Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

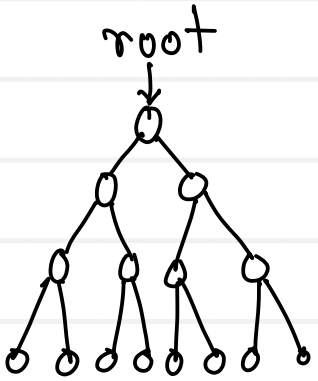
BST - Time complexity of operations

n - no. of nodes
h - height of tree

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

Capacity : max number of nodes for given height.

h	n
-1	0
0	1
1	3
2	7
3	15



Add	:	O(h)	O(log n)
delete	:	O(h) or	O(log n)
search	:	O(h)	O(log n)
Traverse	:	O(n)	

$$n \approx 2^h$$

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

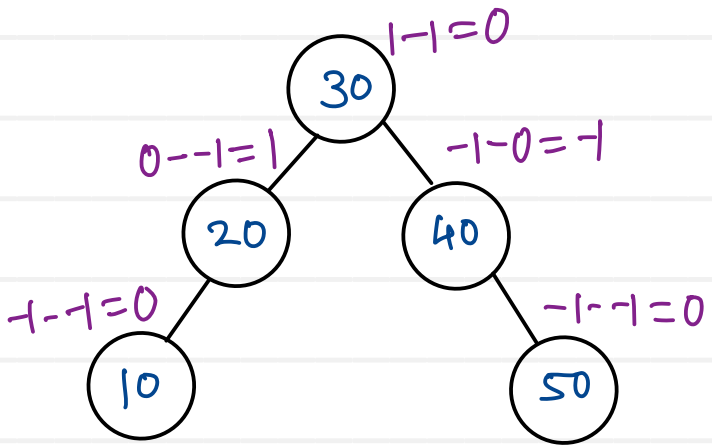
$$h = \frac{\log n}{\log 2}$$

Time $\propto$ height

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

Skewed Binary Search Tree

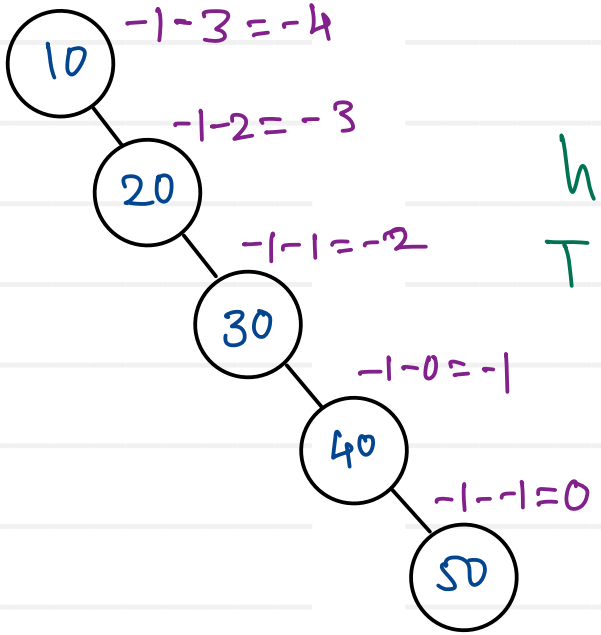
Keys : 30, 40, 20, 50, 10



$h = \log n$

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

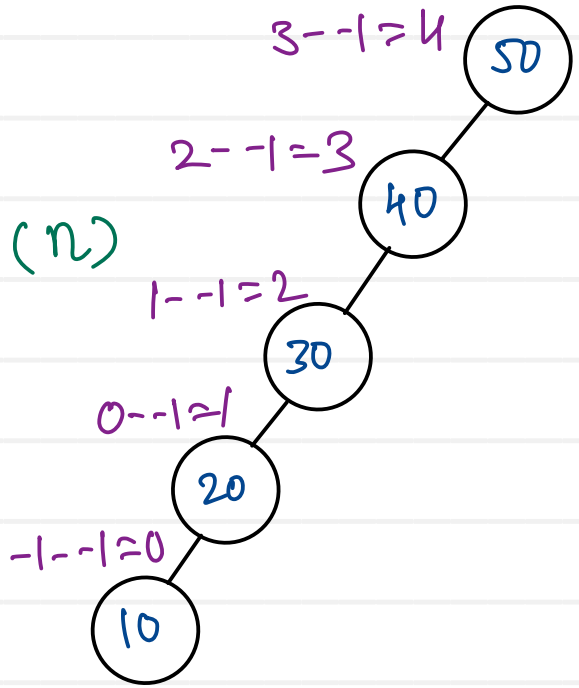
Keys : 10, 20, 30, 40, 50



$h = n$

$T(n) = O(n)$

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 search tree
  - Left skewed binary search tree
  - Right skewed binary search 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

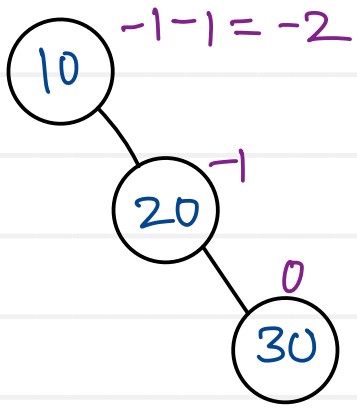
AVL tree
Red Black tree
splay tree

- 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

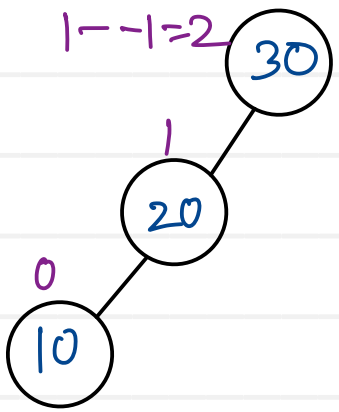
Balance factor = Height (left sub tree) - Height (right sub tree)

- 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 (having balance factors other than -1, 0 or +1)

