Sunbeam Institute of Information Technology
Pune and Karad

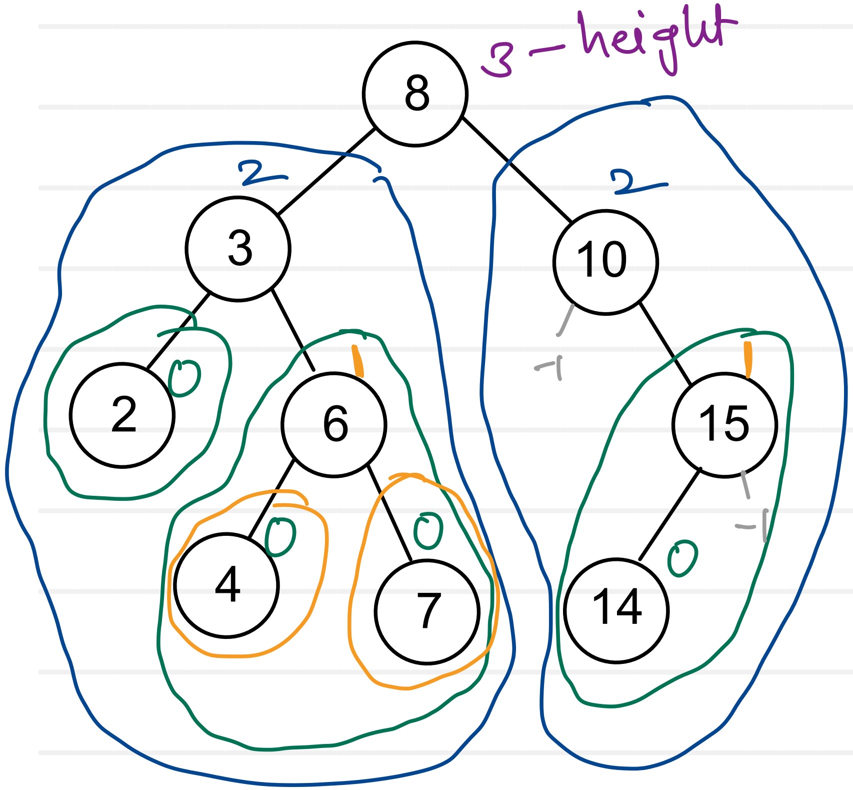
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

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

```
int height(Node trav) {
    if( trav == null)
        return -1;
    int hl = height( trav. left);
    int hr = height( trav.right);
    int max = hl > hr ? hl : hr;
    return max+1 ;
}
```

Height(NULL tree) = -1
Height(Leaf node) = 0

BST - Time complexity of operations

n - no. of nodes
h - height of tree

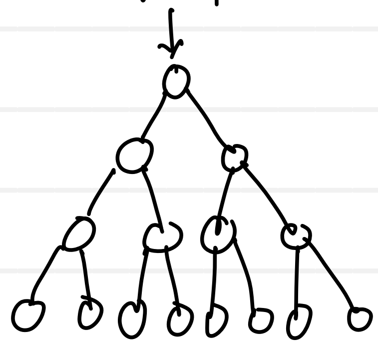
$$\boxed{n = 2^{h+1} - 1}$$

Add : $O(h)$ $O(\log n)$
Delete : $O(h)$ $O(\log n)$
Search : $O(h)$ $O(\log n)$

Traverse : $O(n)$

Capacity : max number of nodes for given height.

Height	No. of Node
-1	0
0	1
1	3
2	7
3	15



$2^h \approx n$

$\log 2^h = \log n$

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

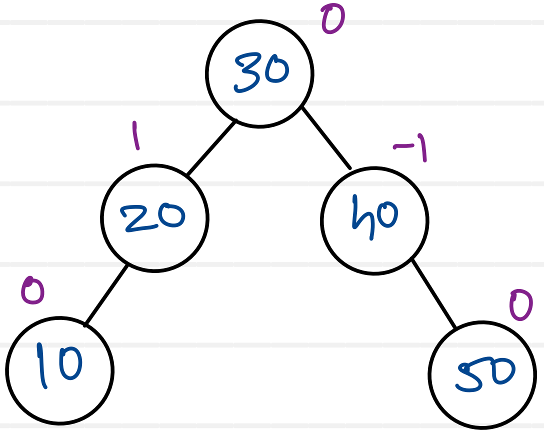
Time $\propto h$

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

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

Skewed Binary Search Tree

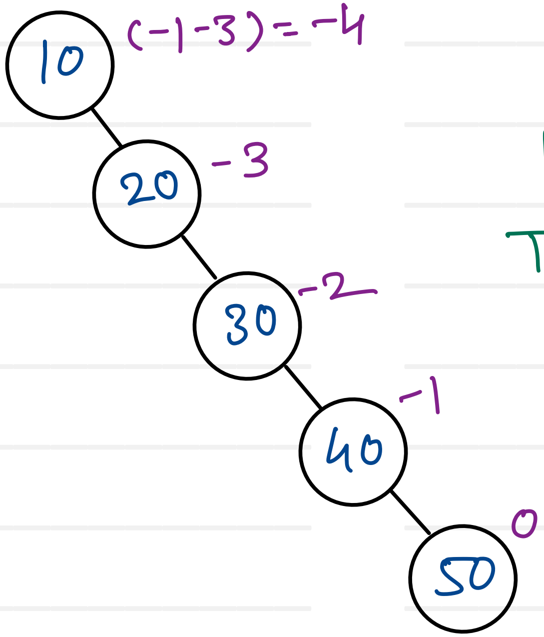
Keys : 30, 40, 20, 50, 10



$h = \log n$

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

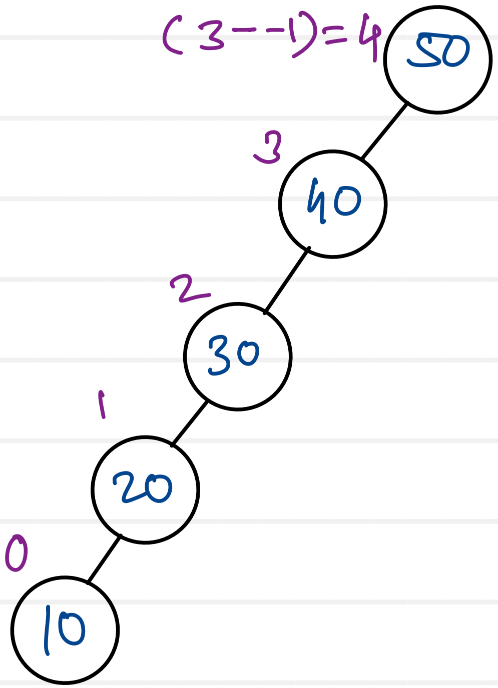
Keys : 10, 20, 30, 40, 50



$h = n$

$T(n) = O(n)$

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)

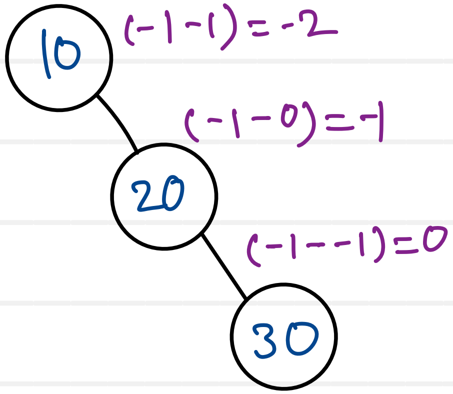
Balanced BST

- 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

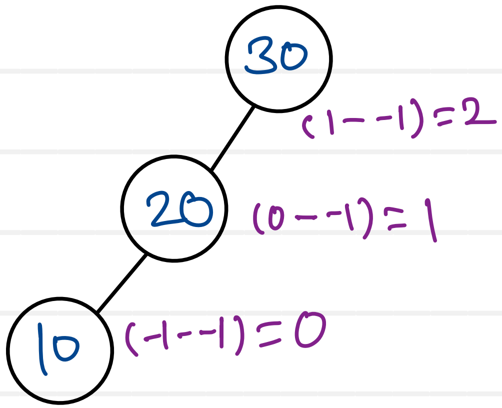
Balance factor = Height (left sub tree) - Height (right sub tree)

