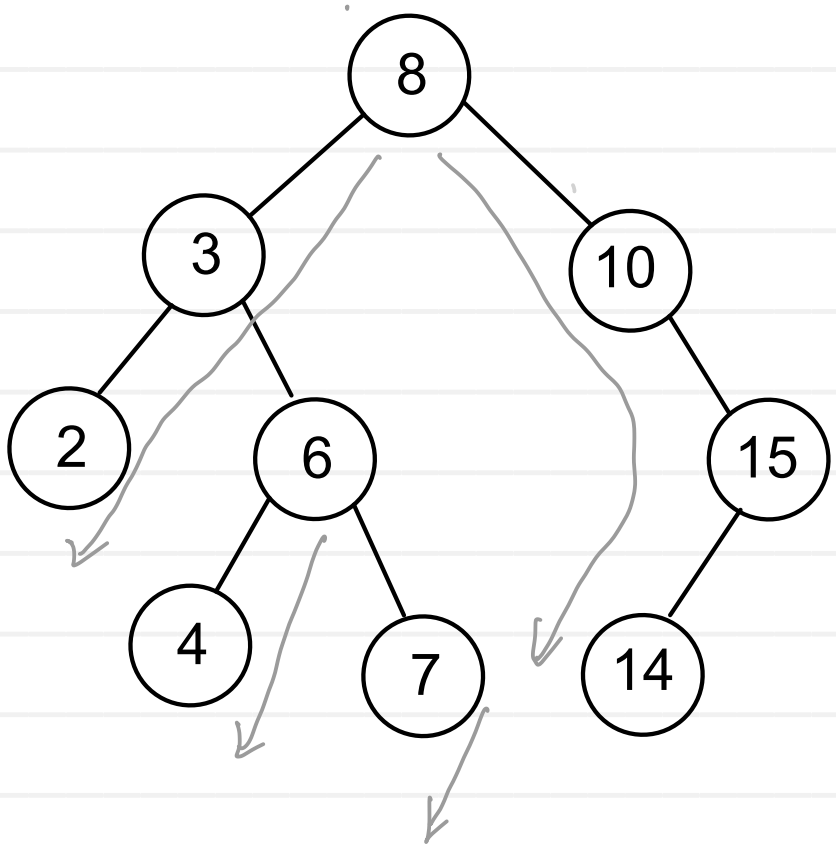

**Sunbeam Institute of Information Technology**
**Pune and Karad**

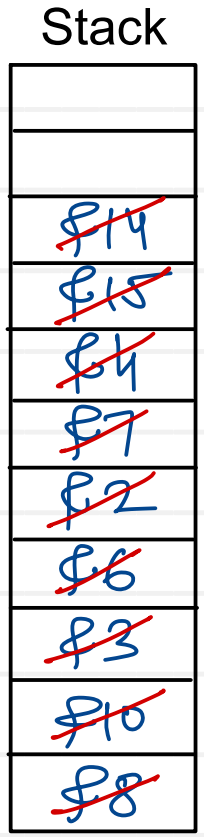
# Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