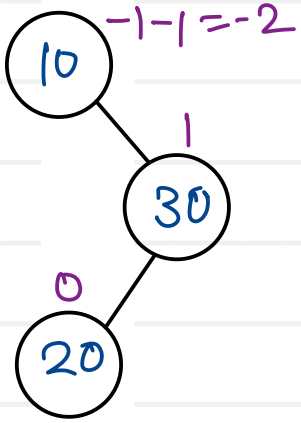
Keys : 10, 20, 30



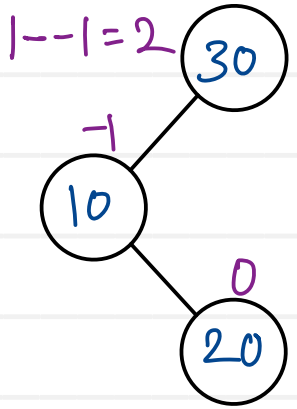
Keys : 30, 20, 10



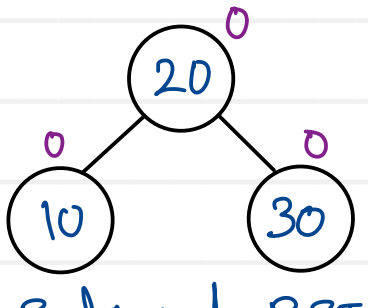
Keys : 10, 30, 20



Keys : 30, 10, 20

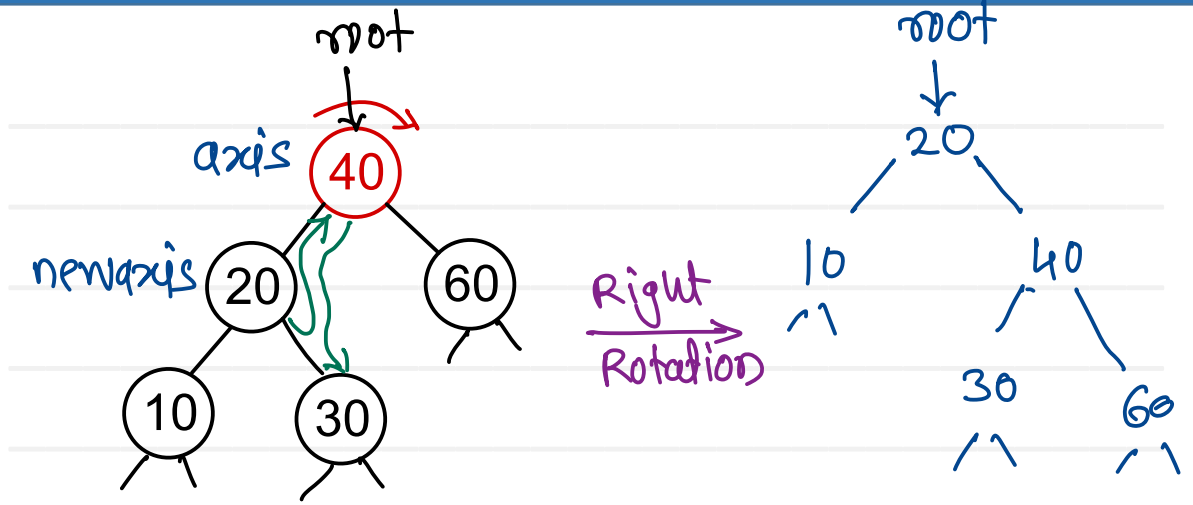


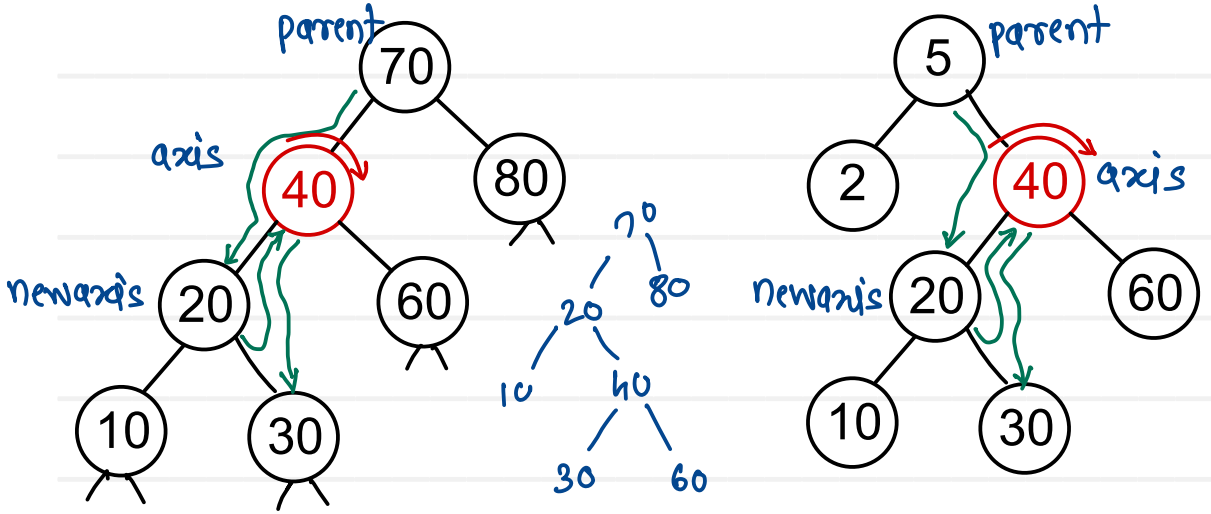
Keys : 20, 10, 30
Keys : 20, 30, 10



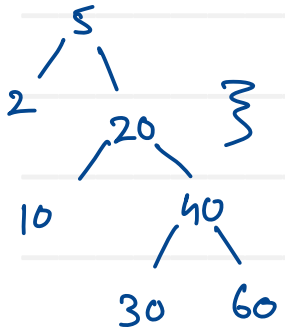
Balanced BST

Right Rotation

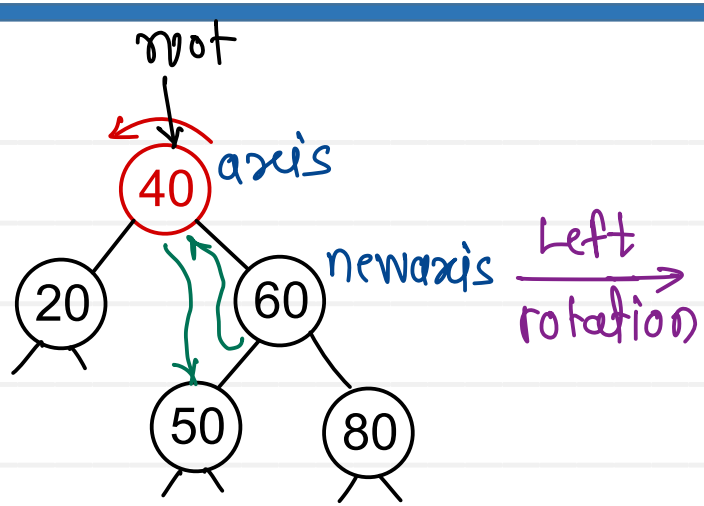




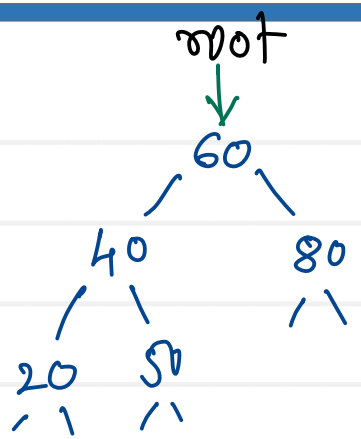

```
rightRotation( axis, parent) {
    newaxis = axis.left;
    axis.left = newaxis.right;
    newaxis.right = axis;
    if(axis == root)
        root = newaxis;
    else if( axis == parent.left)
        parent.left = newaxis;
    else if( axis == parent.right)
        parent.right = newaxis;
```



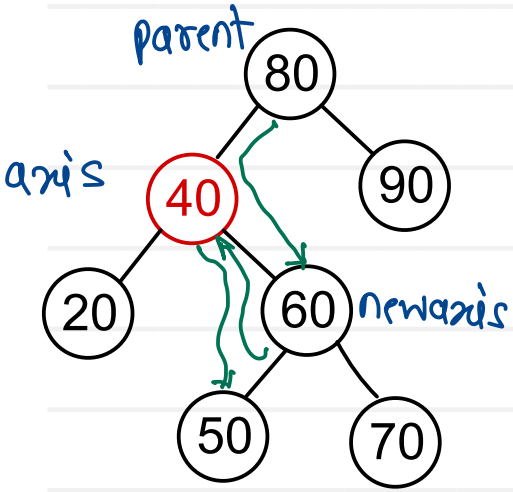
Left Rotation

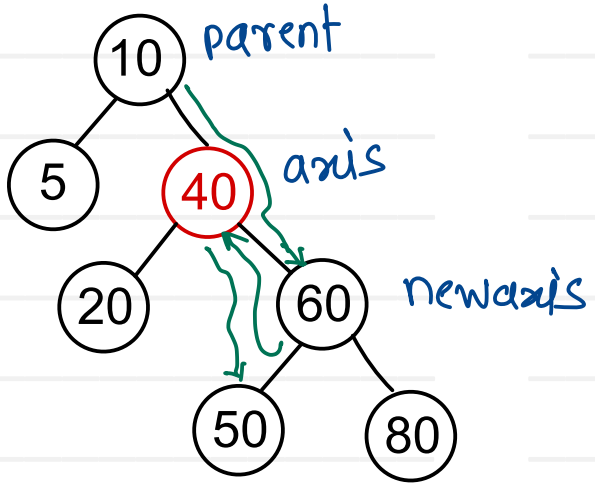


Left rotation




```
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;
}
```

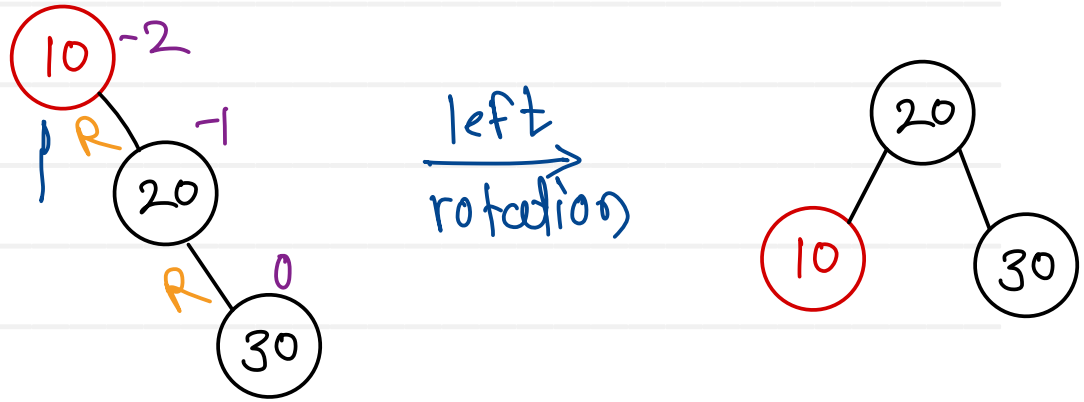




Rotation cases (Single Rotations)

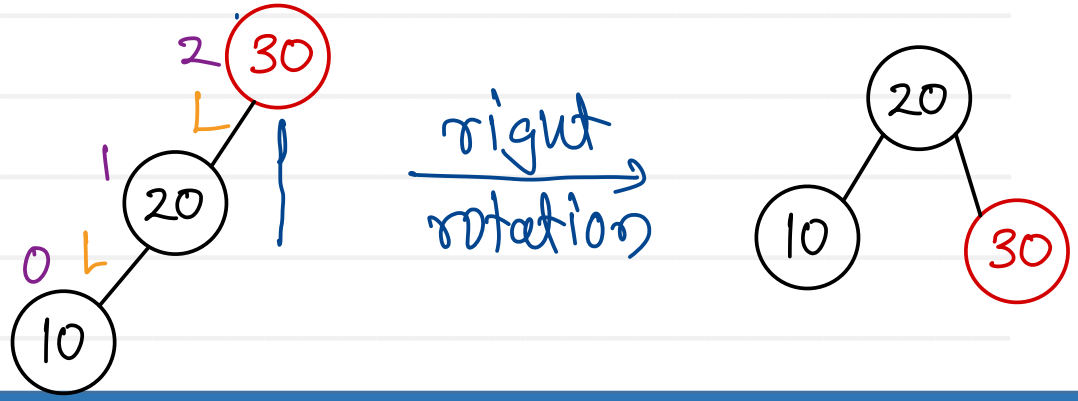
RR Imbalance → apply left rotation on imbalance node

Keys : 10, 20, 30



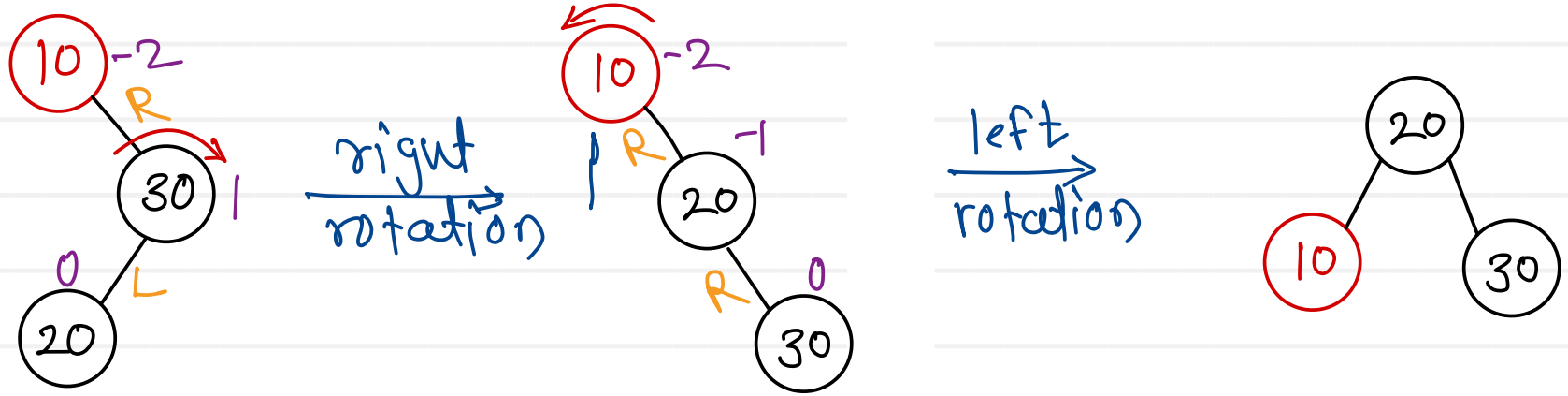
LL Imbalance : apply right rotation on imbalance node