- 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, +1}

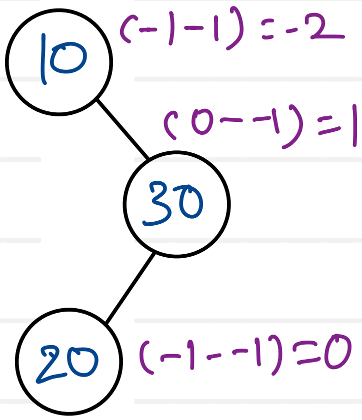
Keys : 10, 20, 30



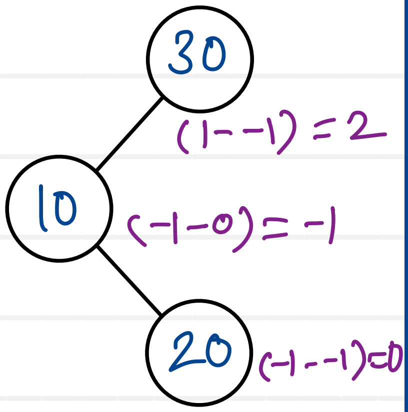
Keys : 30, 20, 10



Keys : 10, 30, 20

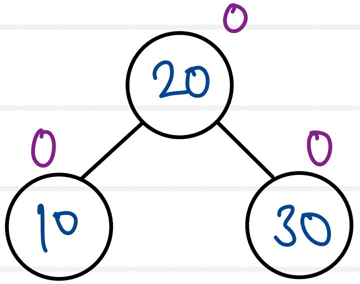


Keys : 30, 10, 20



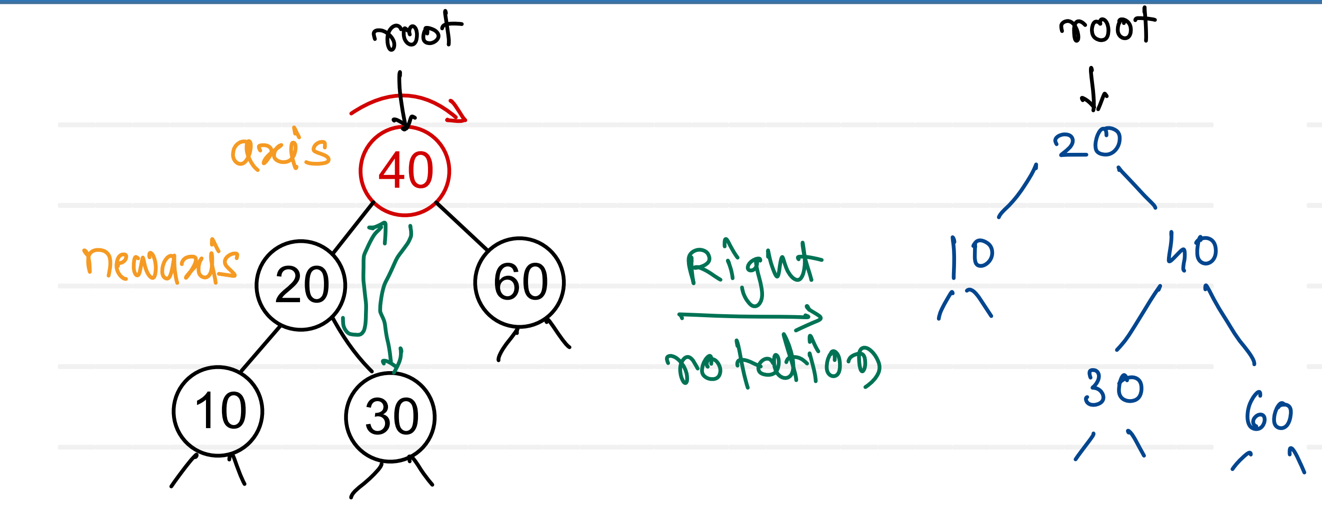
Keys : 20, 10, 30

Keys : 20, 30, 10

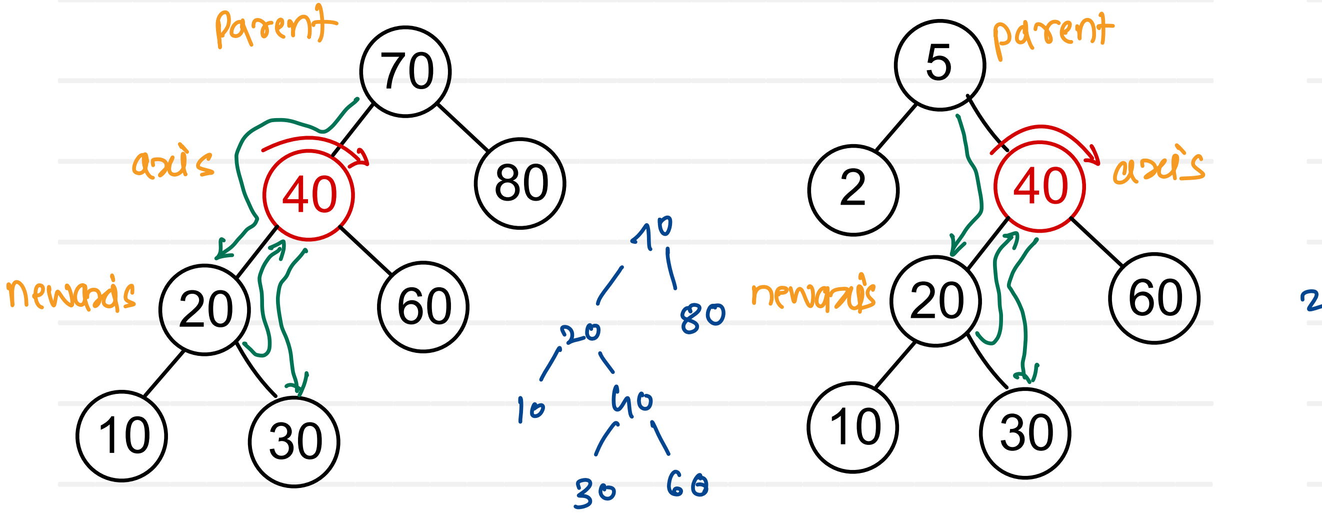


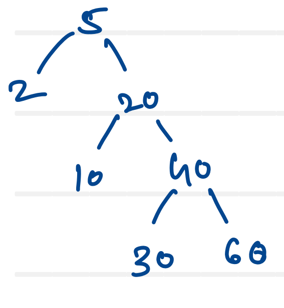
Balanced BST

Right Rotation

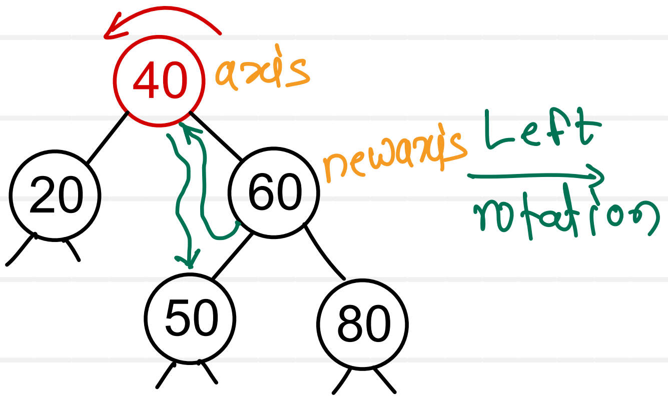


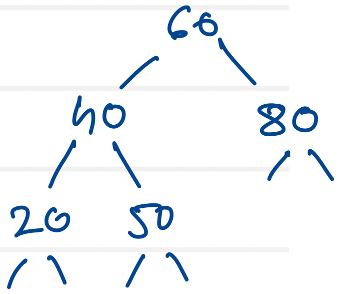

```
void rightRotation(axis, parent) {
    newaxis = axis.left;
    axis.left = newaxis.right;
    newaxis.right = axis;