# Binary Search Tree - DFS Traversal (Depth First Search)

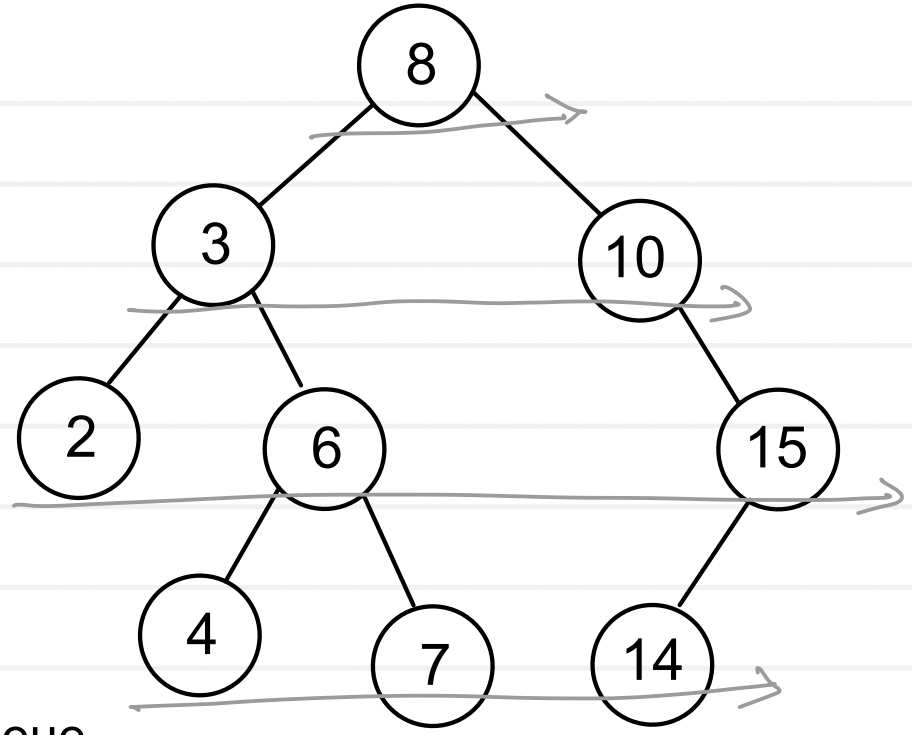



1. Push root node on stack
2. Pop one node from stack
3. Visit ( print ) popped node
4. If right exists, push it on stack
5. If left exists, push it on stack
6. While stack is not empty, repeat step 2 to 5

Traversal : 8, 3, 2, 6, 4, 7, 10, 15, 14

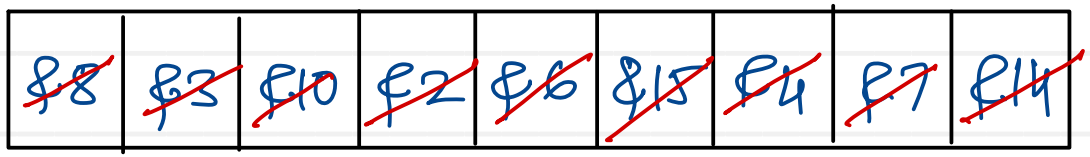
# Binary Search Tree - BFS Traversal (Bredth First Search)




1. Push root node on queue
2. Pop one node from queue
3. Visit ( print ) popped node
4. If left exists, push it on queue
5. If right exists, push it on queue
6. While queue is not empty, repeat step 2 to 5

level order traversal

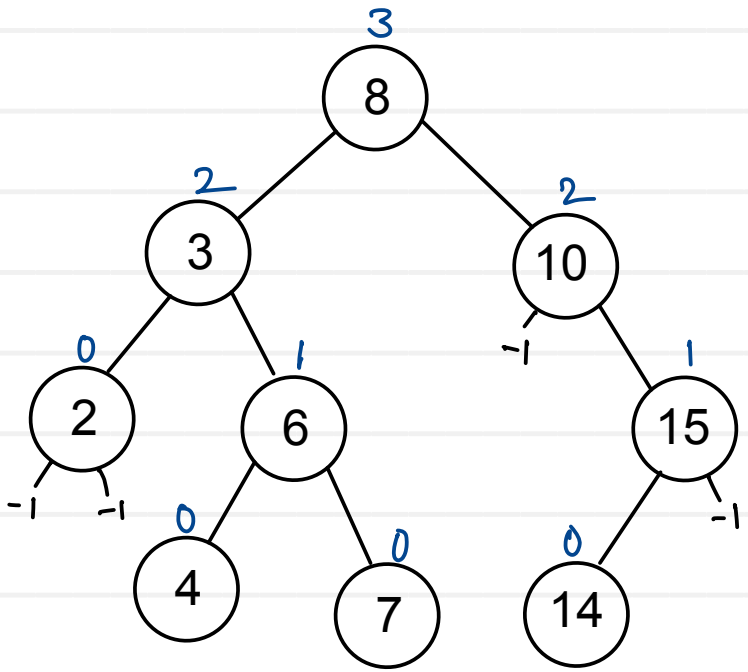
Queue




Traversal : 8, 3, 10, 2, 6, 15, 4, 7, 14

# Binary Search Tree - Height

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

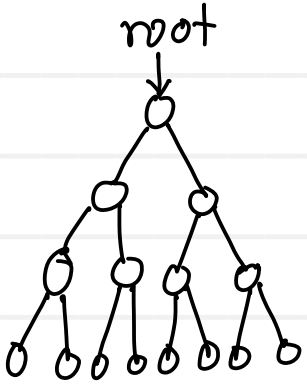



1. If left or right sub tree is absent then return -1
2. Find height of left sub tree
3. Find height of right sub tree
4. Find max height
5. Add one to max height and return

# BST - Time complexity of operations

Capacity : max number of nodes for given height.