Keys : 30, 20, 10

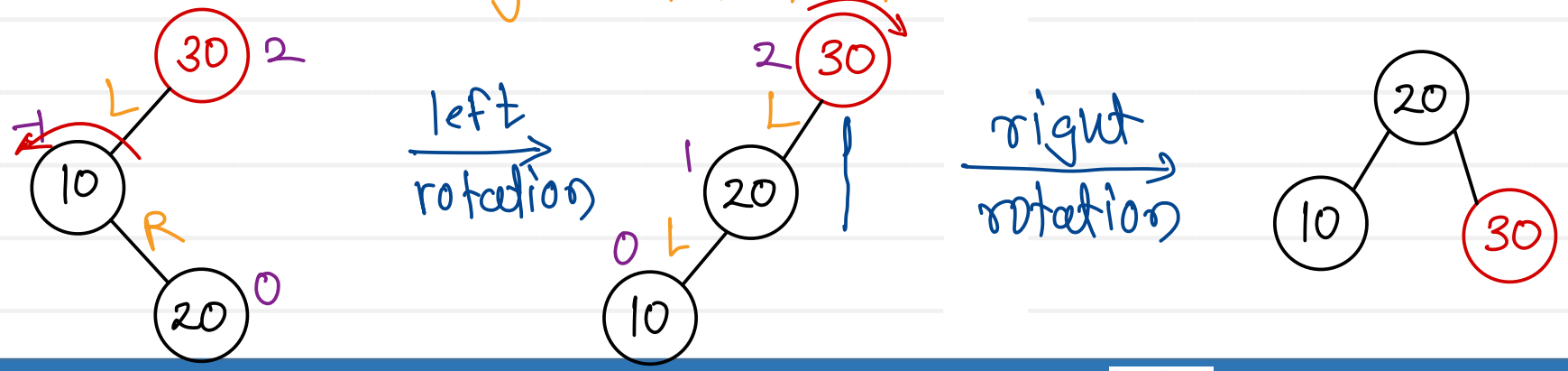


Rotation cases (Double Rotations)

RL Imbalance : - right rotation on right of imbalance node
Keys : 10, 30, 20 - left rotation on imbalance node

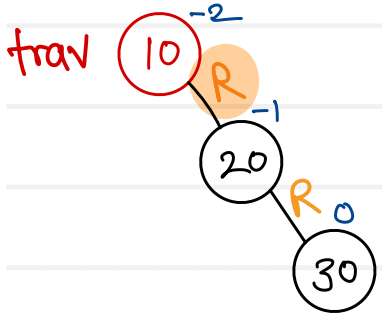


LR Imbalance :- left rotation on left of imbalance node
Keys : 30, 10, 20 - right rotation on imbalance node



RR Imbalance

Keys : 10, 20, 30

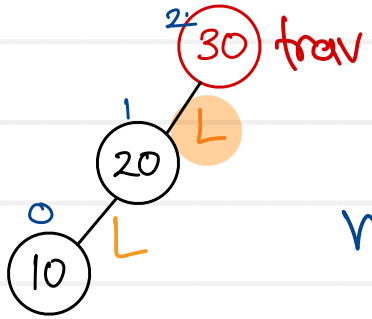


bf < -1

value > trav.right.data

LL Imbalance

Keys : 30, 20, 10

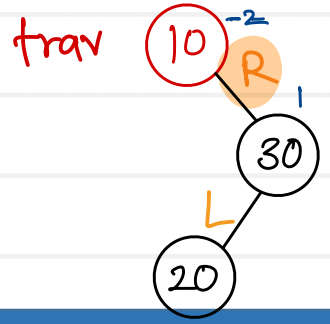


bf > 1

value < trav.left.data

RL Imbalance

Keys : 10, 30, 20

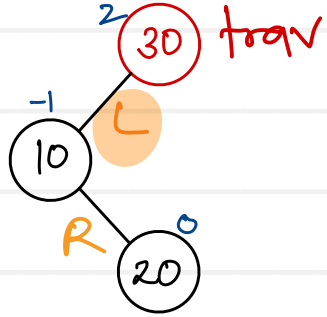


bf < -1

value < trav.right.data

LR Imbalance

Keys : 30, 10, 20



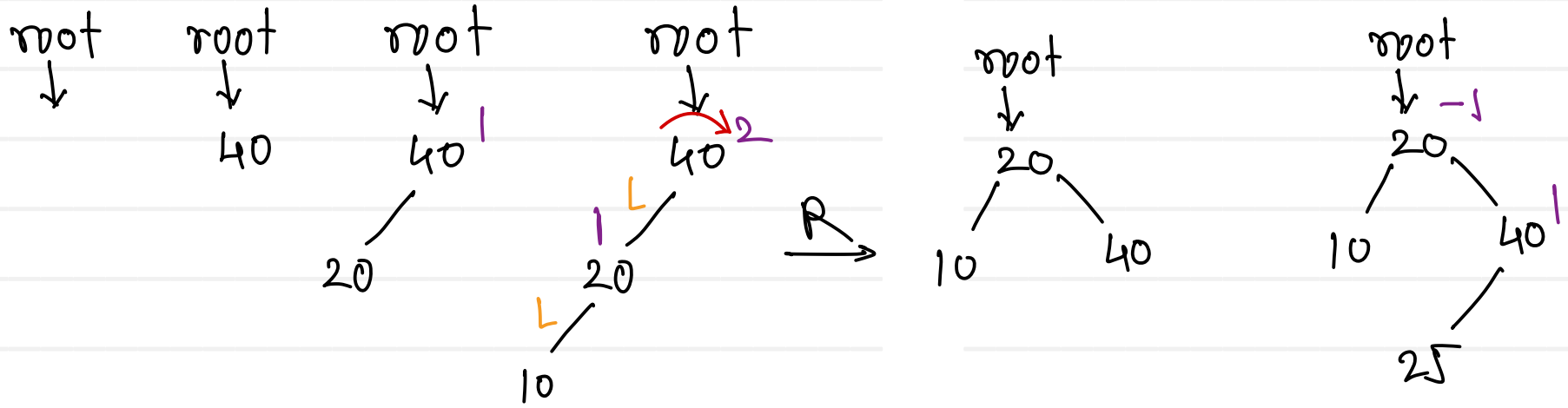
bf > 1

value > trav.left.data

AVL Tree

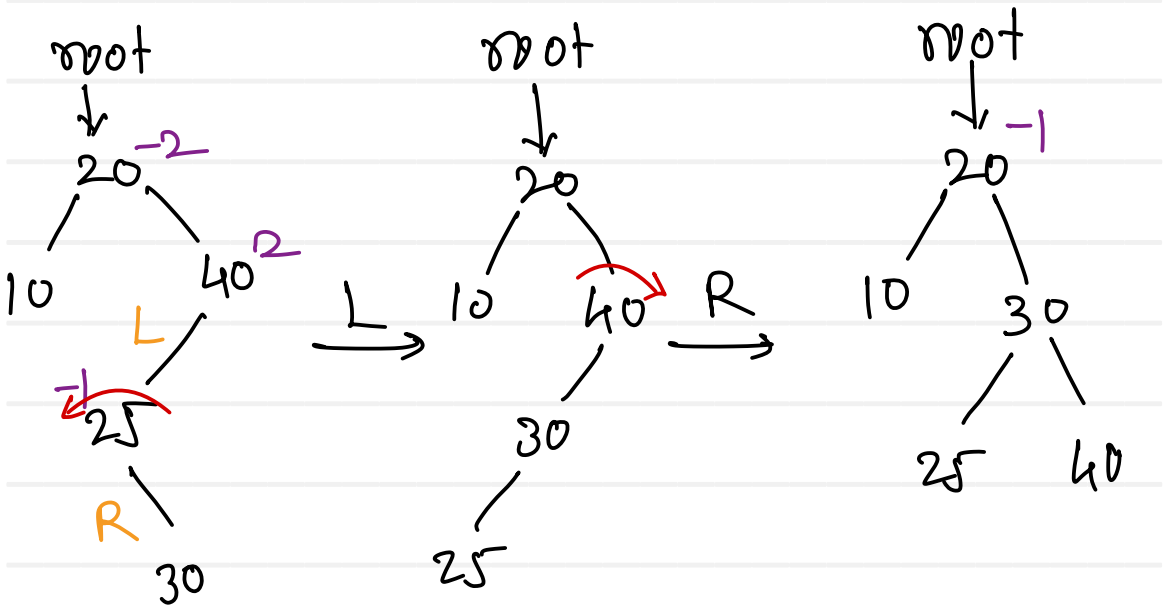
- self balancing binary search tree
- on every insertion and deletion of a node, tree is getting balanced by applying rotations on imbalance nodes
- The difference between heights of left and right sub trees can not be more than one for all nodes
- Balance factors of all the nodes are either -1 , 0 or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