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```



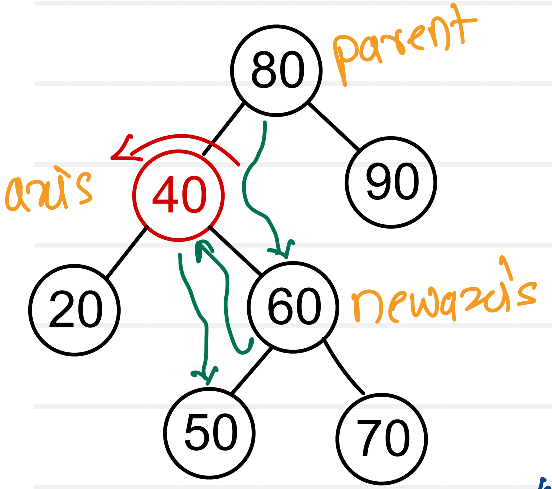


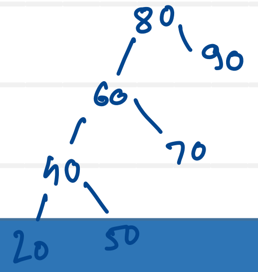
Left Rotation

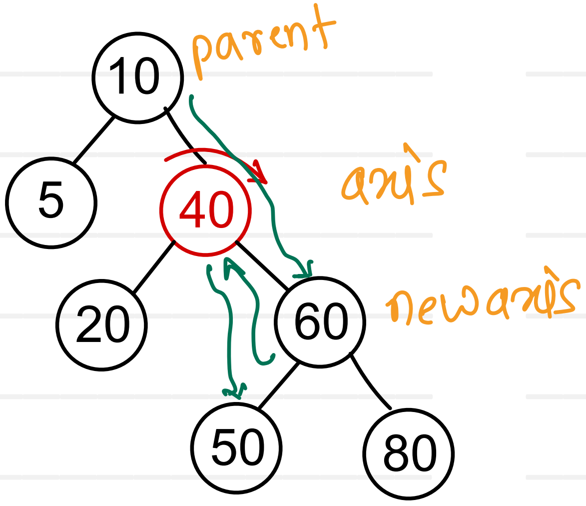


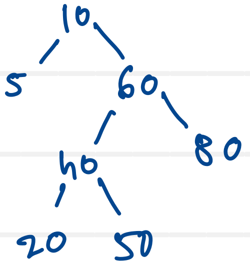



```
void leftRotation(axis, parent){
    newaxis = axis.right;
    axis.right = newaxis.left;
    newaxis.left = axis;
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```









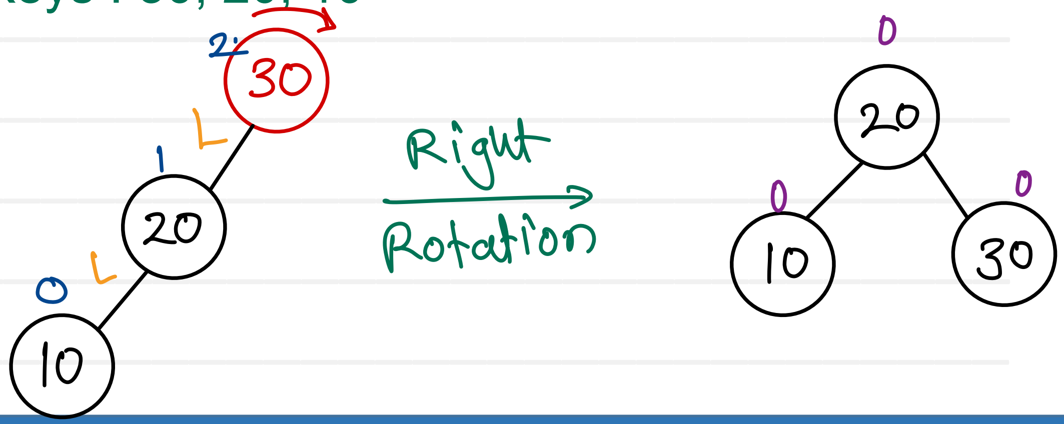
Rotation cases (Single Rotation)

RR Imbalance

Keys : 10, 20, 30

LL Imbalance

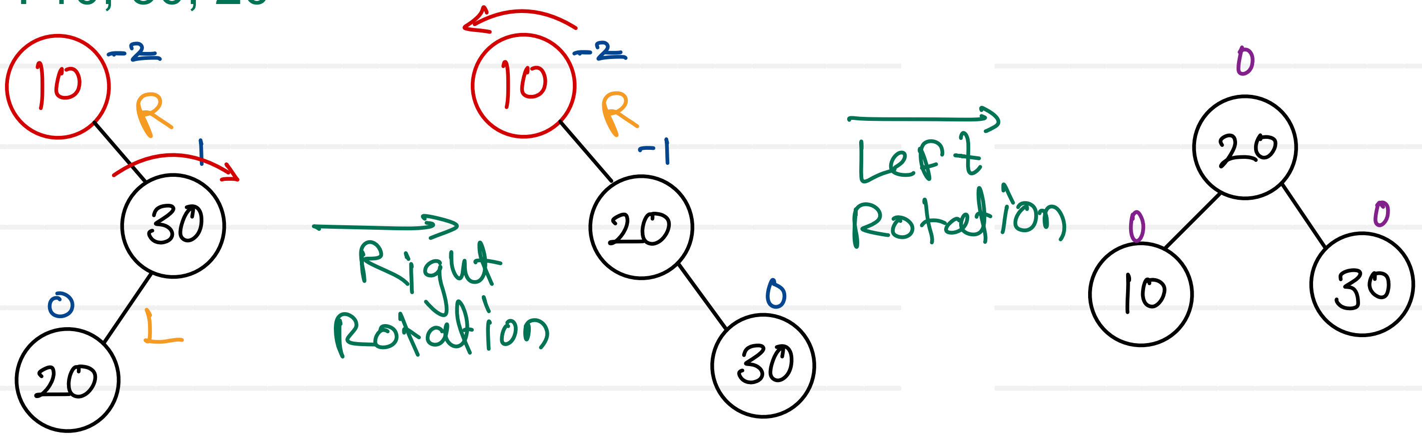
Keys : 30, 20, 10



Rotation cases (Double Rotation)

