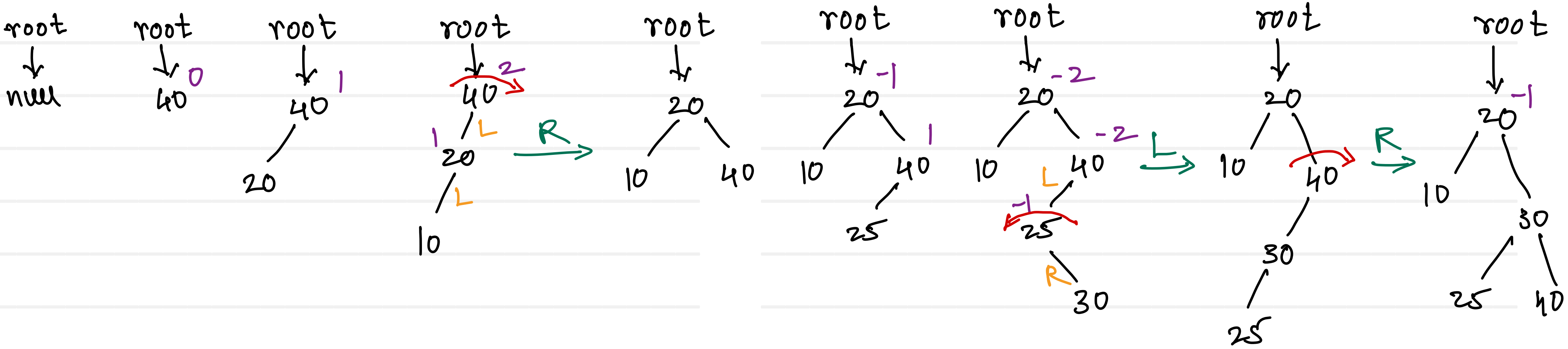
AVL Tree

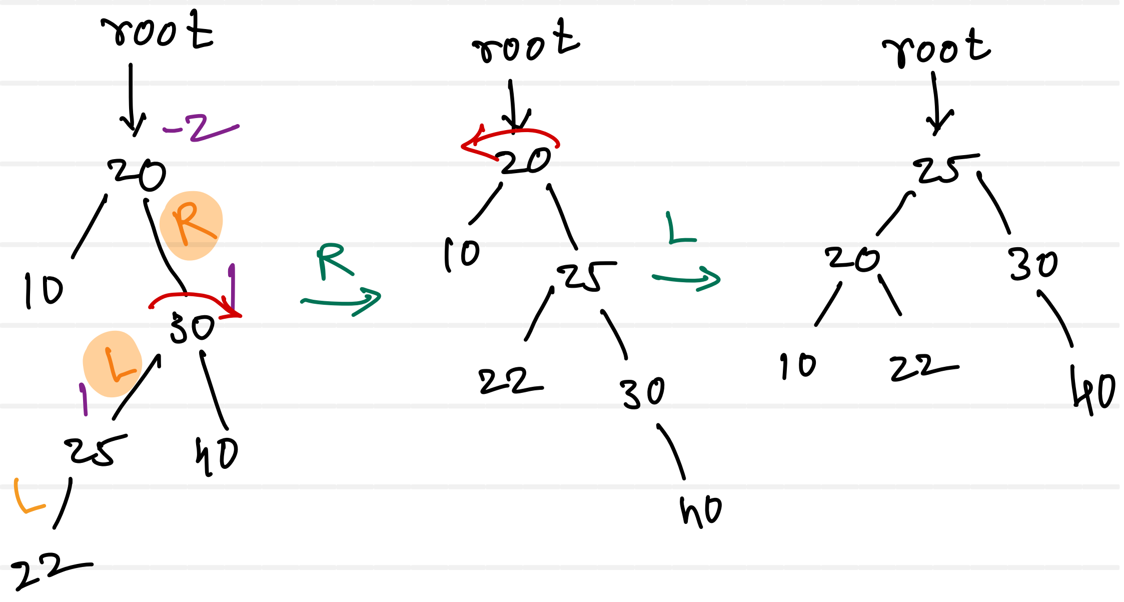
- self balancing binary search tree
- on every insertion and deletion of a node, tree is getting balanced by applying rotations on imbalance nodes
- The difference between heights of left and right sub trees can not be more than one for all nodes
- Balance factors of all the nodes are either -1 , 0 or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