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

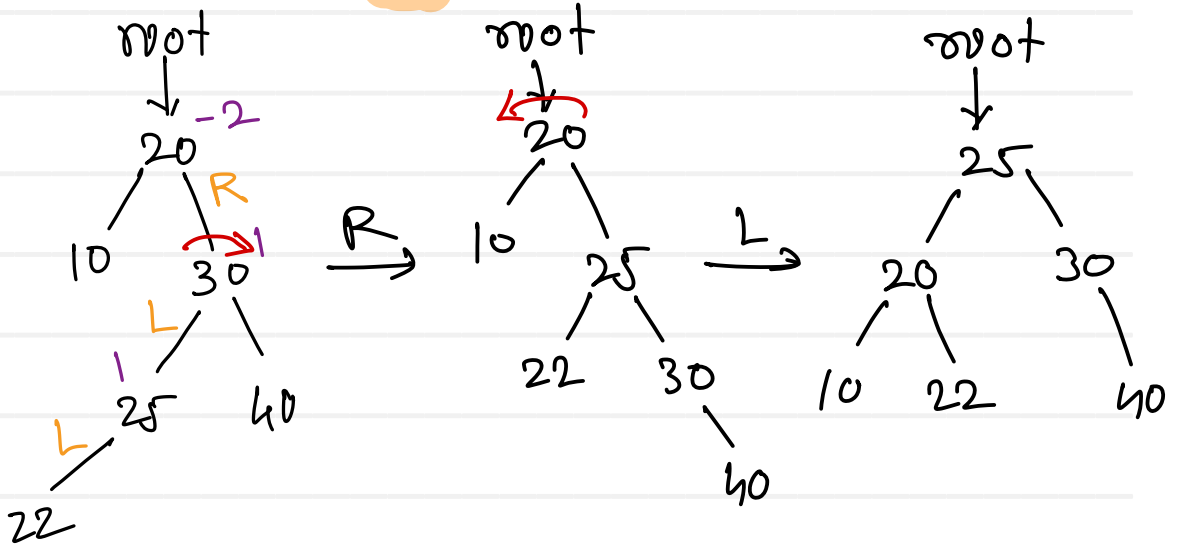


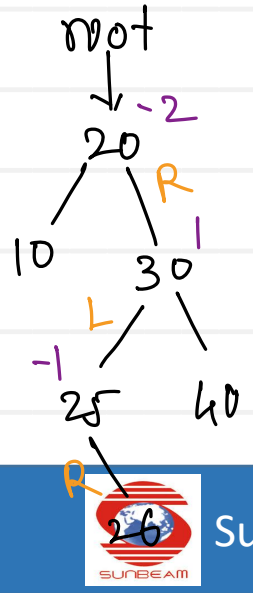
AVL Tree

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



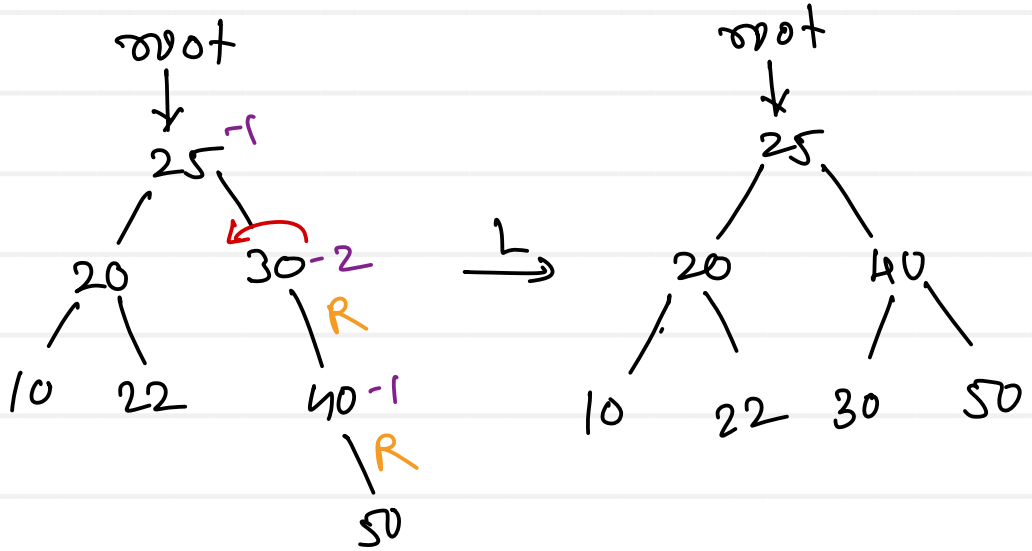
RLL....
RLR...