RL Imbalance

Keys : 10, 30, 20



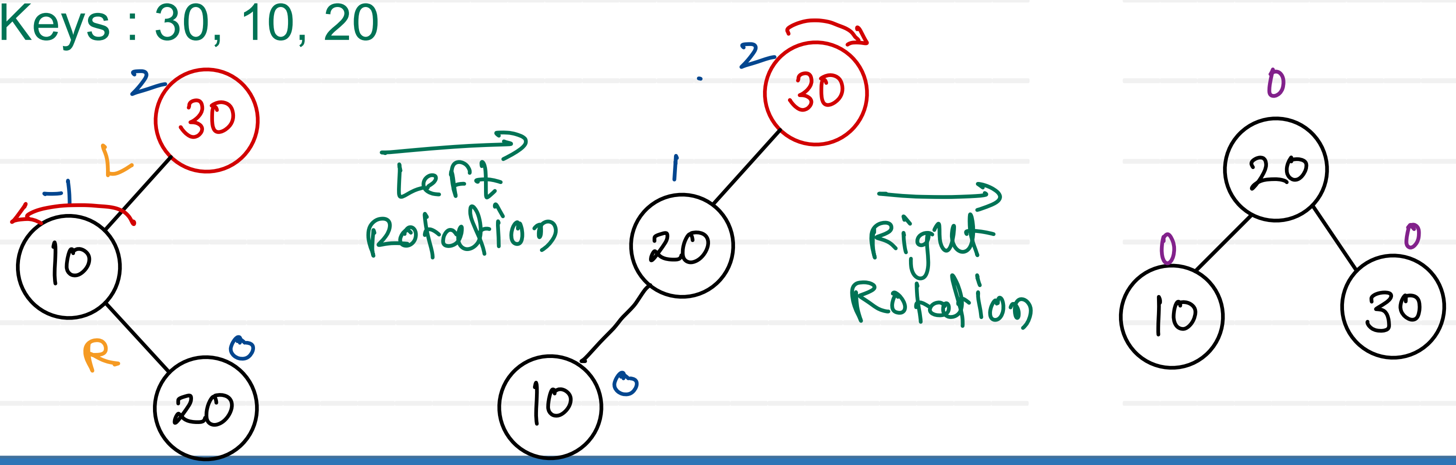
RL

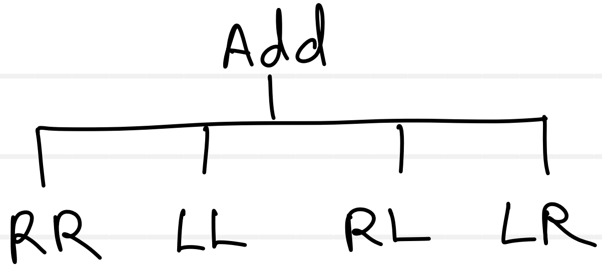
R → right of imbalance node

L → imbalance node

LR Imbalance

Keys : 30, 10, 20





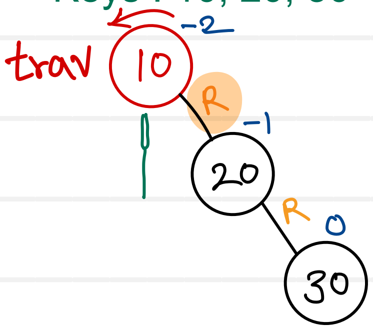
$bf = \{-1, 0, +1\}$

$bf < -1$

$bf > +1$

RR Imbalance

Keys : 10, 20, 30

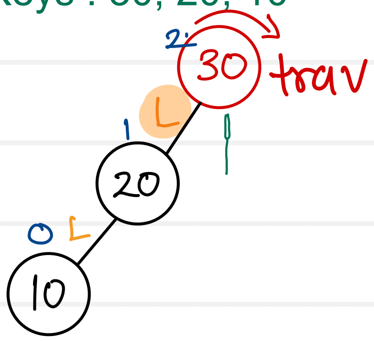


value > trav.right.data

(30 > 20)

LL Imbalance

Keys : 30, 20, 10

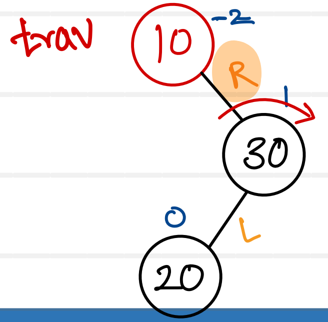


value < trav.left.data

(10 < 20)

RL Imbalance

Keys : 10, 30, 20

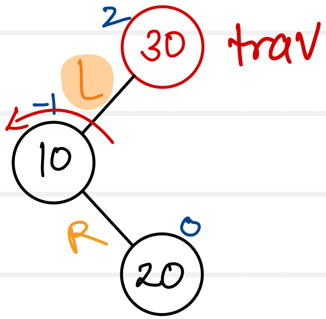


value < trav.right.data

(20 < 30)

LR Imbalance

Keys : 30, 10, 20



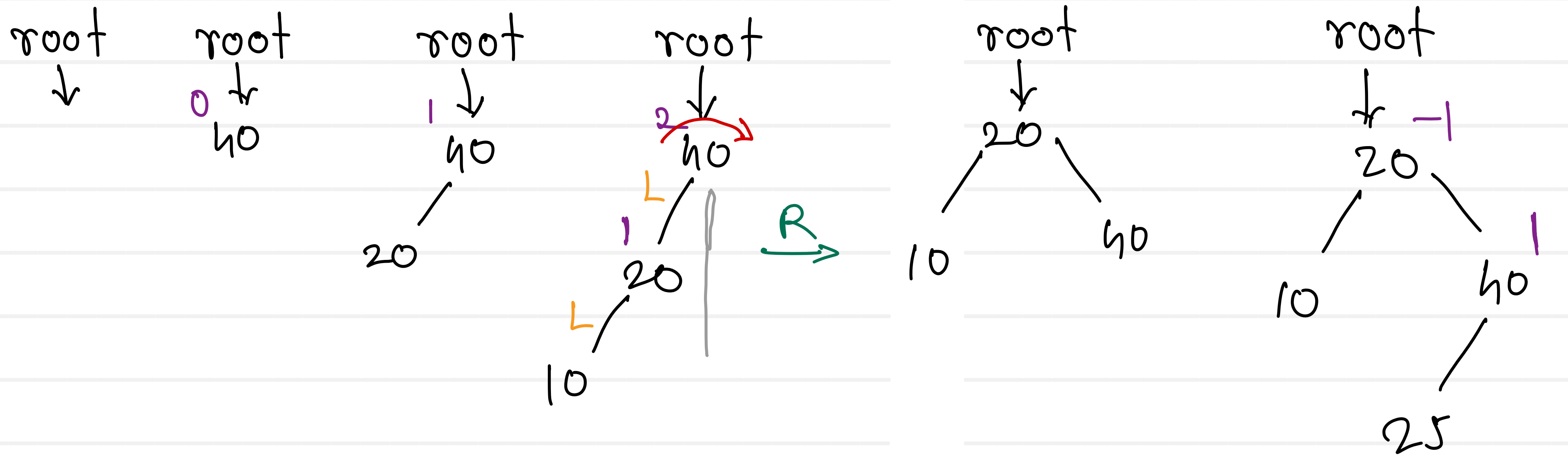
value > trav.left.data

(20 > 10)

AVL Tree

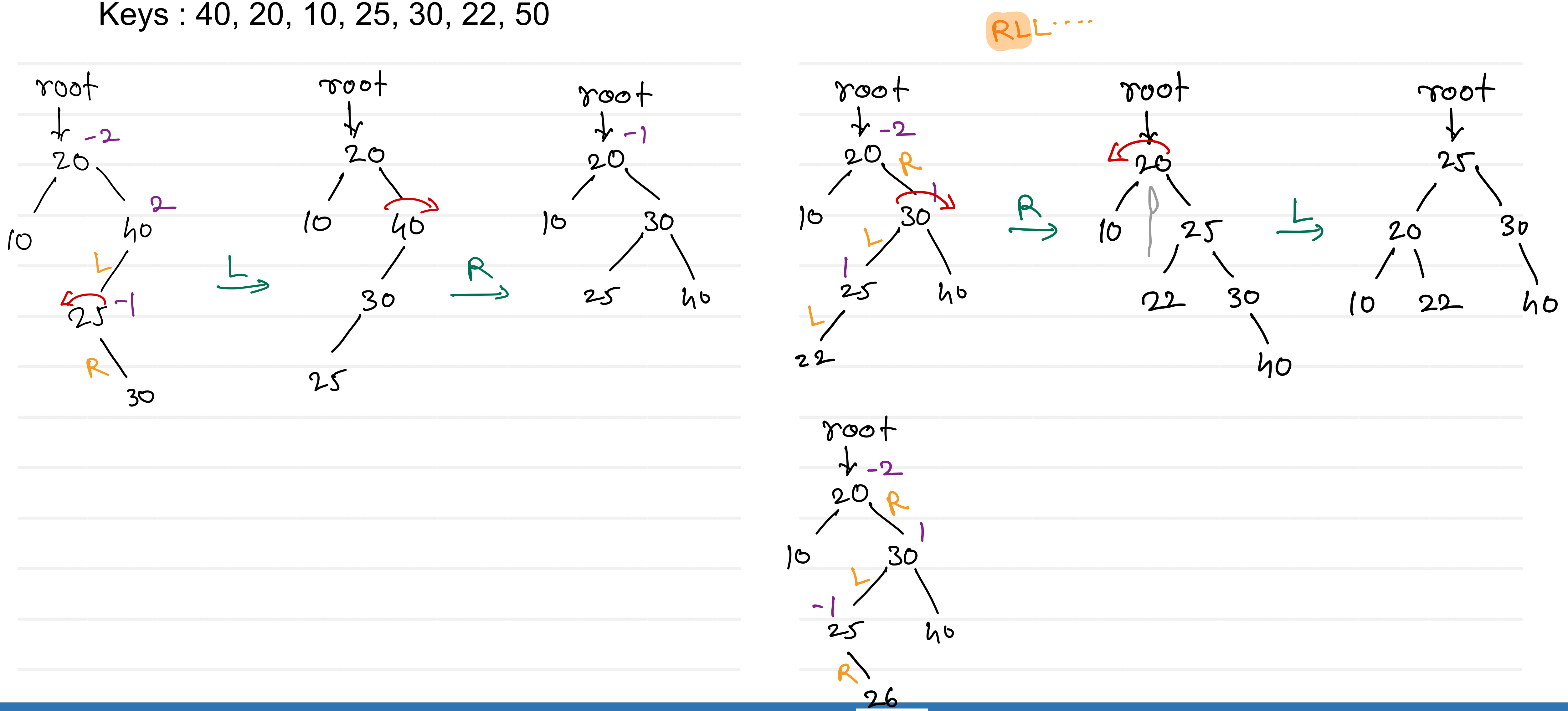
- self balancing binary search tree
- on every insertion and deletion of a node, tree is getting balanced by applying rotations on imbalance nodes
- The difference between heights of left and right sub trees can not be more than one for all nodes
- Balance factors of all the nodes are either -1 , 0 or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