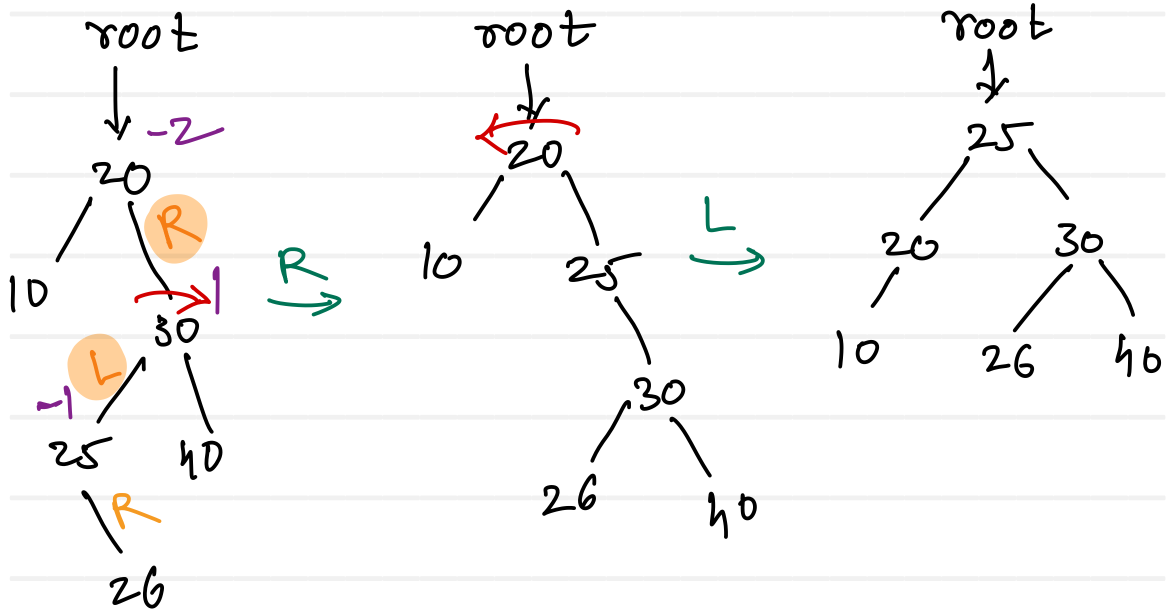
Keys : 40, 20, 10, 25, 30, 22, 50



AVL Tree

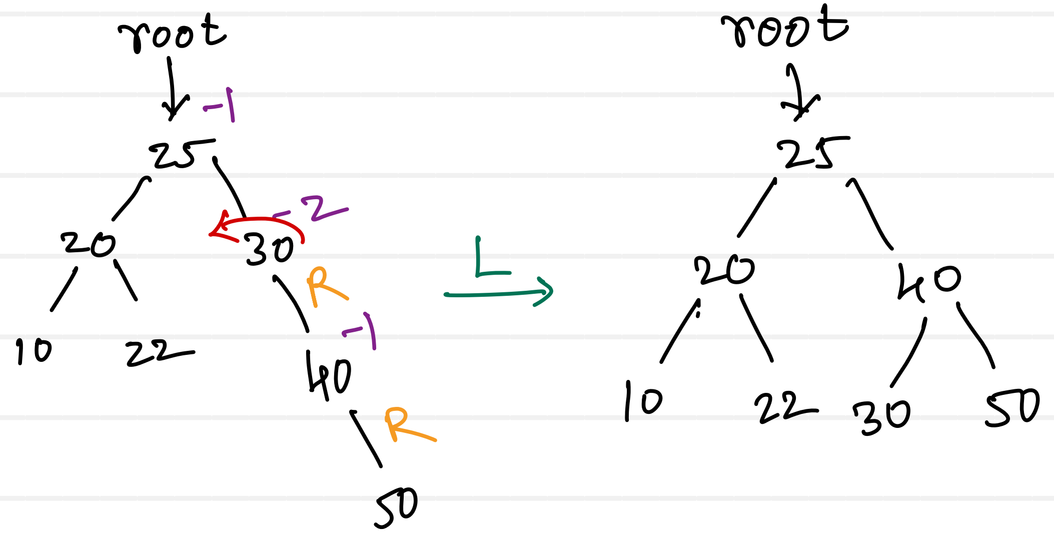