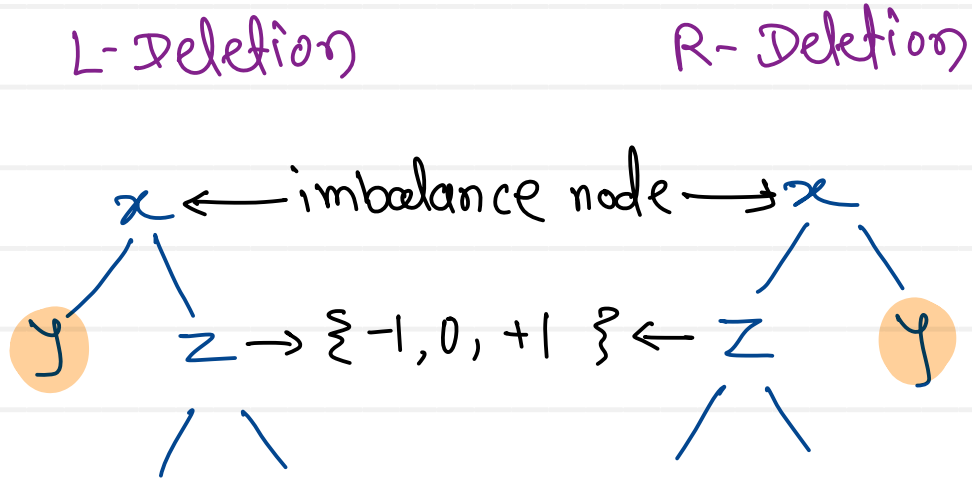




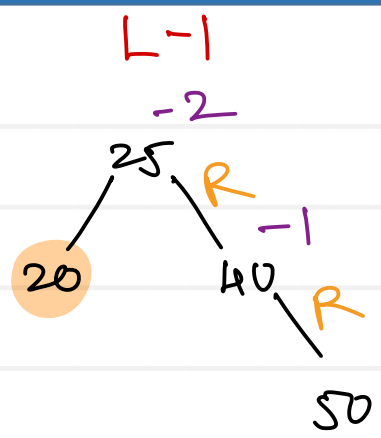
AVL Tree

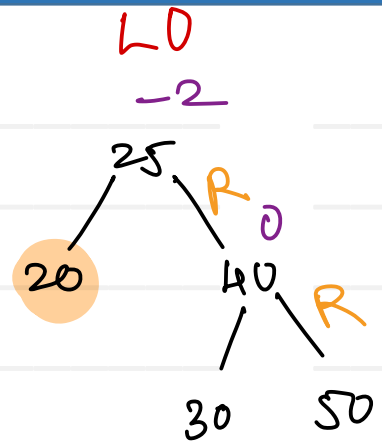
Keys : 40, 20, 10, 25, 30, 22, 50

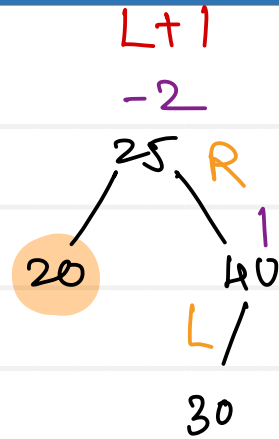




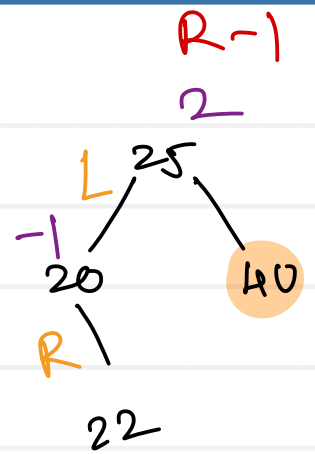
AVL Tree L - Deletion

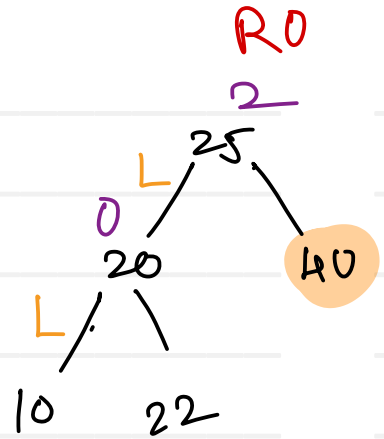


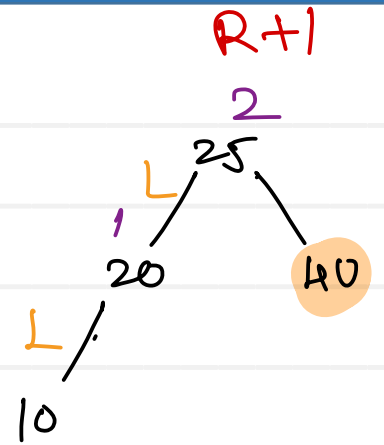




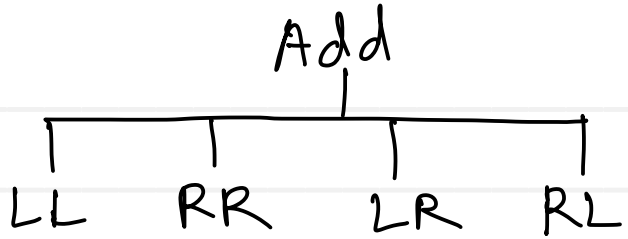
R-Deletion

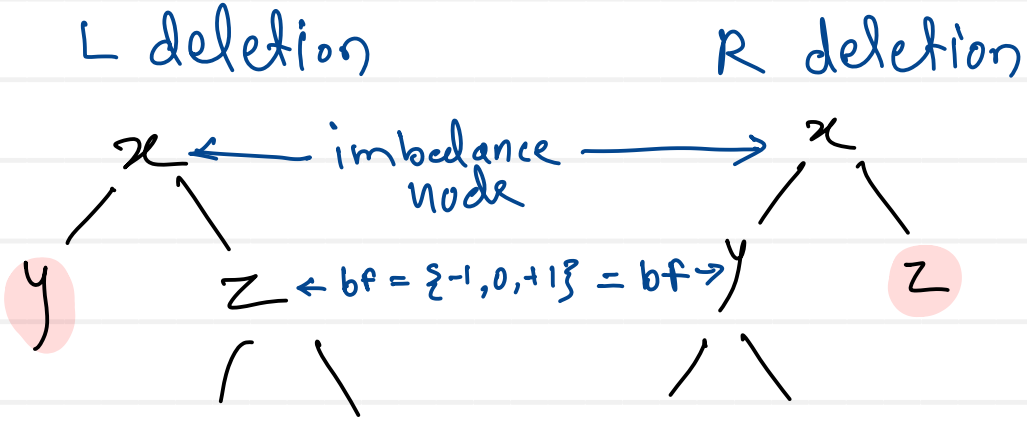


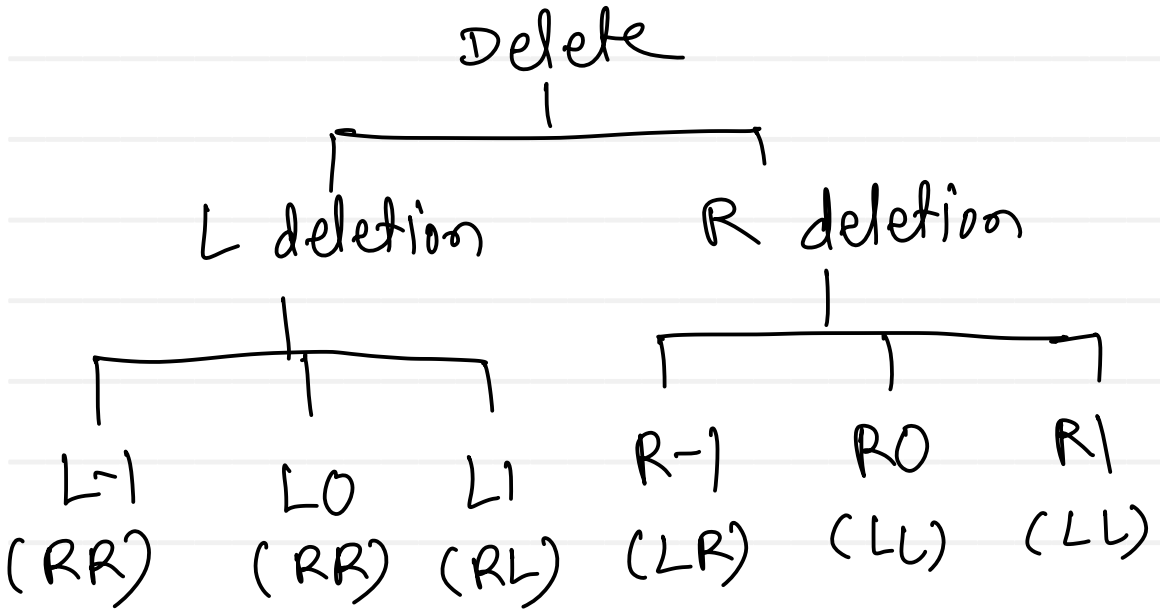




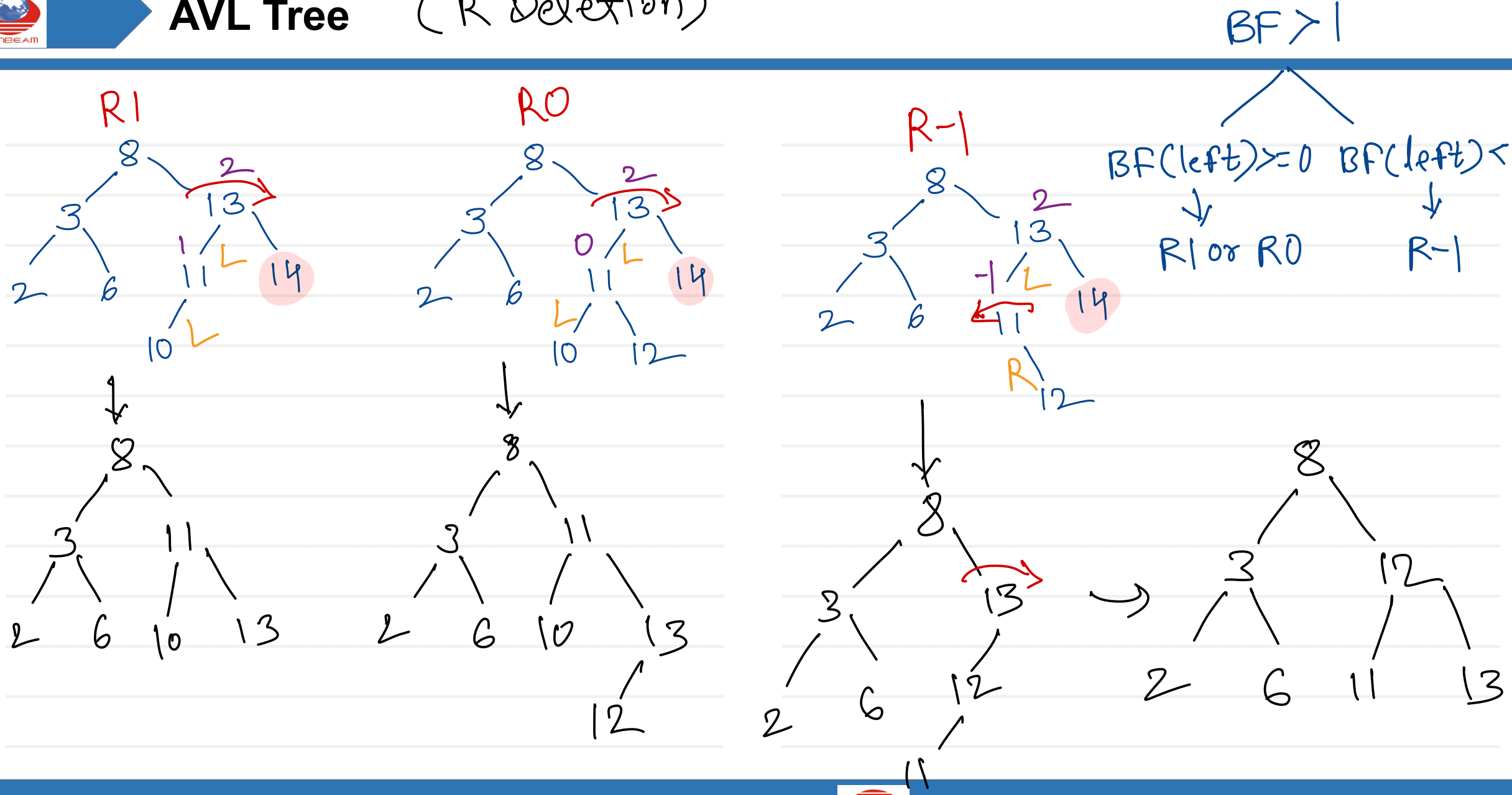
AVL Tree


Add
LL
RR
LR
RL

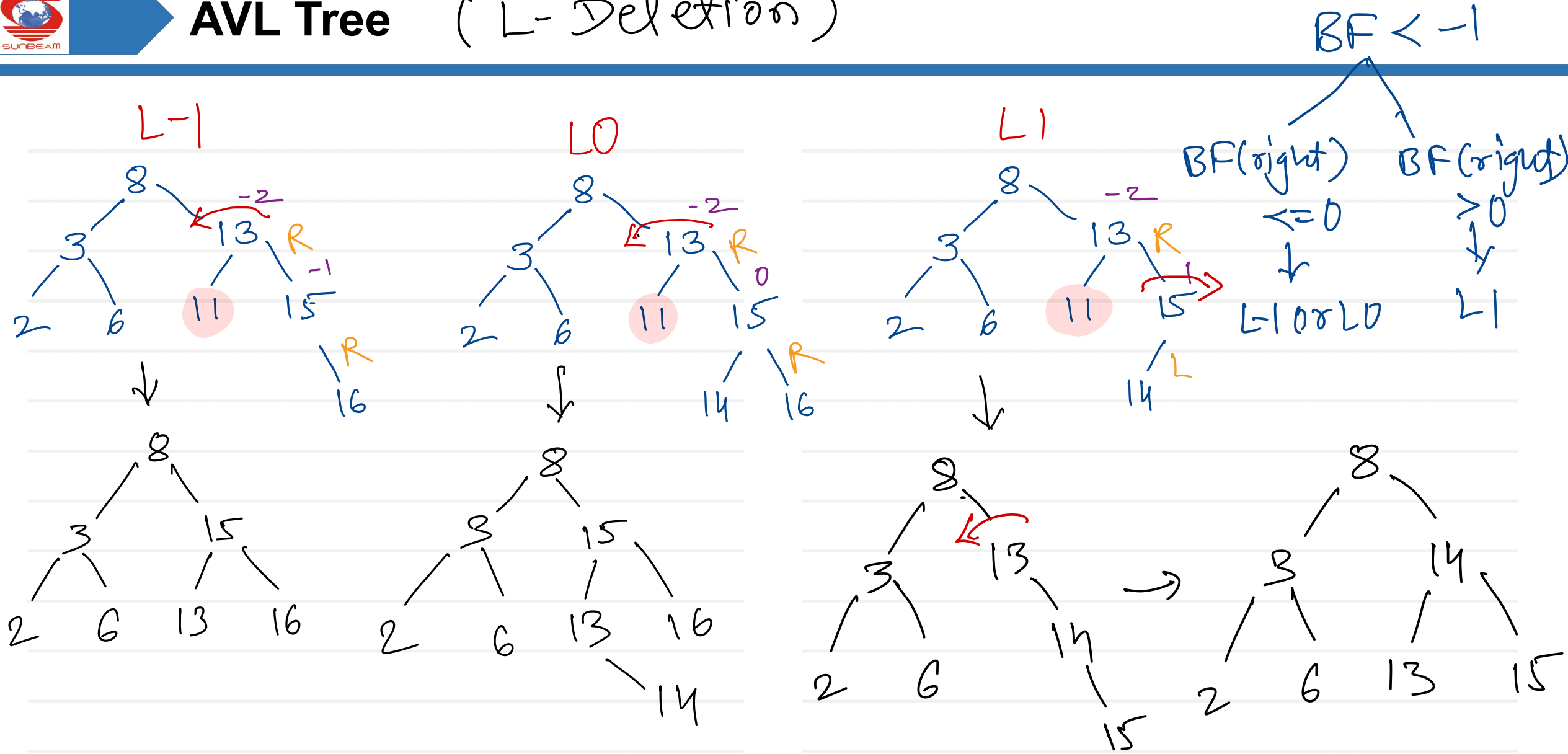

L deletion
R deletion
imbedance node
bf = {-1,0,+1} = bf


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

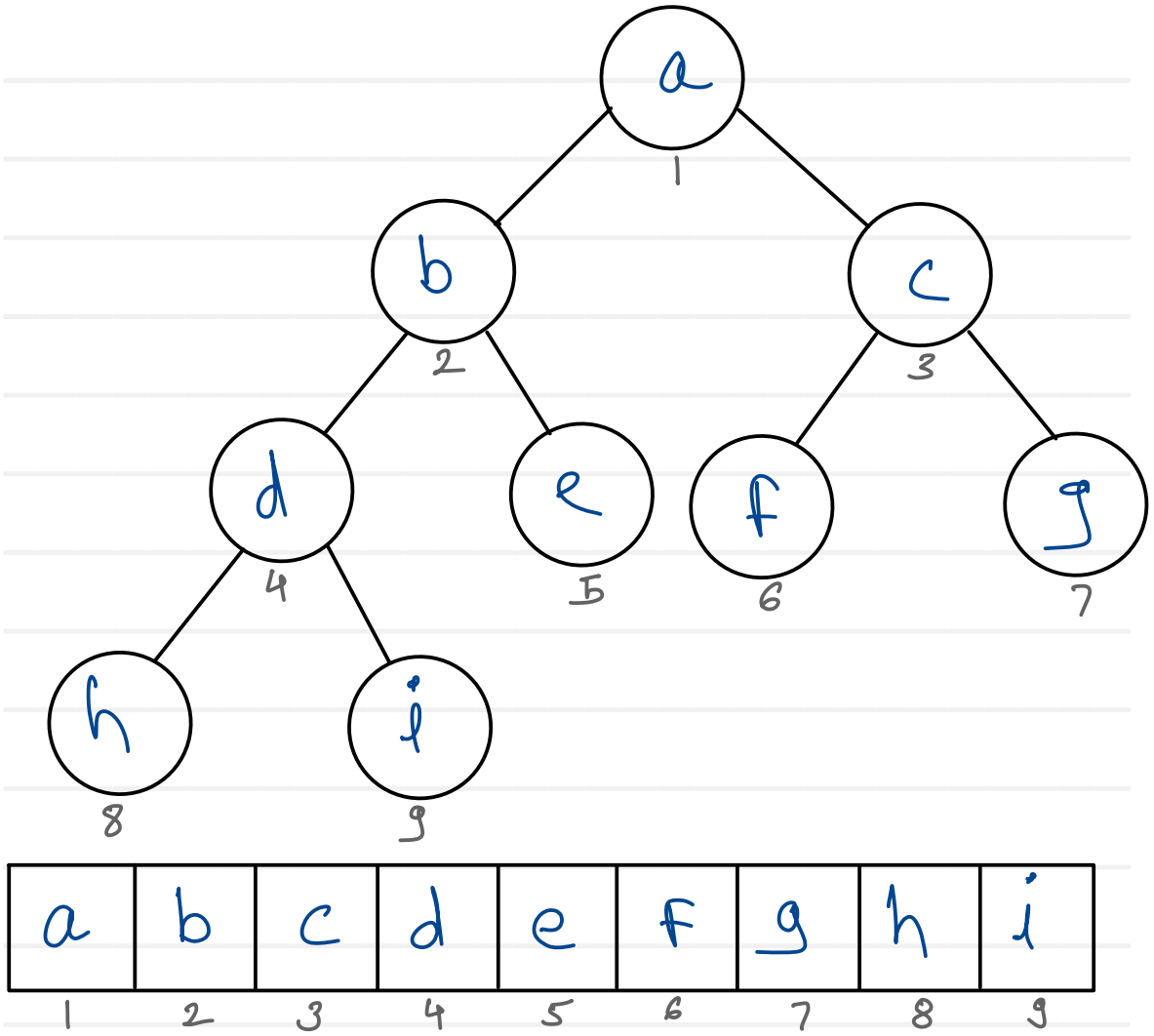
(R Deletion)


BF > 1
BF(left) >= 0
BF(left) < 0
R1 or R0
R-1
R1
R0
R-1
8
3
13
2
6
11
14
10
12
R
2
6
10
13
12
11