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



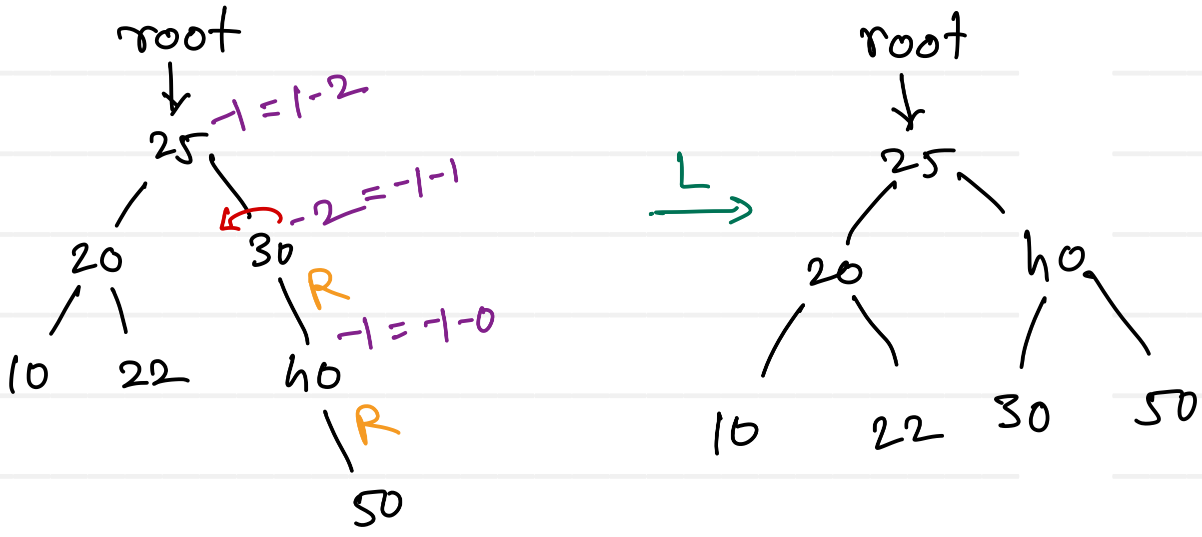
AVL Tree

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



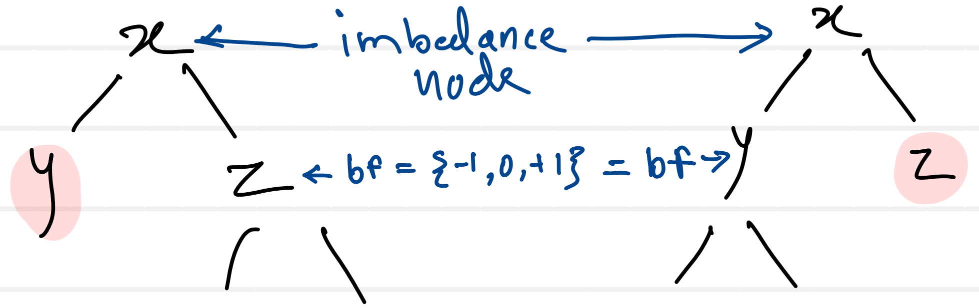
AVL Tree

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



AVL Tree

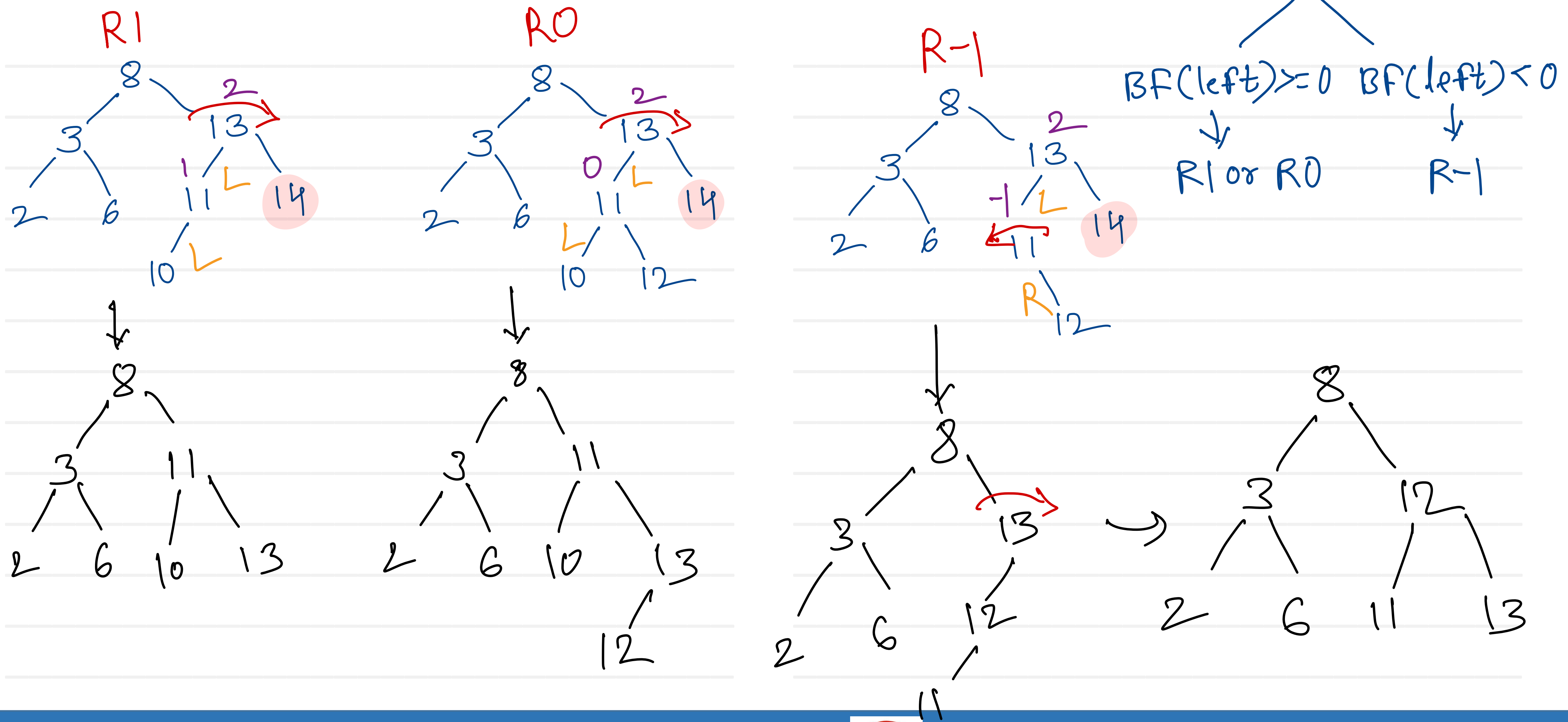