| height | No. of Node |
|---|---|
| -1 | 0 |
| 0 | 1 |
| 1 | 3 |
| 2 | 7 |
| 3 | 15 |




n – no. of nodes
h – height of tree

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

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

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

$$2^h \approx n$$

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

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

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

Time $\propto$ height

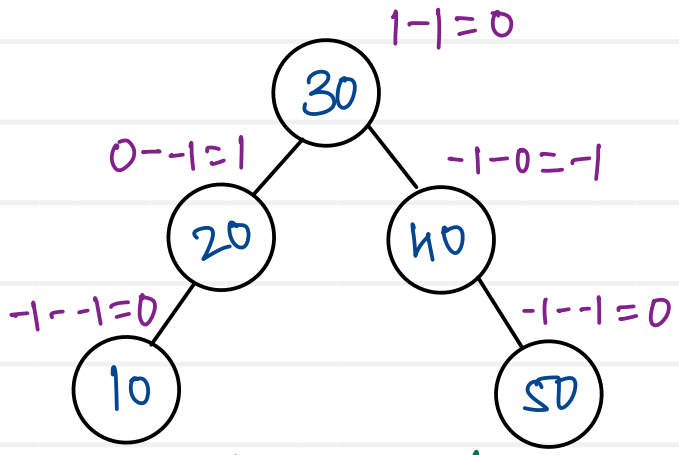
Time $\propto \frac{\log n}{\log 2}$

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

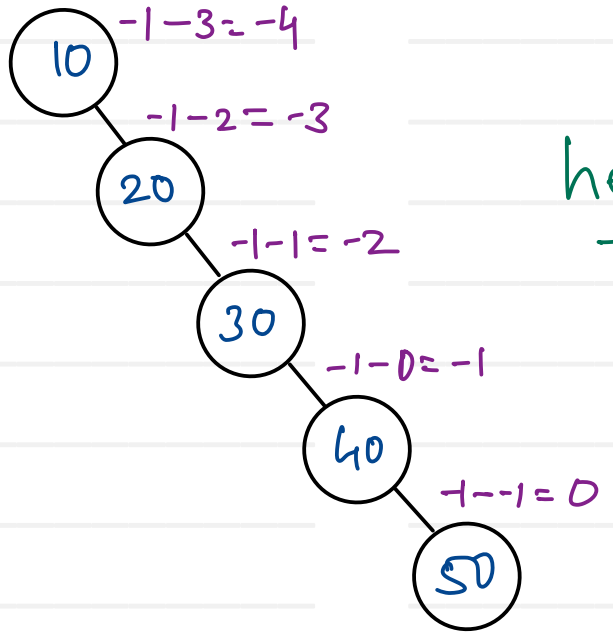
Traversal : O(n)