AVL Tree (L- Deletion)



Complete Binary Tree or Heap



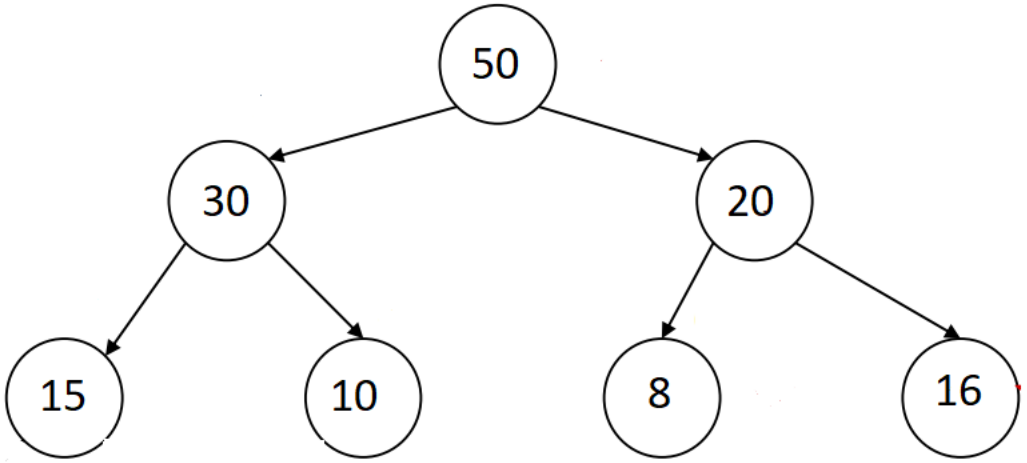
- Complete Binary Tree (height = h)
- All levels should be completely filled except last
- All leaf nodes must be at level h or h-1
- All leaf nodes at level h must aligned as left as possible
- Array implementation of Complete Binary Tree is called as heap

- Array indices are used to maintain parent child relationship.

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

Heap Types – Max and Min

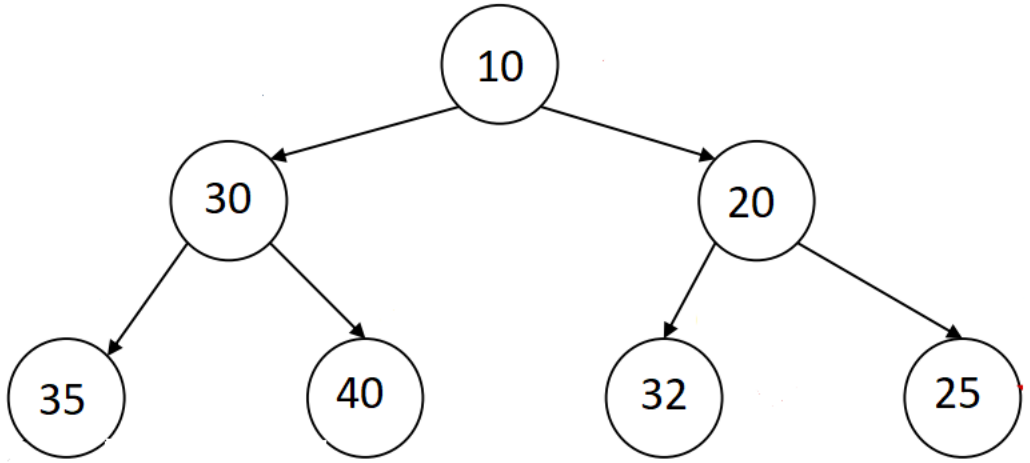
Max Heap



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

- Max heap is a heap data structure in which each node is greater than both of its child nodes.

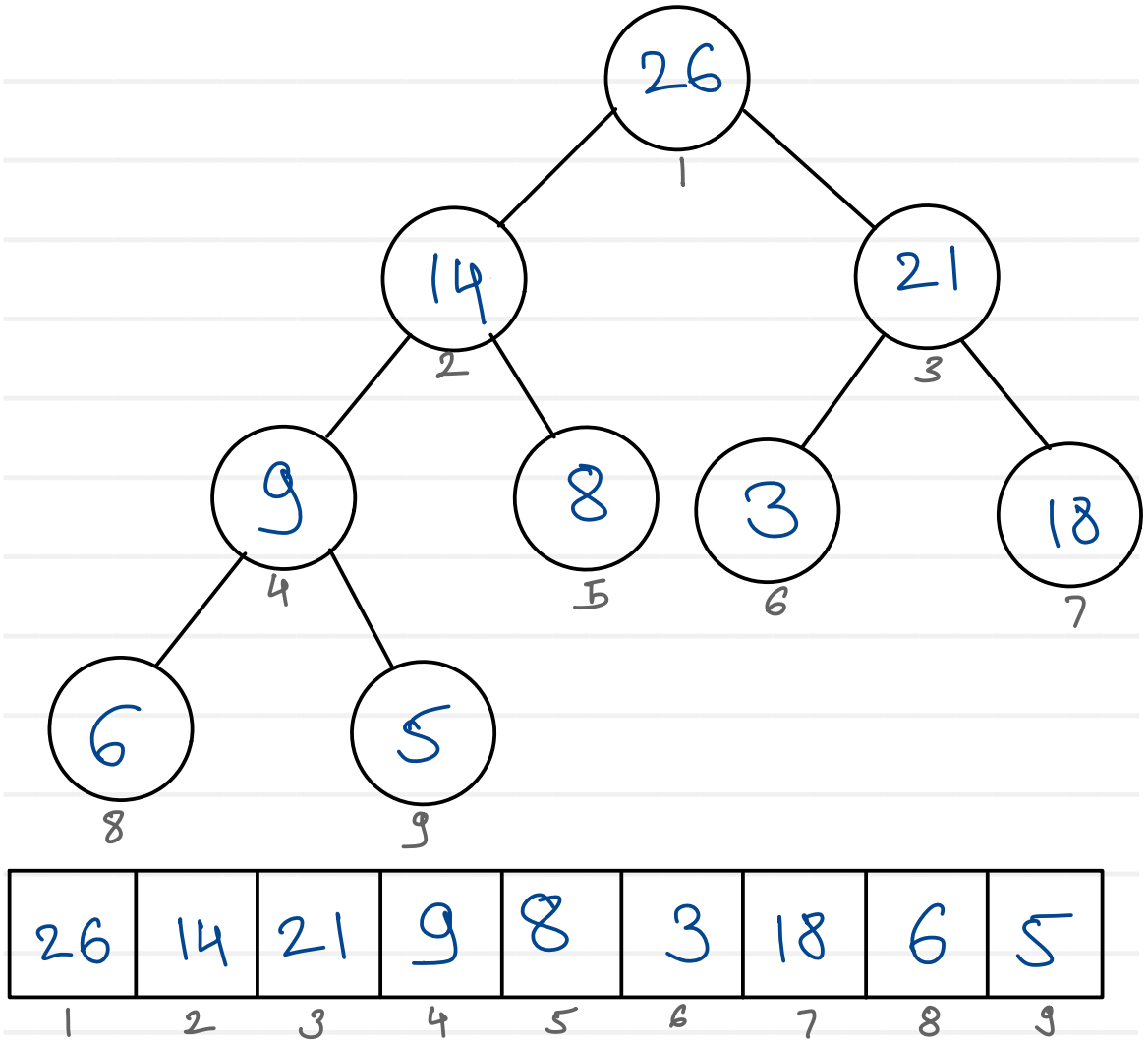
Min Heap



10	30	20	35	40	32	25
1	2	3	4	5	6	7

- Min heap is a heap data structure in which each node is smaller than both of its child nodes.