L deletion R deletion



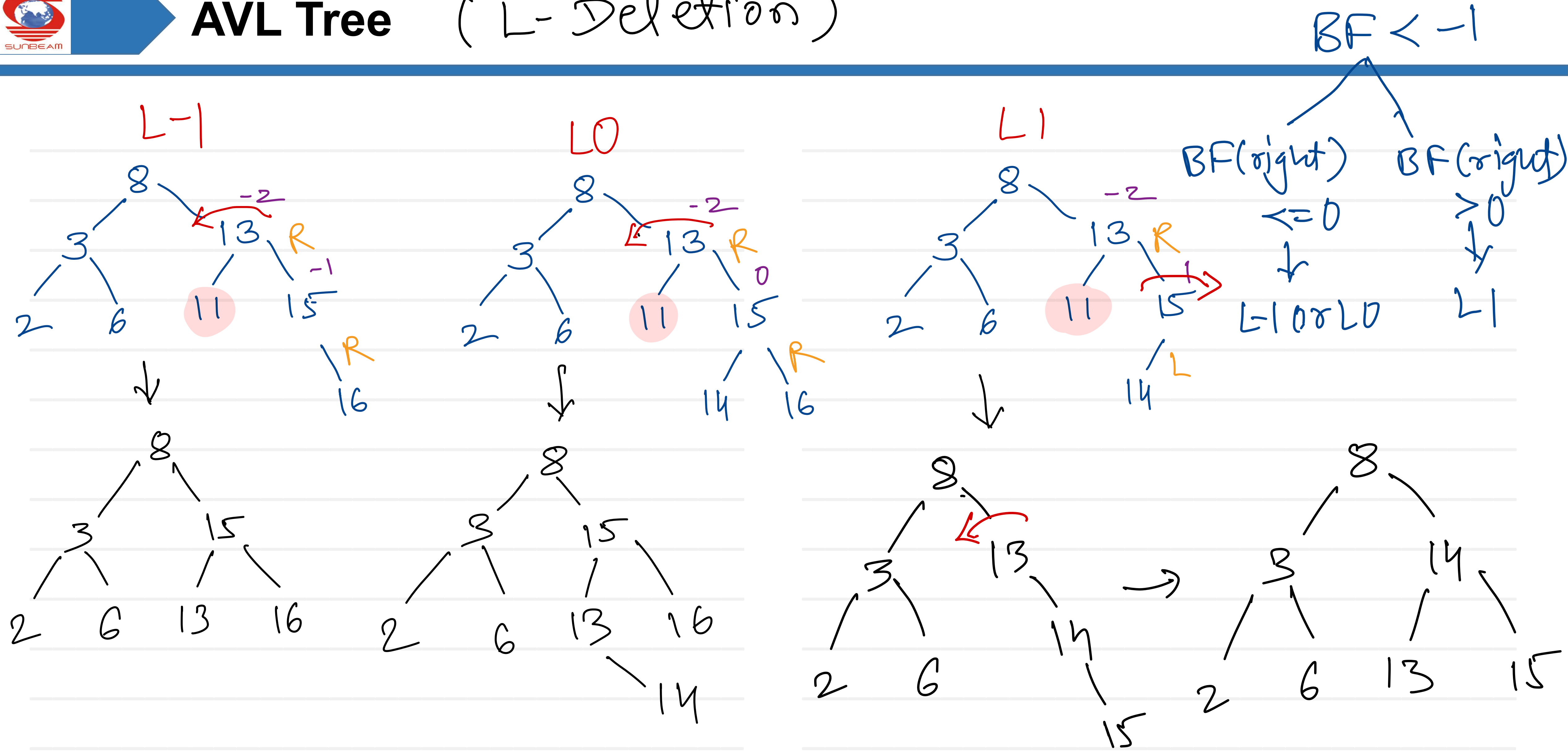
- Add
  - LL
  - RR
  - LR
  - RL

- Delete
  - L deletion
    - L-1 (RR)
    - L0 (RR)
    - L1 (RL)
  - R deletion
    - R-1 (LR)
    - R0 (LL)
    - R1 (LL)

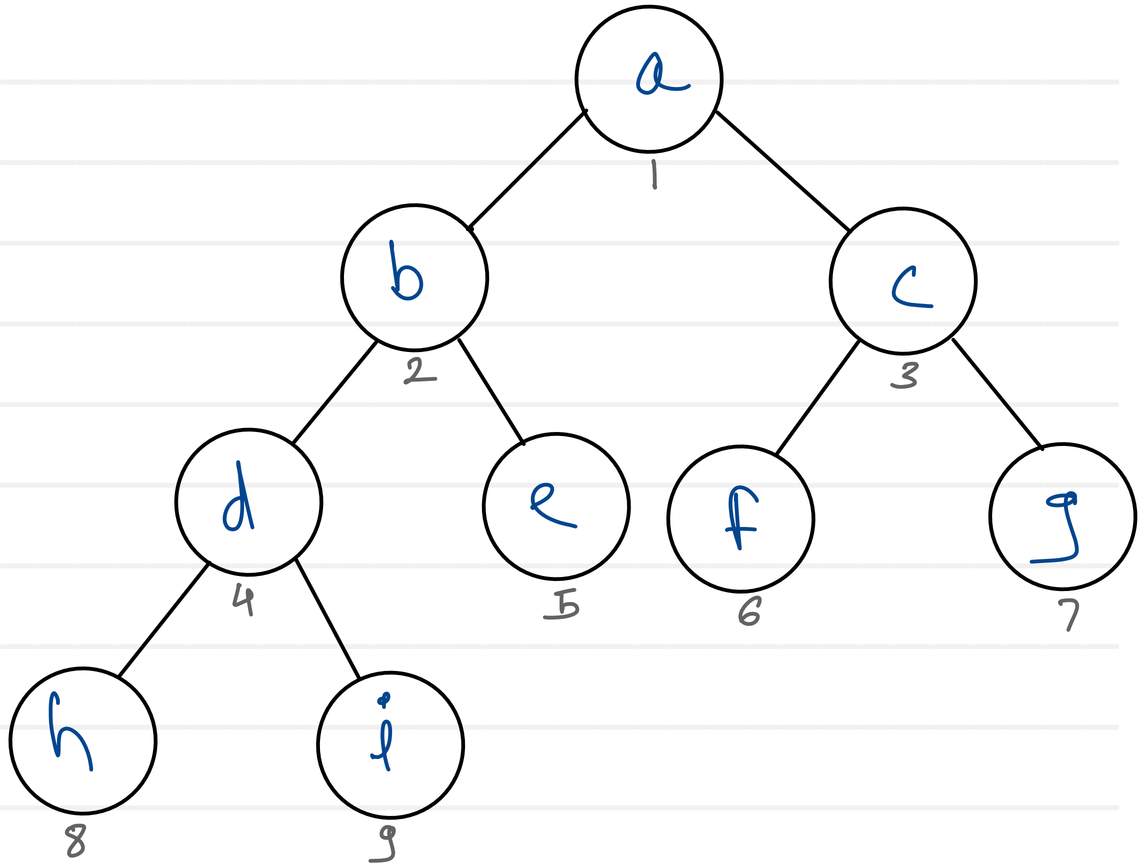
AVL Tree (R Deletion)



AVL Tree (L- Deletion)