Keys : 40, 20, 10, 25, 30, 22, 50

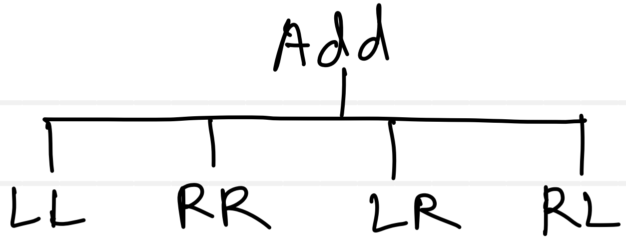


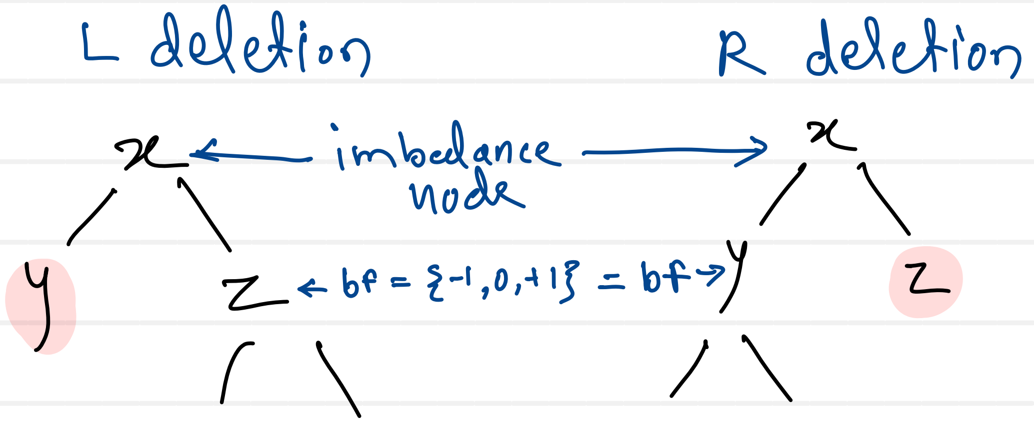