Heap - Create heap (Add)



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

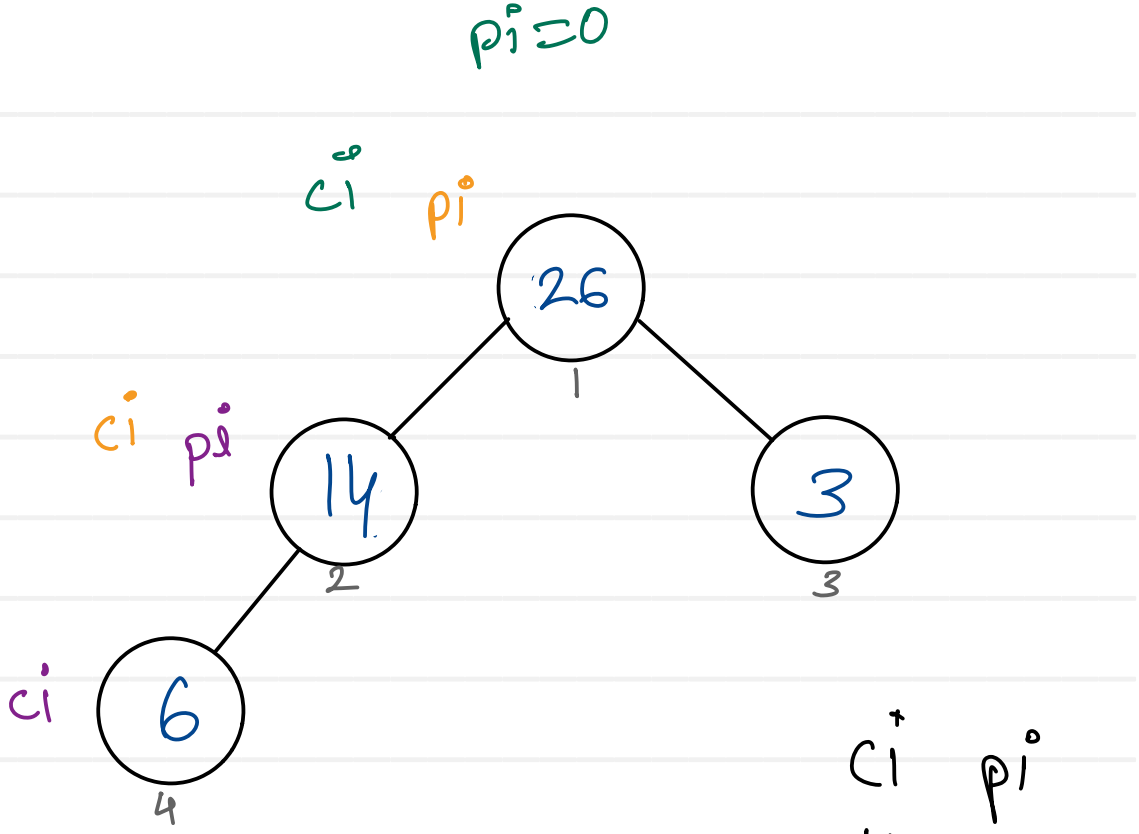
Algorithm :

1. add value at first empty index from left side into array.
2. adjust position of newly added value by comparing it will all its ancestors one by one.

- to adjust position of newly added value, need to traverse from leaf to root positions.

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

Heap - Add

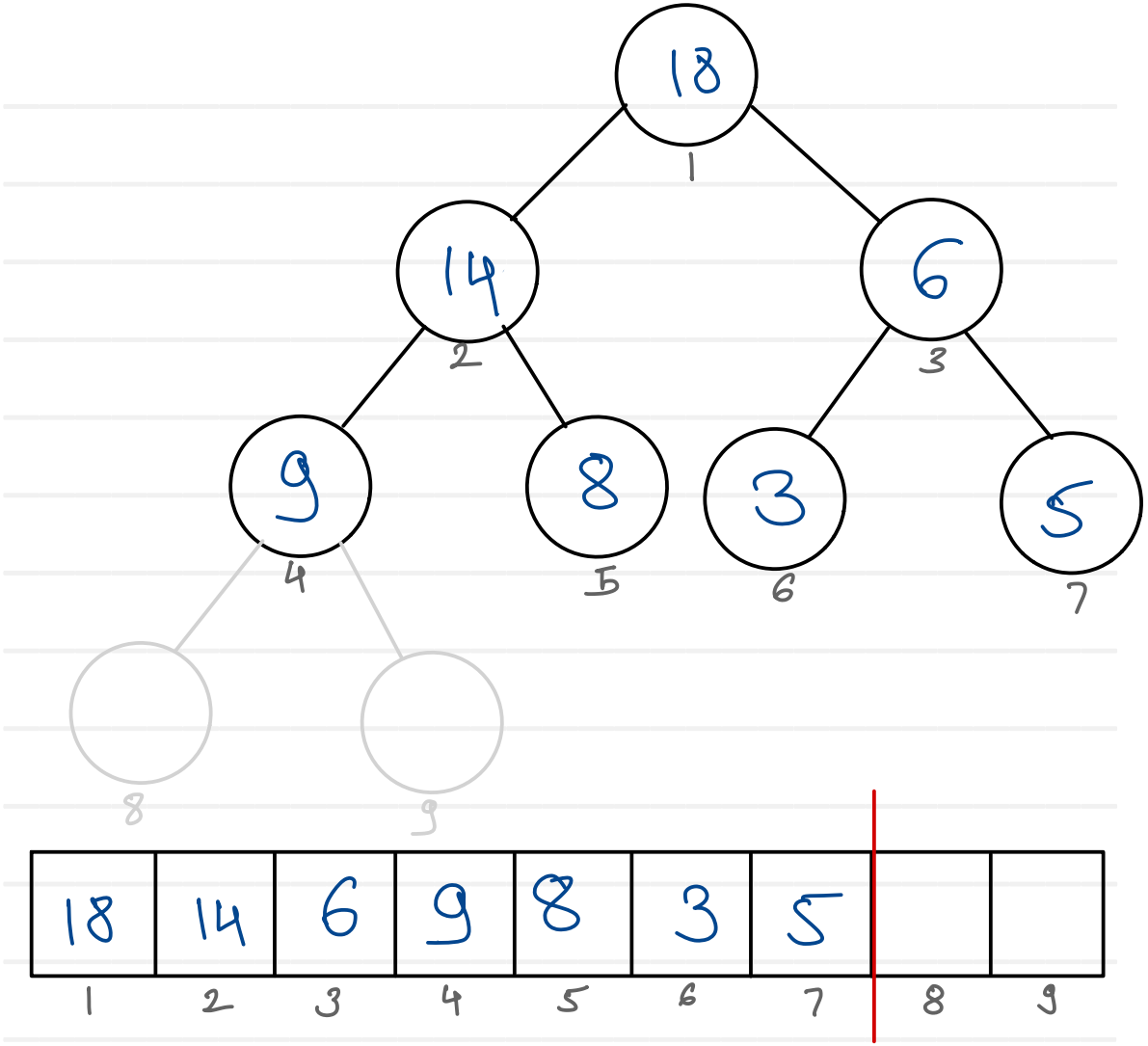



```
int arr[10];
int SIZE = 0;
```

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)



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, 21

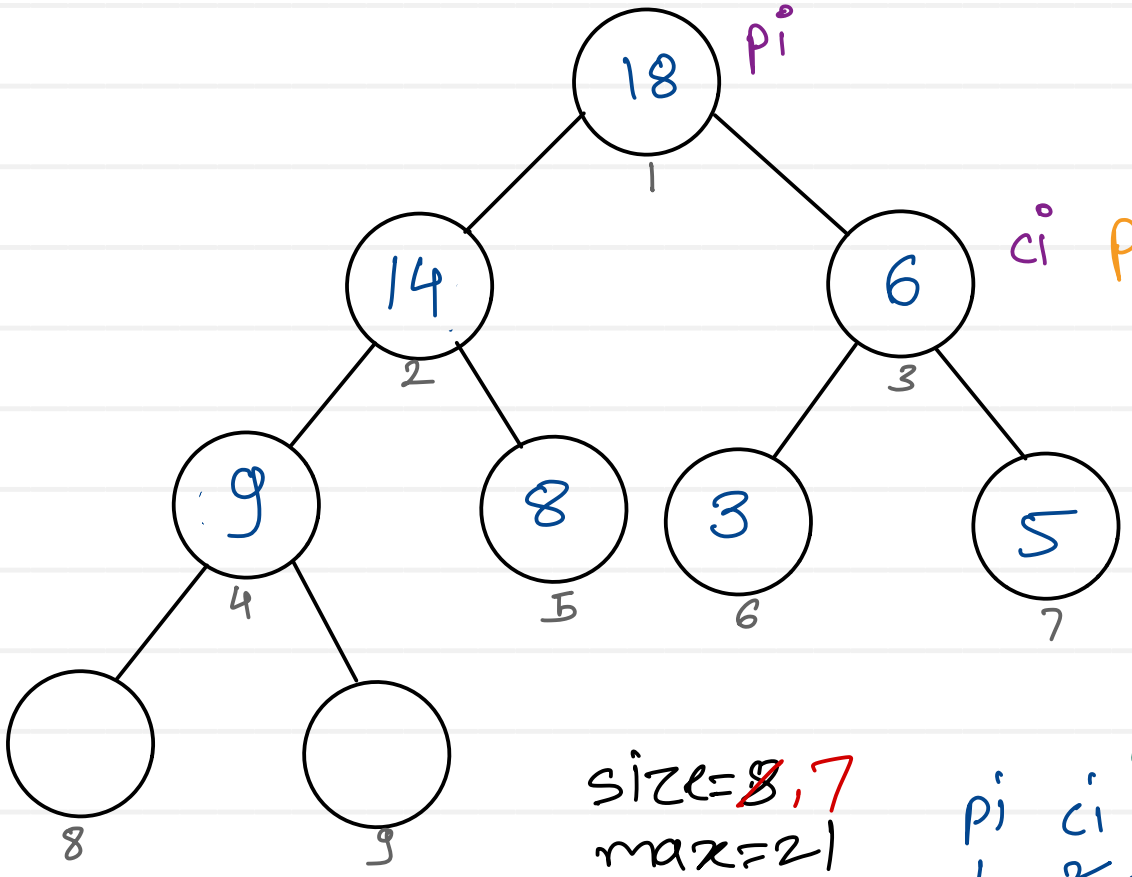
Algorithm :

1. place last value of heap at root position.
2. adjust position of new root by comparing it with all its descendents one by one

