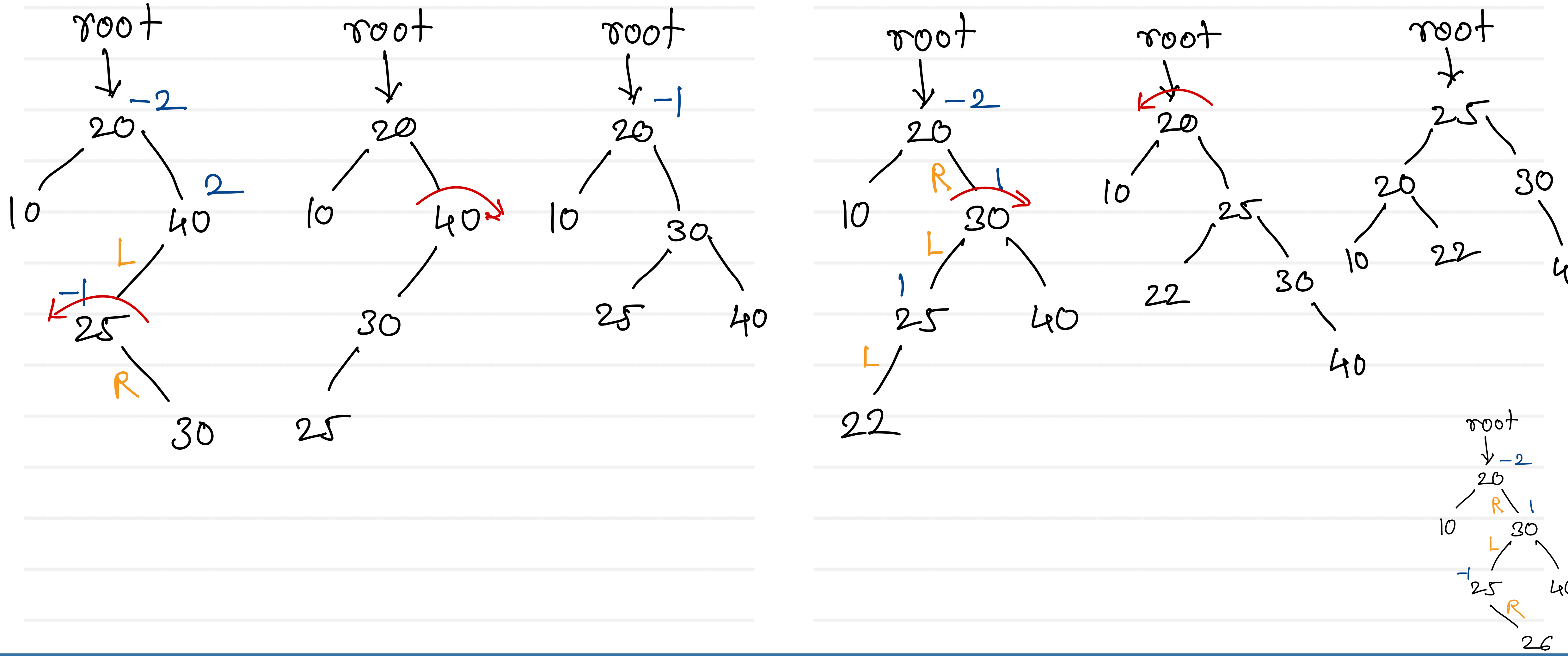
- self balancing binary search tree
- on every insertion and deletion of a node, tree is getting balanced by applying rotations on imbalance nodes
- The difference bet heights of left and right sub trees can not be more than one for all nodes
- Balance factors of all the nodes are either -1 , 0 or +1
- All operations of AVL tree are performed in O(log n) time complexity