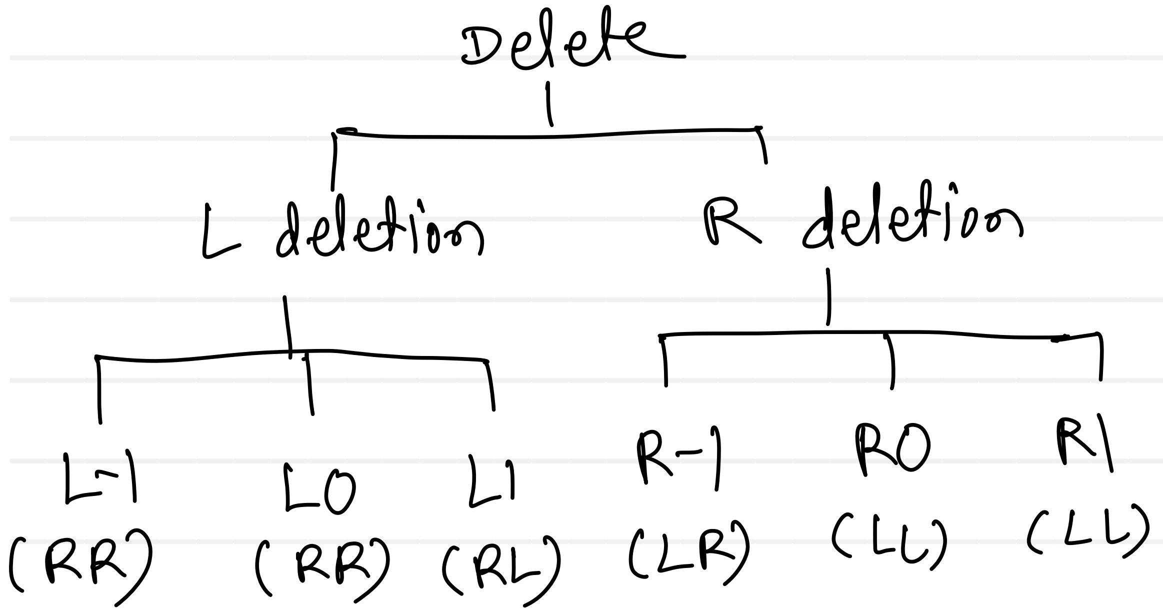
AVL Tree

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

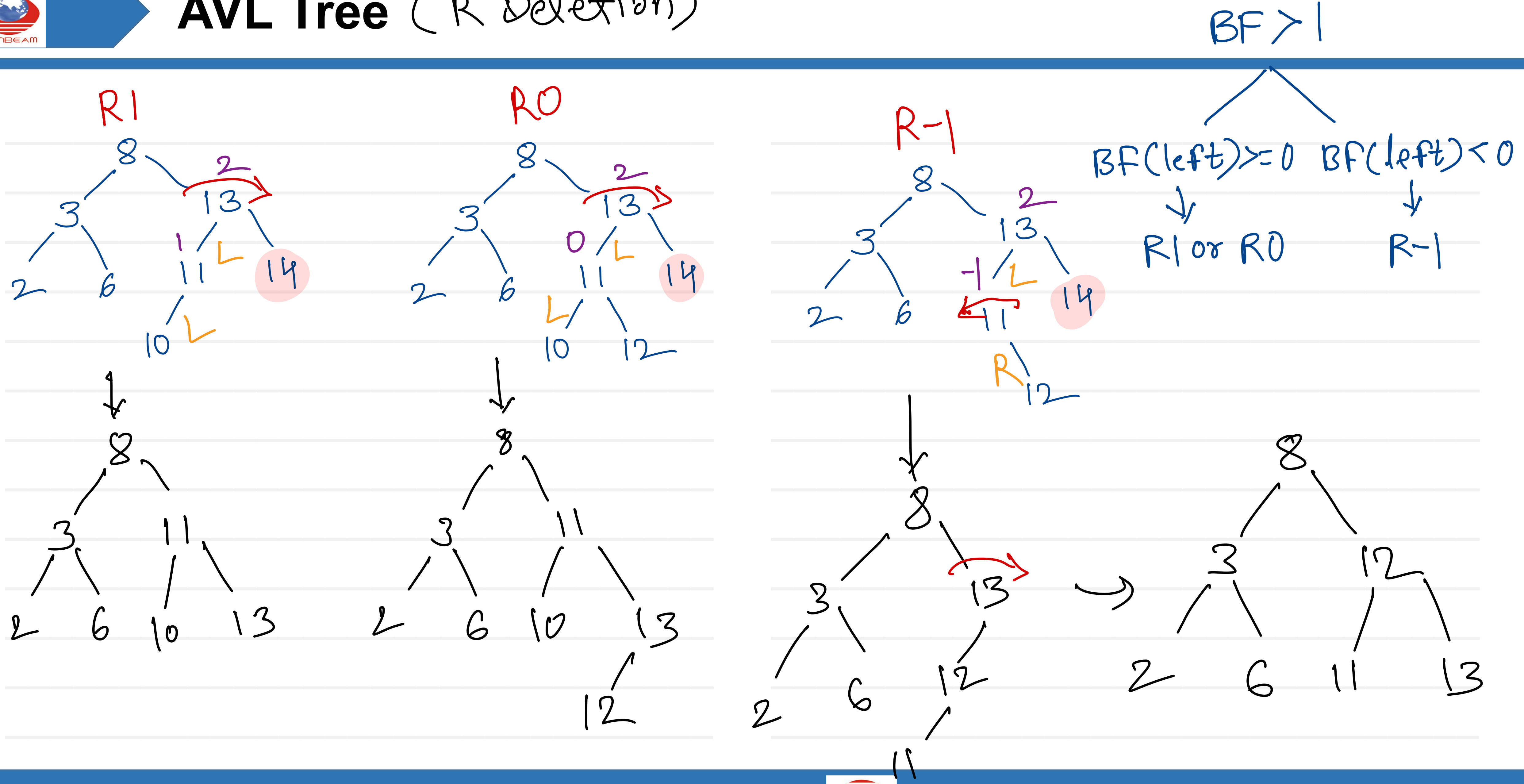