- to adjust position of new root, need to traverse from root to leaf position

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

Heap - Delete

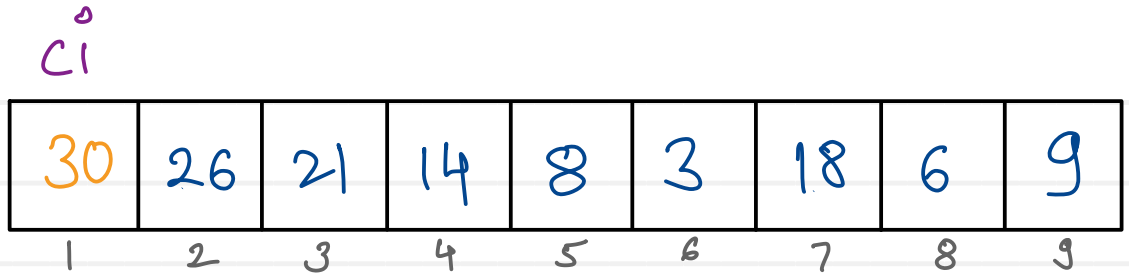


size=8,7
max=21

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;
```

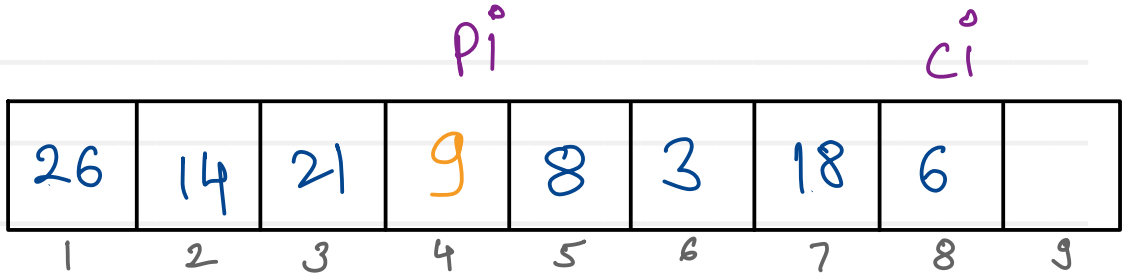


SIZE = 9

ci	pi
9	4
4	2
2	1
1	0 — stop

ci = pi

pi = ci/2



max = 30

SIZE = 8

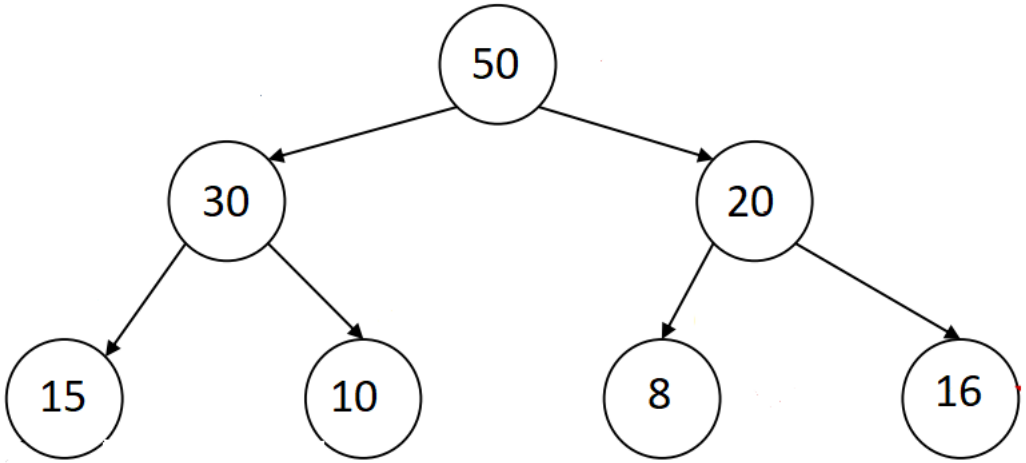
pi	ci
1	2
2	4
4	8

pi = ci

ci = pi * 2

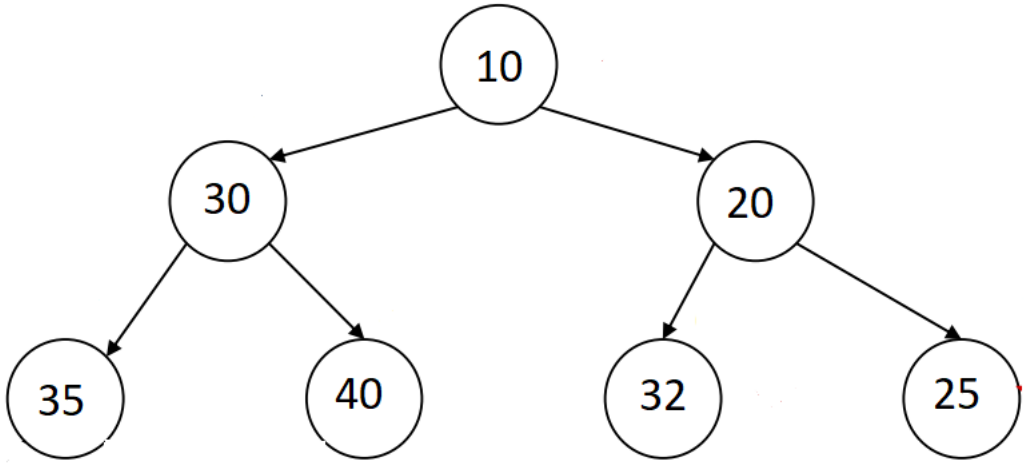
Priority Queues

Higher number
Higher priority



- In Max heap always root element which has highest value is removed

Lower number
Higher priority



- In Min heap always root element which has lowest value is removed

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

Priority Queue

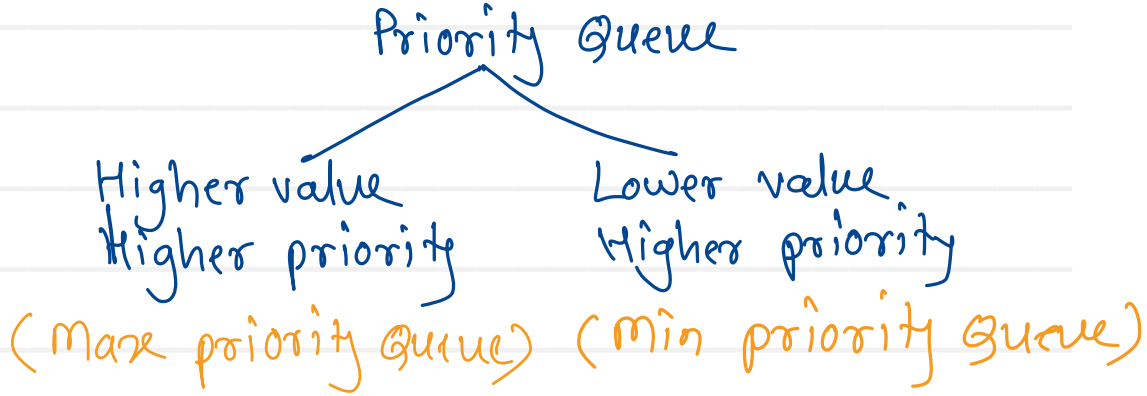
- Queue where high priority data is always peeked or deleted.
- Priority can be implemented using array, linked list and heap data structures.
- An array is used to store a value and its associated priority. In some simpler implementations, the value itself might represent the priority (e.g., lower value means higher priority).
- Array and linked list implementation of priority queue often leads to less efficient performance compared to heap-based implementations for insertion and deletion operations.

- **Ordered vs. Unordered Array**
  - **Ordered Array** / linked list
    - Elements are kept sorted by priority. Insertion requires shifting existing elements to maintain order, leading to O(n) time complexity for insertion.
    - Deletion of the highest priority element is O(1) as it's typically at the beginning or end of the array.
  - **Unordered Array:** / linked list
    - Elements are inserted without regard to order, making insertion O(1).
    - Deletion of the highest priority element requires searching the entire array to find it, resulting in O(n) time complexity for deletion.

Priority Queue Implementation

- Priority : number associated with value
- Priority range is defined by programmer

  e.g. Priority range : 1 to 10



- Every element of priority queue will have two parts:

  value : any data type

  priority : integer

```
struct item {
    int value;
    int priority;
};
```

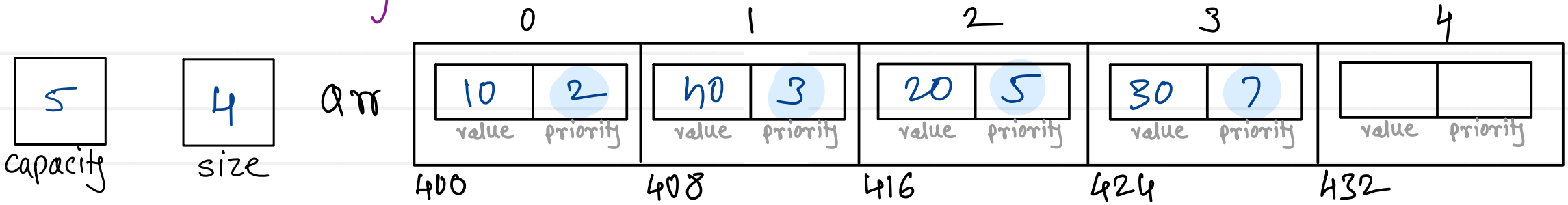
```
struct priorityQueue {
    struct item arr[5];
    int capacity;      (maxSize)
    int size;
};
```

Operations :

1) Enqueue
2) Dequeue
3) Peek
4) isEmpty → size == 0
5) isFull → size == capacity

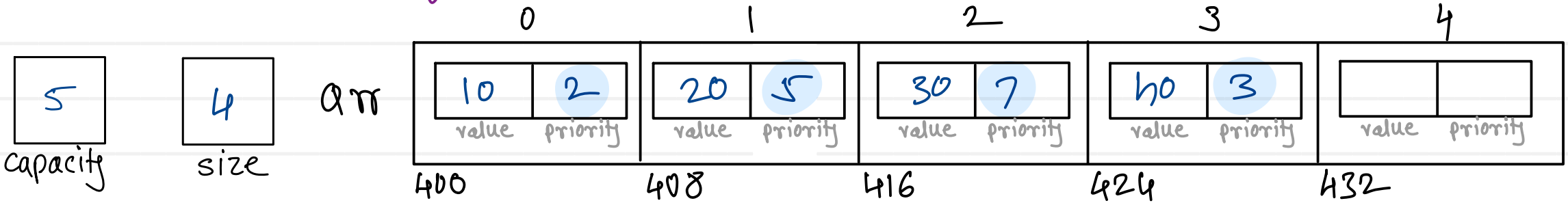
Priority Queue Implementation

Ordered array:



insert - O(n) delete - O(1) peek - O(1)
for shifting - O(n)

Unordered array:



insert - O(1) delete - O(n) peek - O(n)
for shifting - O(n)

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com