Complete Binary Tree or Heap

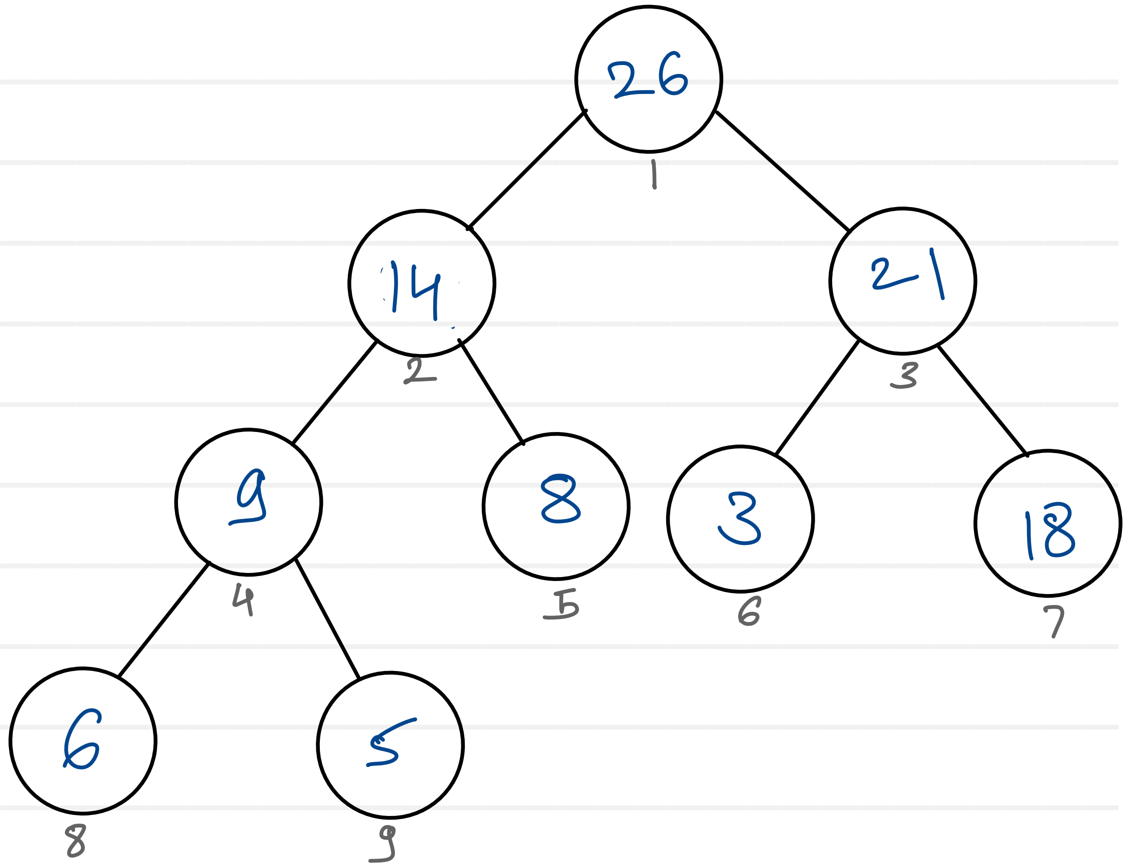


a	b	c	d	e	f	g	h	i
1	2	3	4	5	6	7	8	9

- Complete Binary Tree (height = h)
- All levels should be completely filled except last
- 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 Complete Binary Tree is called as heap

node – i^{th} index
parent node – $i/2$ index
left child – $i*2$ index
right child – $i*2+1$ index

Heap - Create heap (Add)



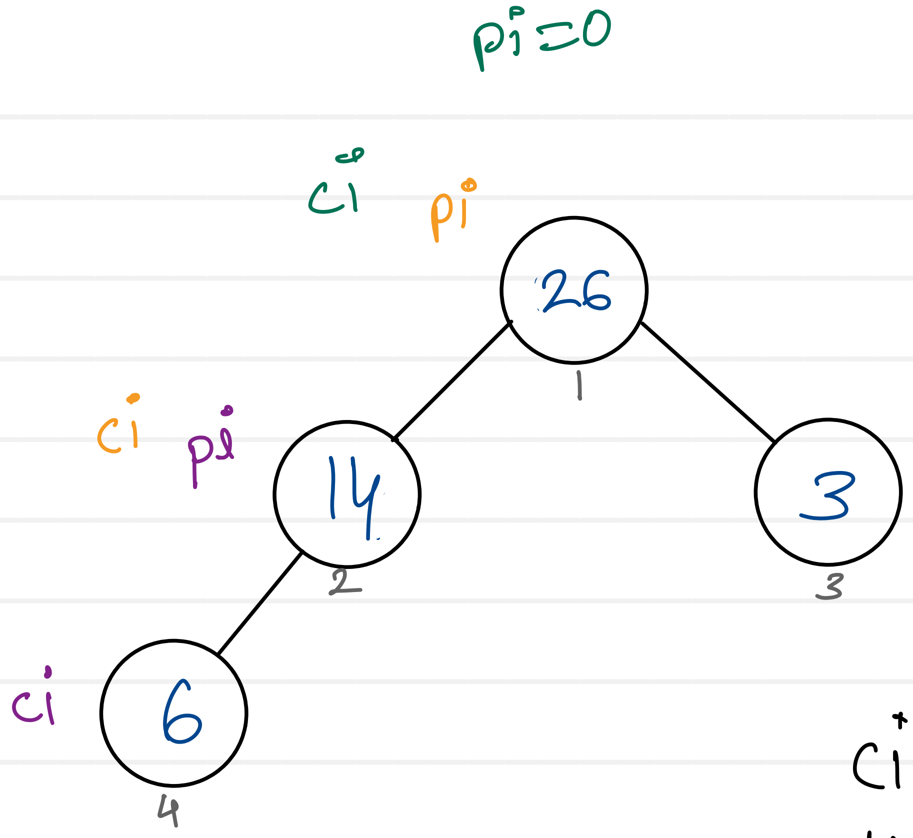
26	14	21	9	8	3	18	6	5
1	2	3	4	5	6	7	8	9

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

1. add new value at first empty index from left
2. adjust position of newly added value by comparing with its ancestors one by one to make it heap.

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

Heap - Add

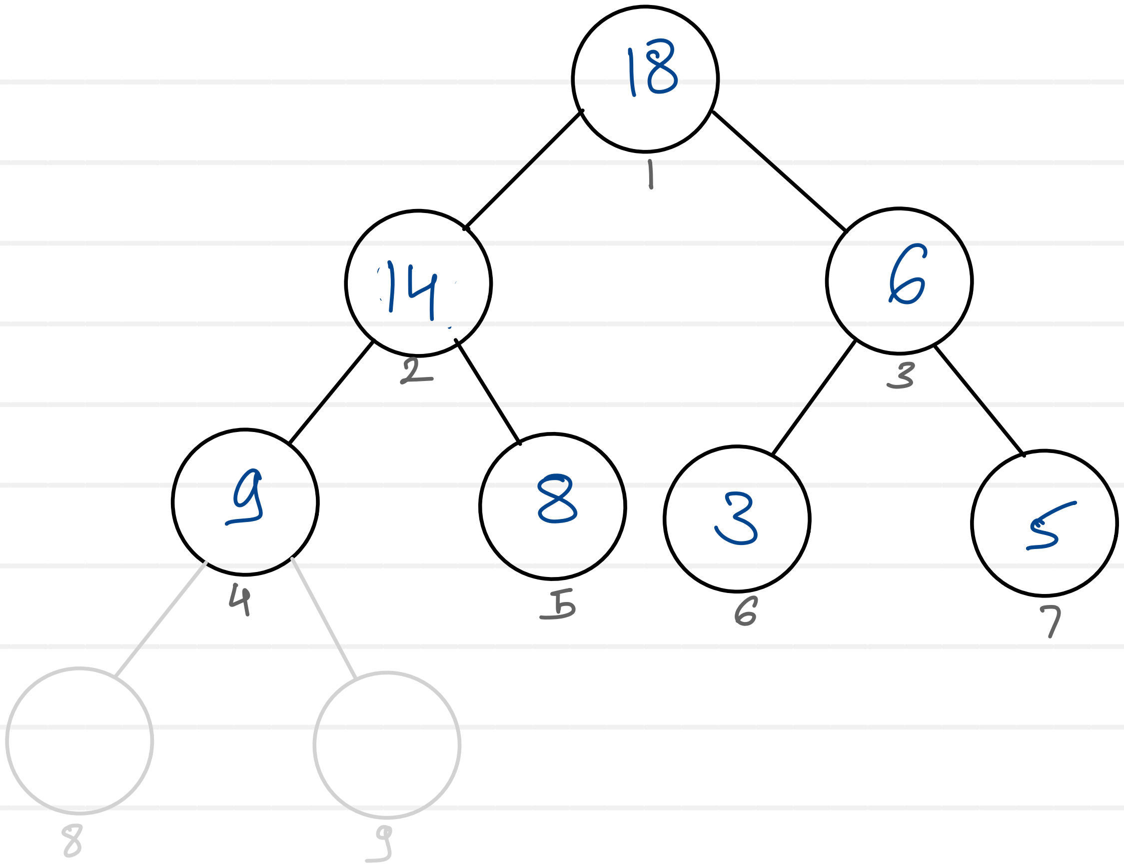



```
int arr[10];
int SIZE = 0;
```

ci	pi
4	2
2	1
1	0

```
void addHeap(int value){
    SIZE++;
    arr[SIZE]=value;
    int ci = SIZE;
    int pi = ci/2;
    while(pi >= 1){
        if(arr[pi] > arr[ci])
            break;
        int temp = arr[pi];
        arr[pi] = arr[ci];
        arr[ci] = temp;
        ci = pi;
        pi = ci/2;
    }
}
```

Heap - Delete heap (Delete)