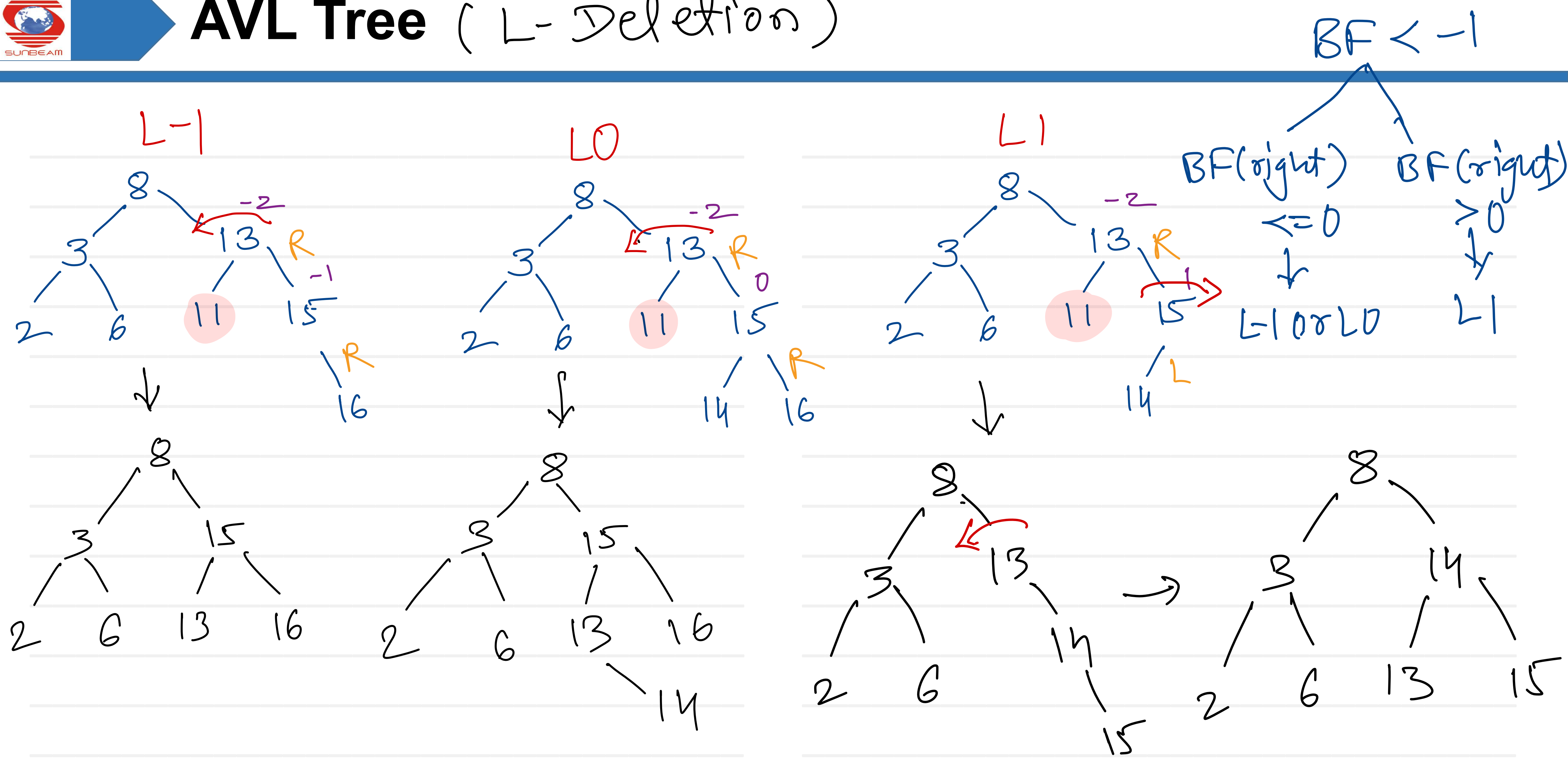




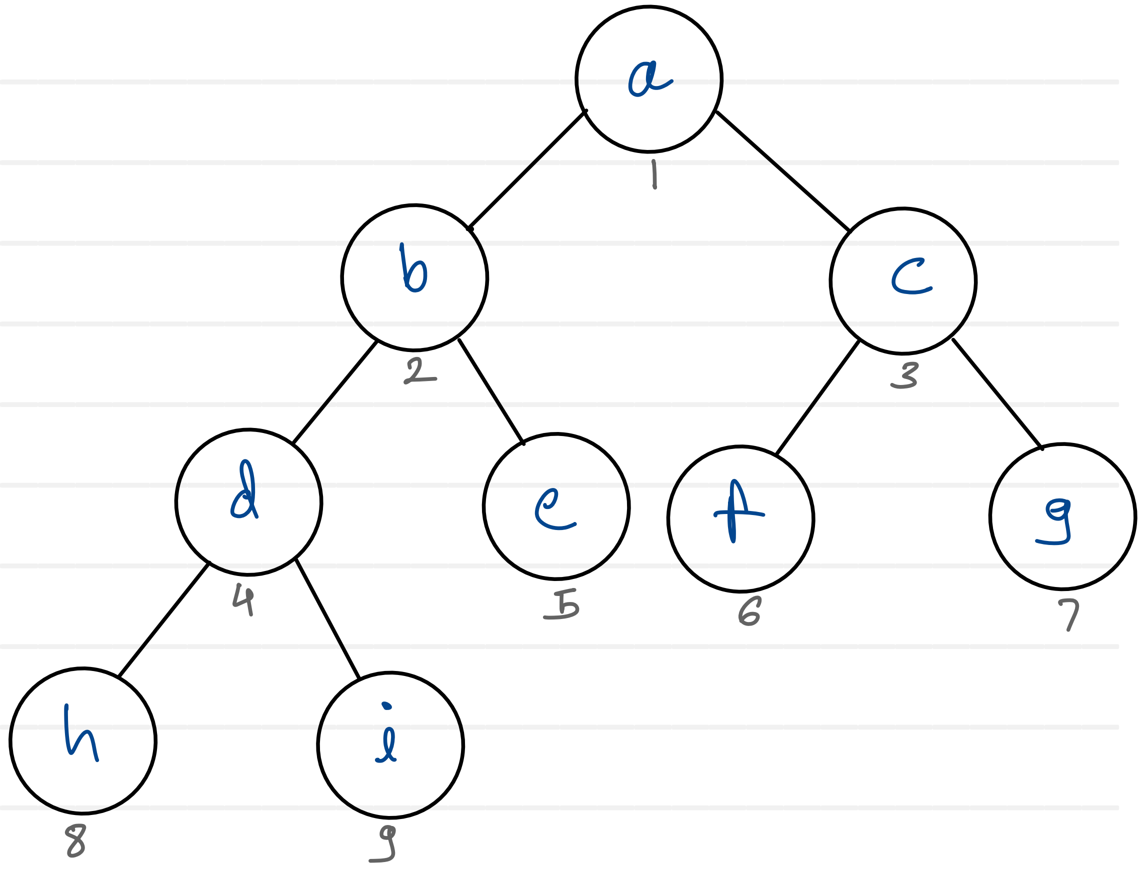
AVL Tree (R Deletion)



AVL Tree (L- Deletion)