Keys : 40, 20, 10, 25, 30, 22, 50



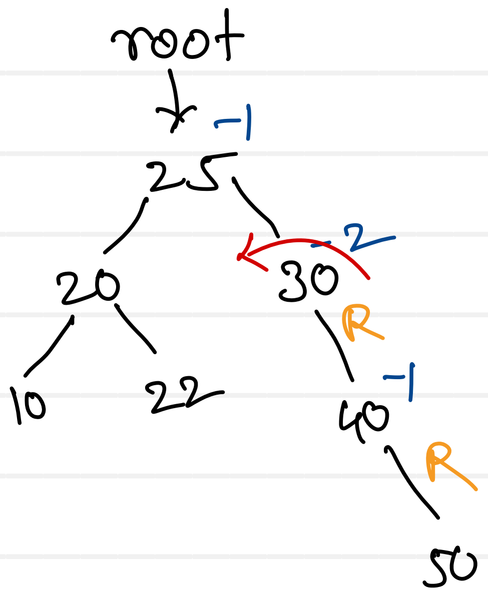
AVL Tree

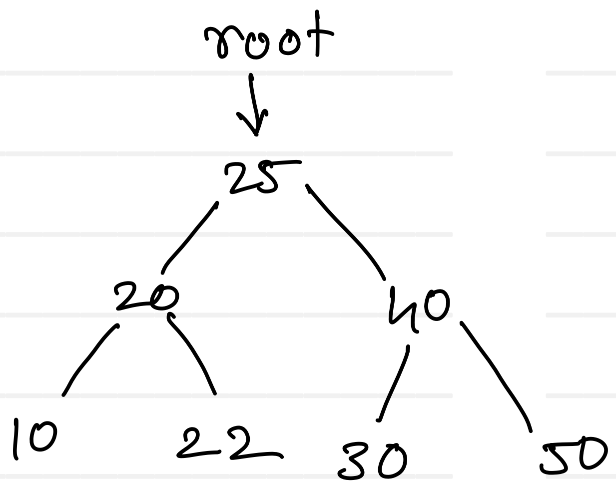
Keys : 40, 20, 10, 25, 30, 22, 50



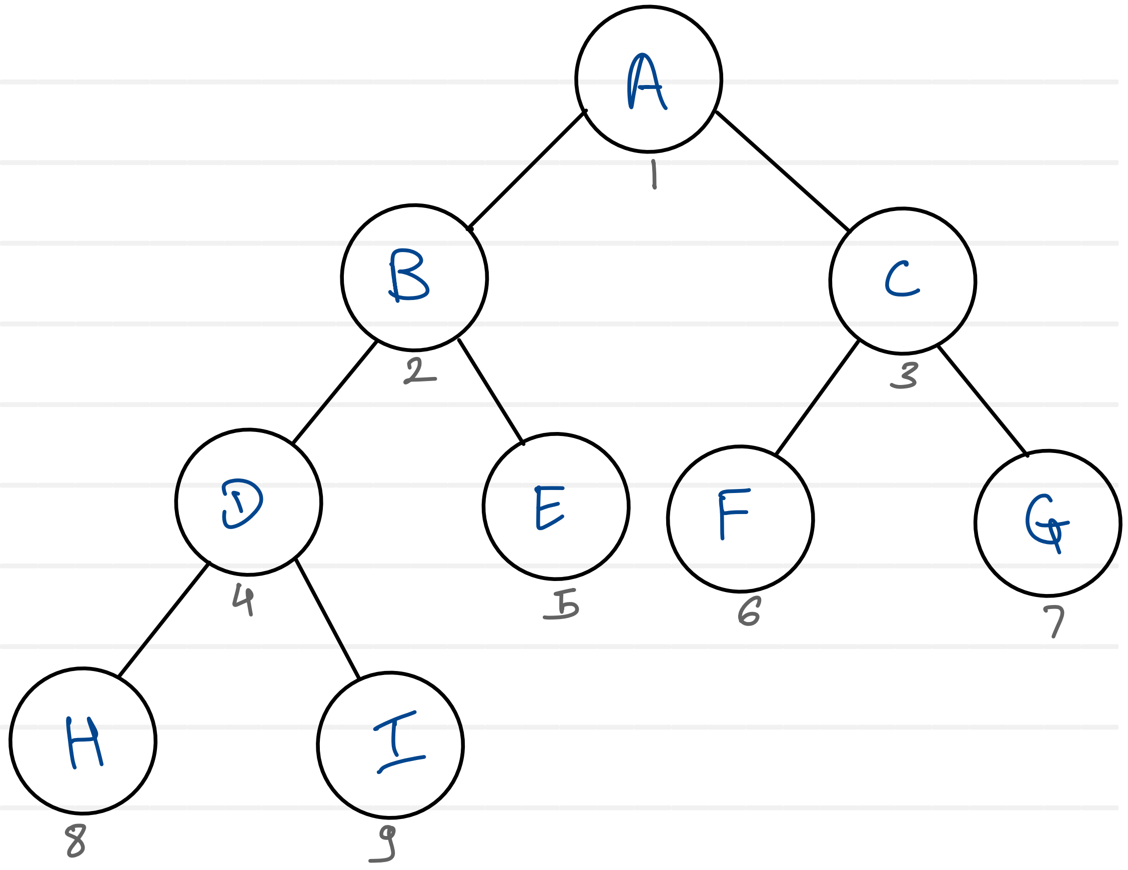
AVL Tree

Keys : 40, 20, 10, 25, 30, 22, 50





Almost Complete Binary Tree or Heap



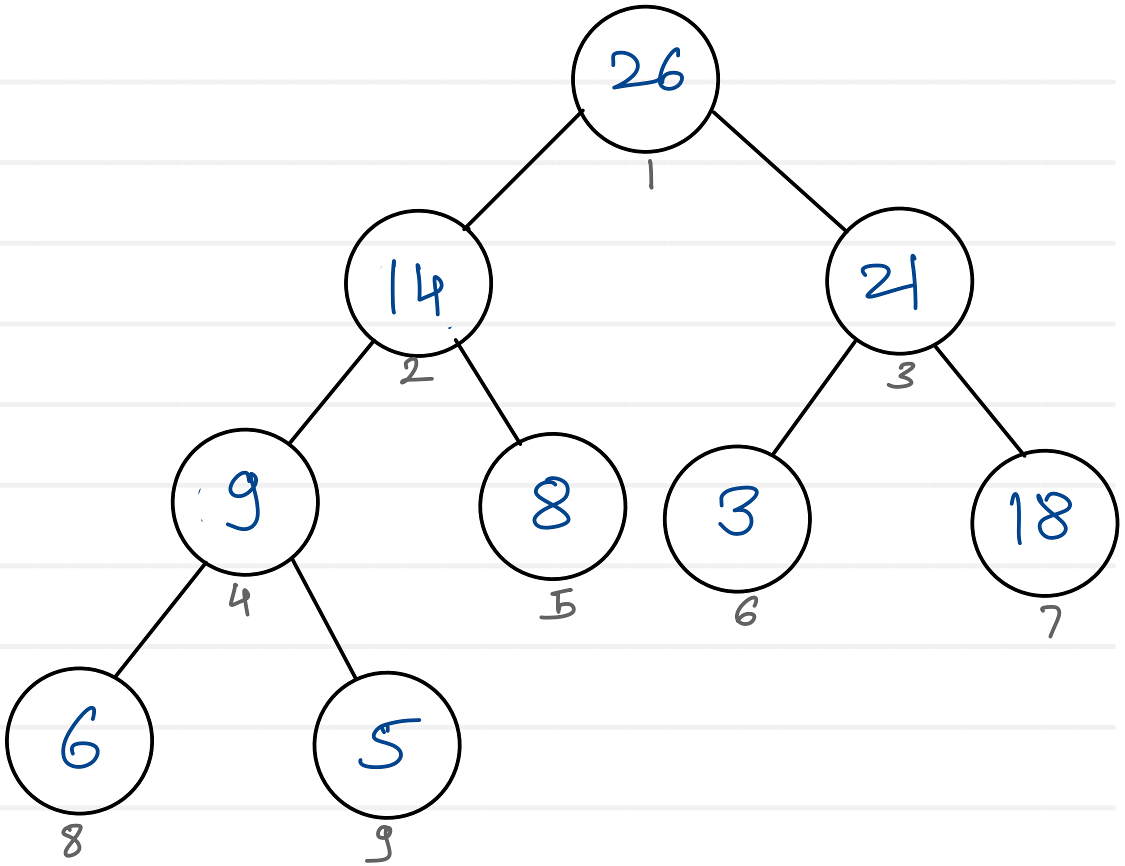
A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9
p					l	r		

- Almost Complete Binary Tree (height = h)
- All leaf nodes must be at level h or h-1
- All leaf nodes at level h must aligned as left as possible

- Array implementation of Almost Complete Binary Tree is called as heap

Node - i
Parent - $i / 2$
left child - $i * 2$
Right child - $i * 2 + 1$

Heap - Create heap (Add)