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

Property : can delete only root node from heap

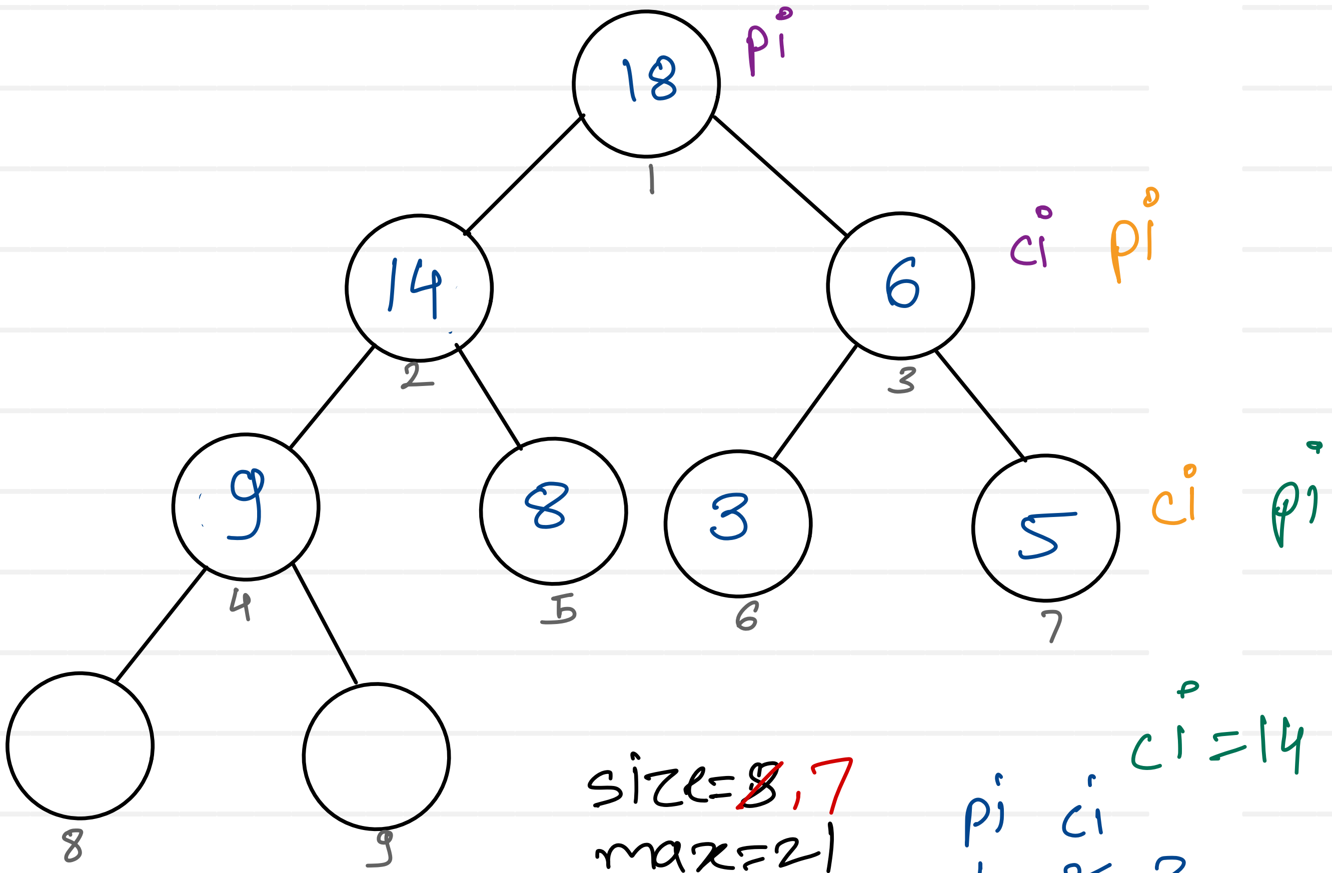
1. in max heap, always maximum element will be deleted from heap.
2. in min heap, always minimum element will be deleted from heap.

max = 26
max = 21

i. move last element of heap at root position.
ii. adjust position of new root by comparing it with all descendants one by one to make heap

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

Heap - Delete



size = ~~8~~, 7
max = 21

ci = 14

pi	ci
1	~~2~~, 3
3	~~6~~, 7

```
int deleteHeap( ) {
    int max = arr[1];
    arr[1] = arr[SIZE];
    SIZE--;
    int pi = 1;
    int ci = pi*2;
    while(ci <= SIZE) {
        if(arr[ci+1] > arr[ci])
            ci = ci+1;
        if(arr[pi] > arr[ci]
            break;
        int temp = arr[pi];
        arr[pi] = arr[ci];
        arr[ci] = temp;
        pi = ci;
        ci = pi*2;
    }
    return max;
}
```

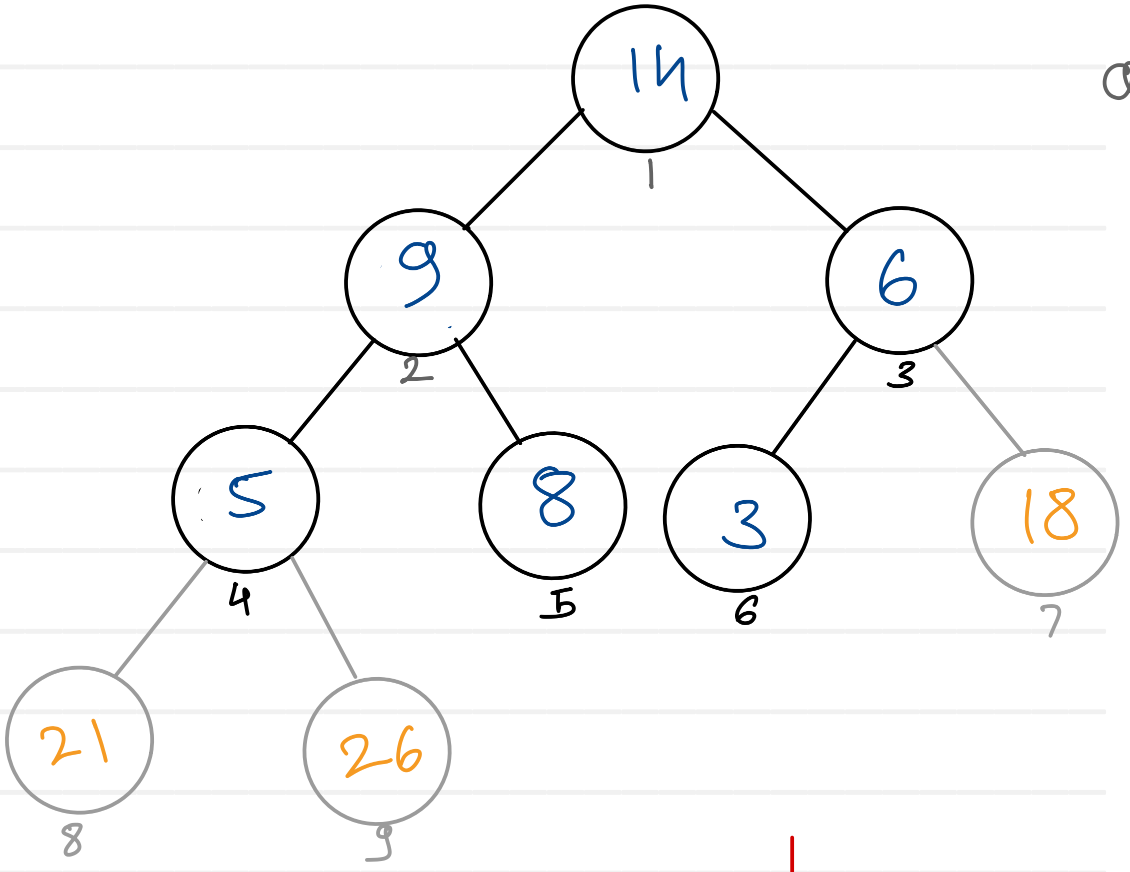
Priority Queue

- Always high priority element is deleted from queue
- value (priority) is assigned to each element of queue
- priority queue can be implemented using array or linked list.
- to search high priority data (element) need to traverse array or linked list
- Time complexity = $O(n)$

- priority queue can also be implemented using heap. because, maximum / minimum value is kept at root position in max heap & min heap respectively.
- push, pop & peek will be performed efficiently

max value → high priority → max heap
min value → high priority → min heap

Heap sort



arr

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

1. create heap from given array
2. delete all elements from heap and place them from right side

create heap – $n * \log n$
delete heap – $n * \log n$

total – $2n \log n$

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

$\boxed{S(n) = O(1)}$

arr

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

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com