Almost Complete Binary Tree or Heap



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

- 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

– Array indices are used to maintain relationship of parent & child

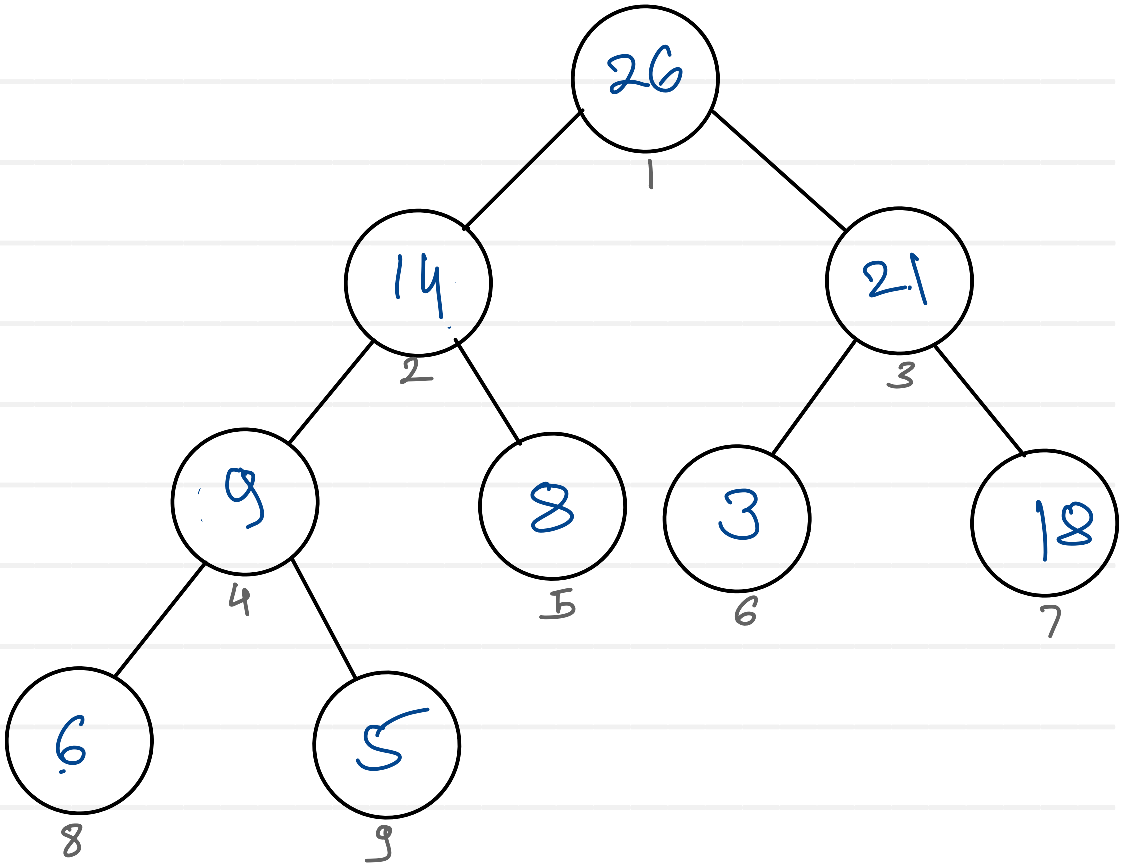
Node → i^{th} index

Parent → $i/2$ index

Left child → $i*2$ index

Right child → $(i*2)+1$ index

Heap - Create heap (Add)



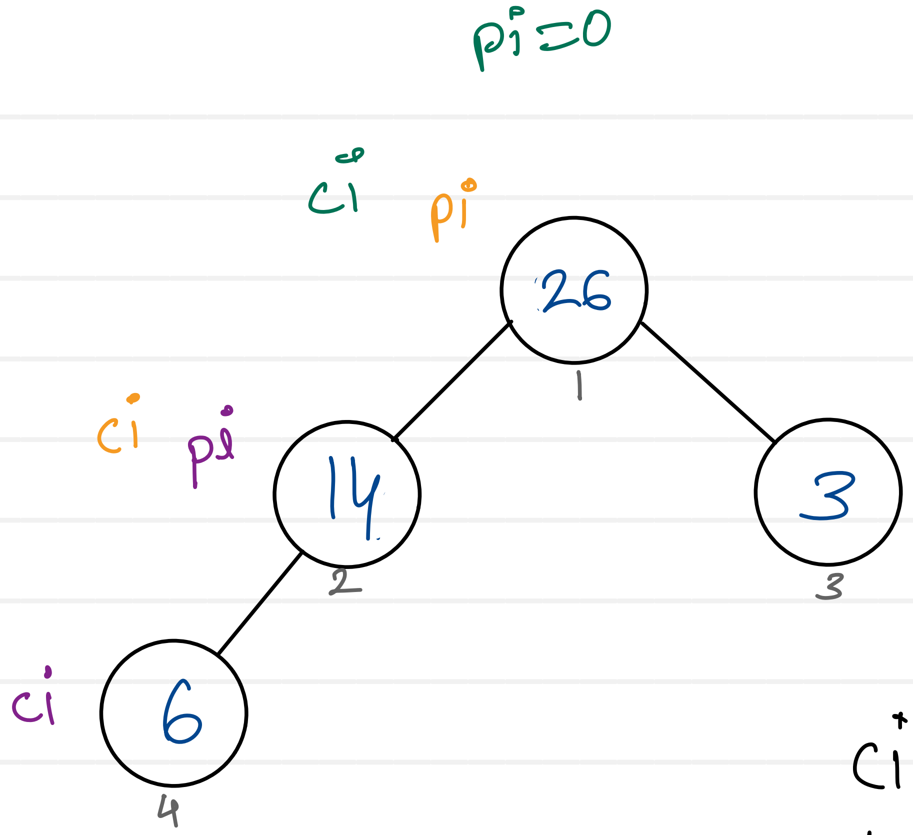
26	14	21	9	8	3	18	6	5
1	2	3	4	5	6	7	8	9

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

i. add new value at first empty index of array from left side
ii. adjust position of newly added value by comparing it with all its ancestors.