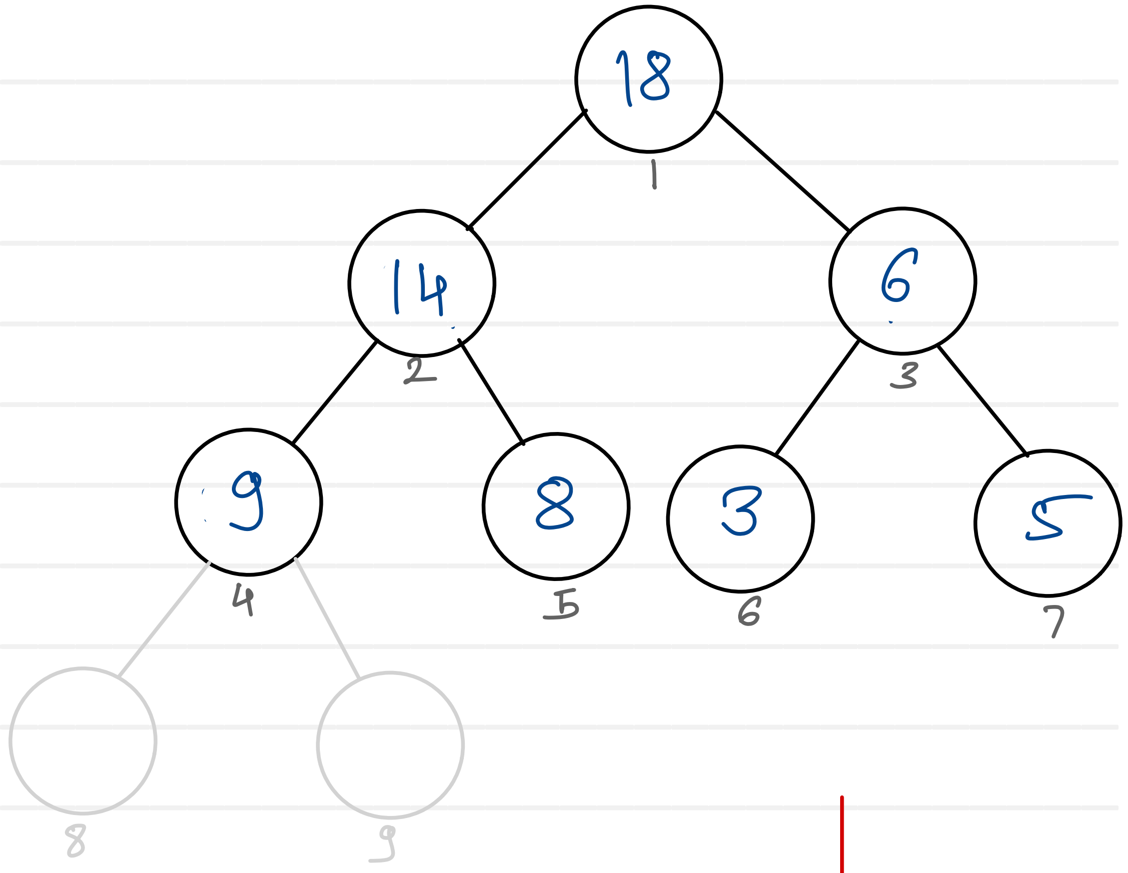
26	14	21	9	8	3	18	6	5
1	2	3	4	5	6	7	8	9

Keys : 6, 14, 3, 26, 8, 18, 21, 9, 5

1. Add new value at first index of array from left side
2. Adjust position of newly added value by comparing it with all its ancestors one by one.

$\boxed{T(n) = O(\log n)}$

Heap - Create heap (Delete)