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

Heap - Add

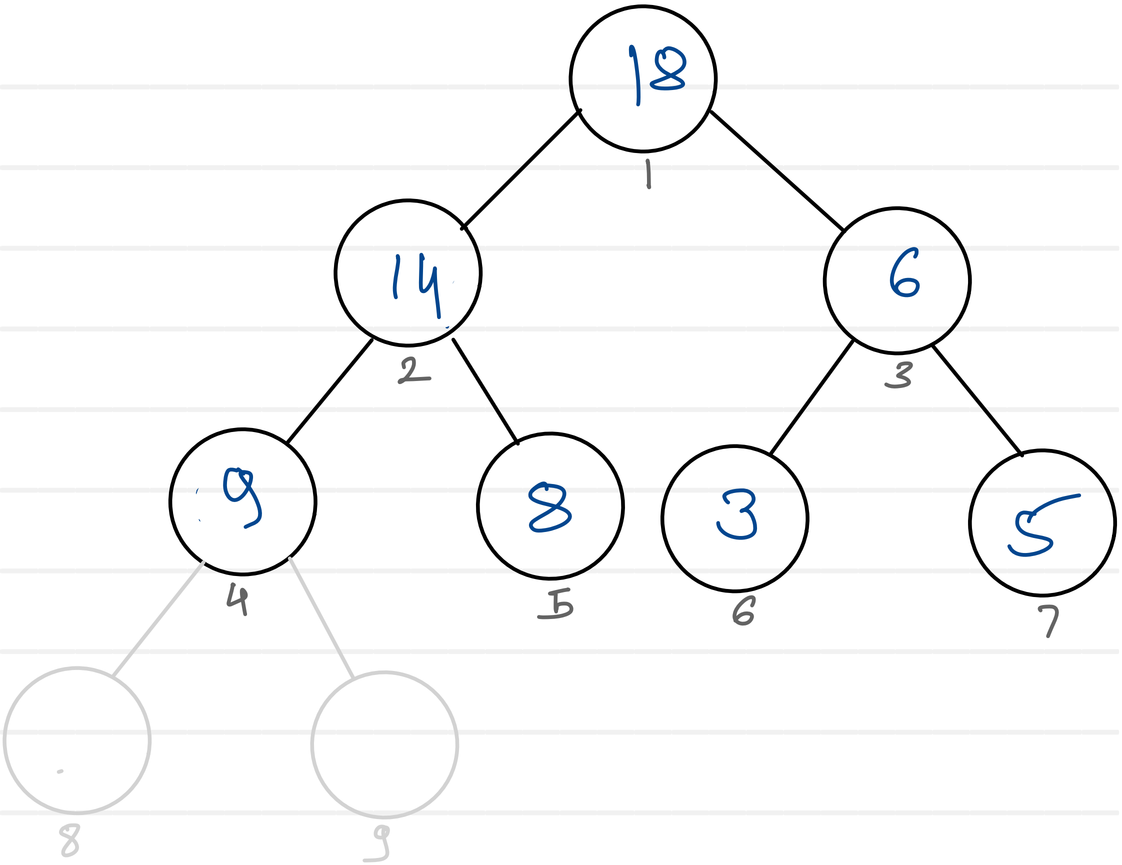



```
int arr[10] ;
int SIZE = 0;
```

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 of heap : can delete only root node

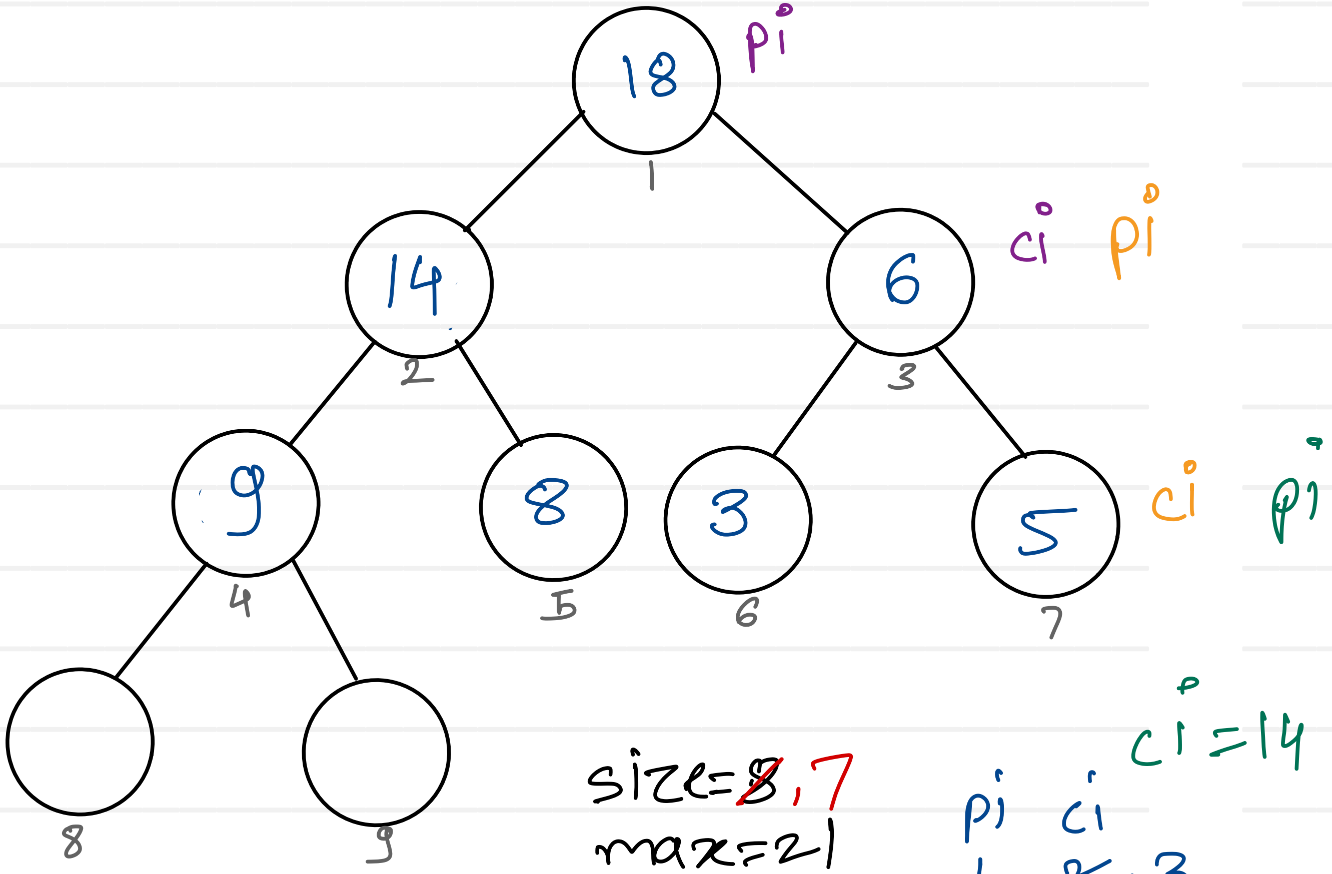
- From max heap, always maximum value will be deleted
- From min heap, always minimum value will be deleted.

Max = 26
Max = 21

i. replace last value of heap at root's place to delete root node (max value)
ii. adjust position of new root by comparing it with all its descendents upto leaf position

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

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;
```

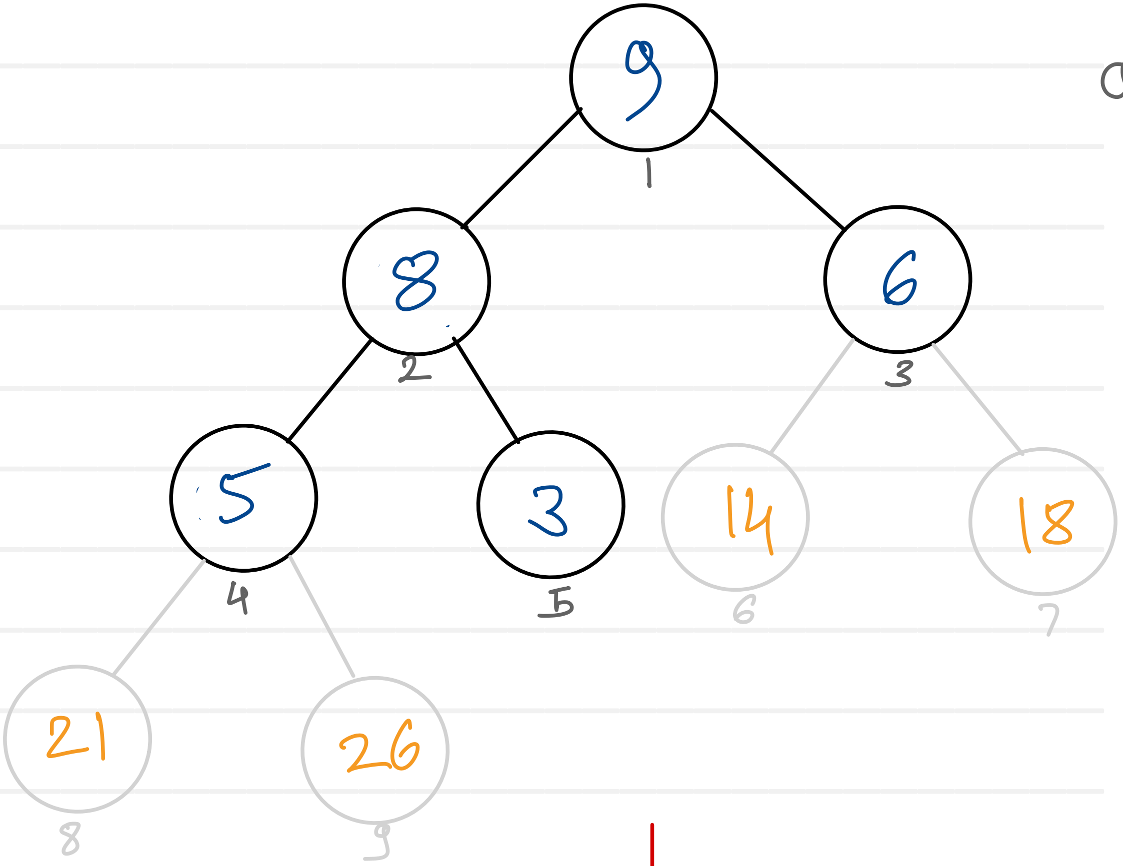
Priority Queue

- 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 queque 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

i. add all elements of array into heap

ii. delete all elements from heap & keep them on deleted locations of heap.

time to add n elements = $n \log n$

time to delete n elements = $n \log n$

$2n \log n$

Time $\propto 2n\log n$

$\boxed{T(n) = O(n\log n)}$ Best / Worst / Avg

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

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com