18	14	6	9	8	3	5		
1	2	3	4	5	6	7	8	9

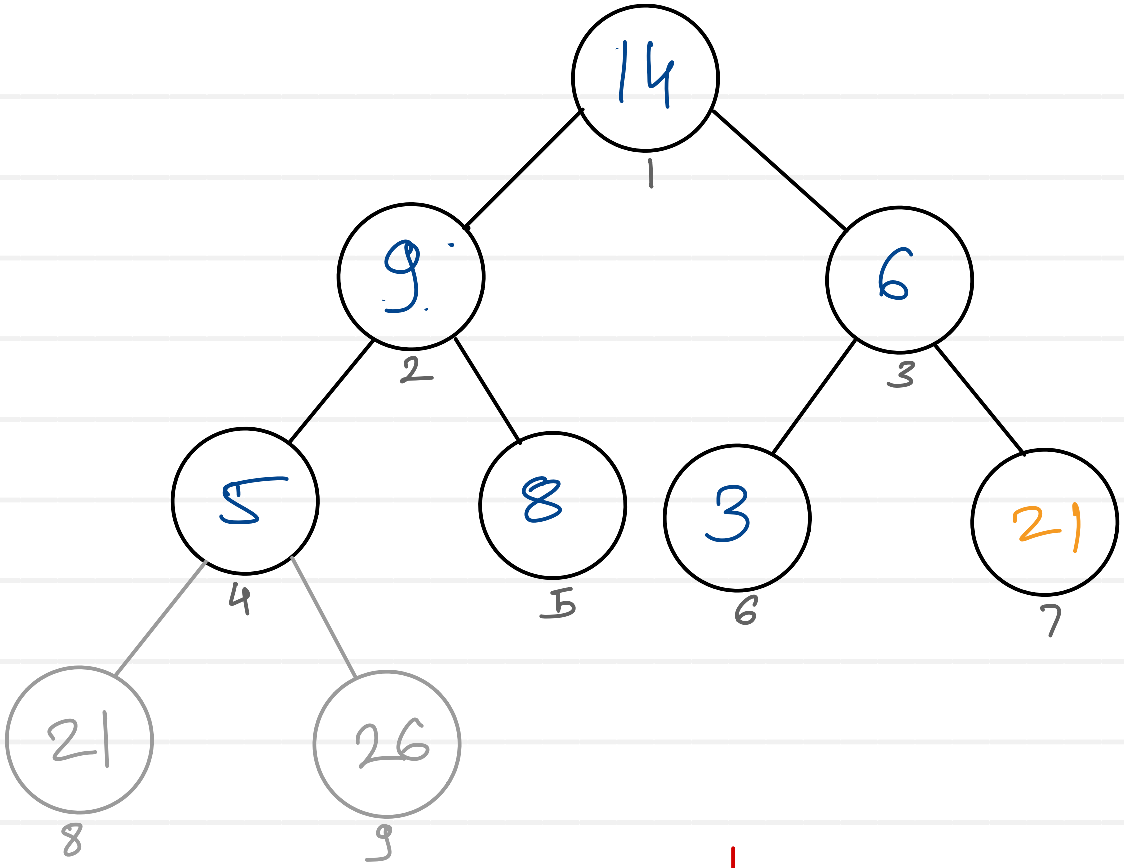
Max = 26

Max = 21

1. Place last element of heap at root position
2. Adjust position of root element by comparing it with all its descendents one by one

$$\boxed{T(n) = O(\log n)}$$

Heap sort



14	9	6	5	8	3	18	21	26
1	2	3	4	5	6	7	8	9

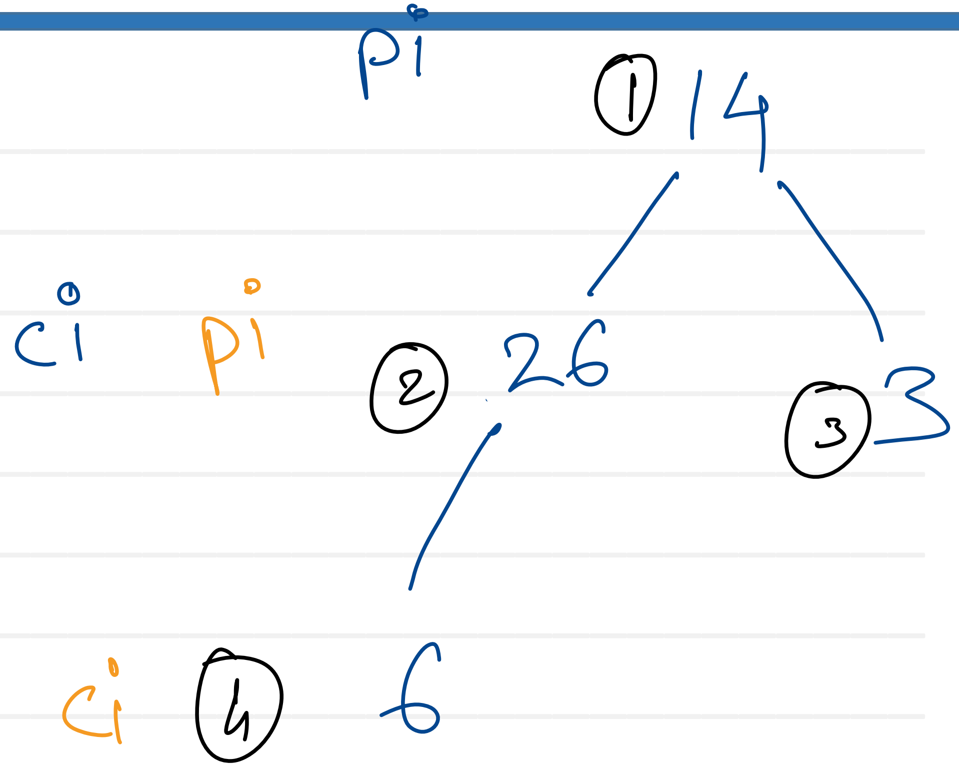
arr

6	14	3	26	8	18	21	9	5
1	2	3	4	5	6	7	8	9

1. make heap $\rightarrow n \log n$
2. delete heap $\rightarrow n \log n$

$2n \log n$

$T(n) = O(n \log n)$



ci	pi
4	2
2	1
1	~~0~~

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com