# Skewed Binary Search Tree

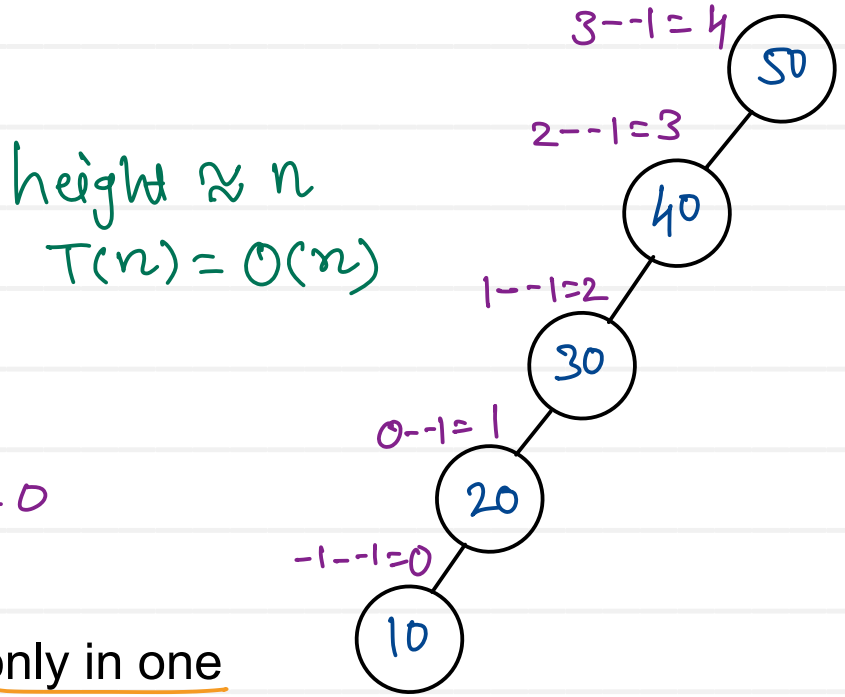
Keys : 30, 40, 20, 50, 10




Keys : 10, 20, 30, 40, 50




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

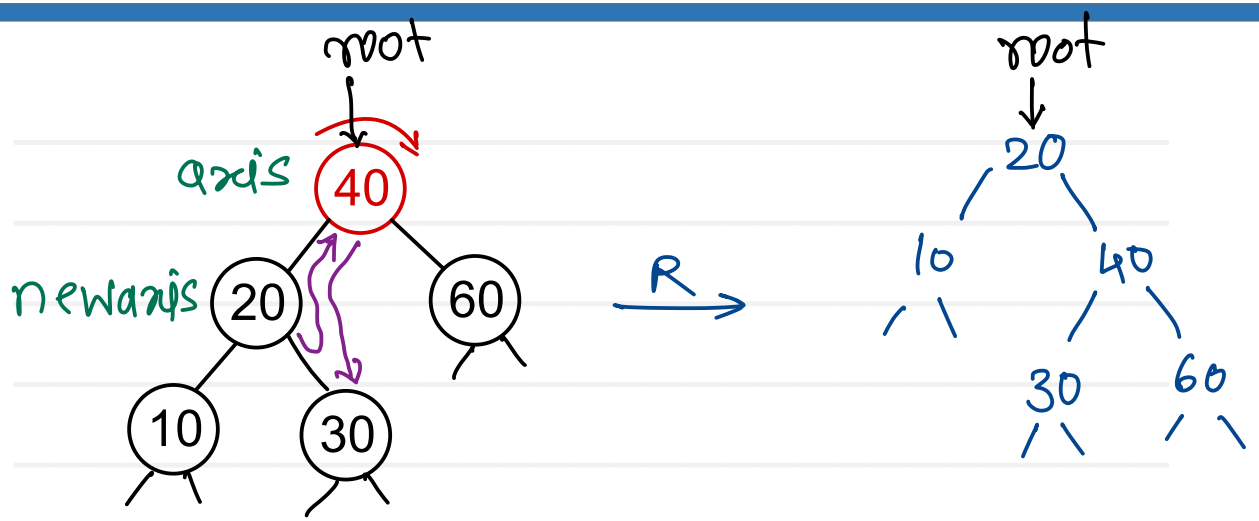
- To speed up searching, height of BST should be minimum as possible
- If nodes in BST are arranged, so that its height is kept as less as possible, is called as Balanced BST

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

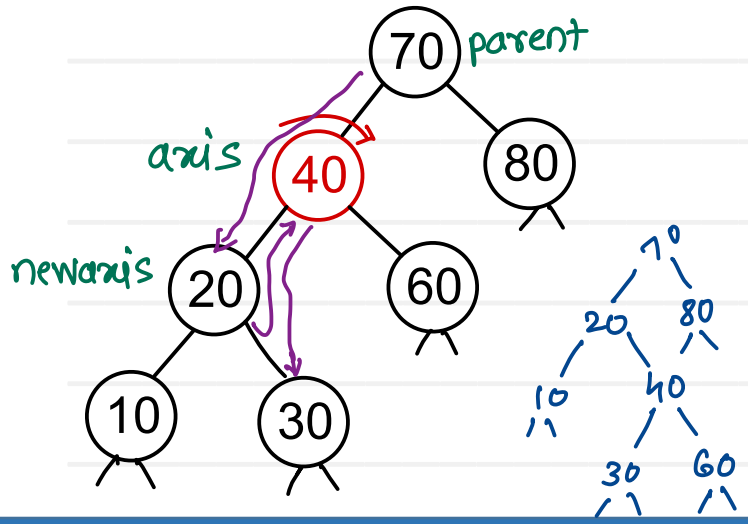
- tree is balanced if balance factors of all the nodes is either -1, 0 or +1
- balance factors = {-1, 0, +1}
- A tree can be balanced by applying series of left or right rotations on imbalance nodes (having balance factor other than -1, 0 or +1)

- types of balanced BST:
  1. AVL tree
  2. Red Black tree
  3. Splay tree

# Right Rotation

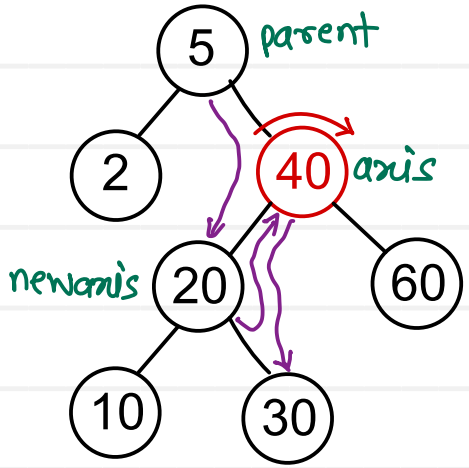


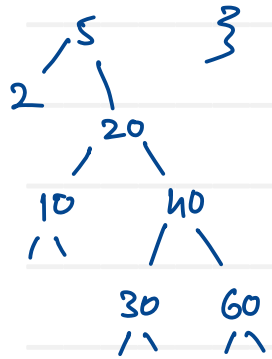

```
rightRotation ( axis, parent) {
    newaxis = axis.left;
    axis.left = newaxis.right;
    newaxis.right = axis;
    if( axis == root)
        root = newaxis;
    else if( axis == parent.left)
        parent.left = newaxis;
    else if( axis == parent.right)
        parent.right = newaxis;
}
```



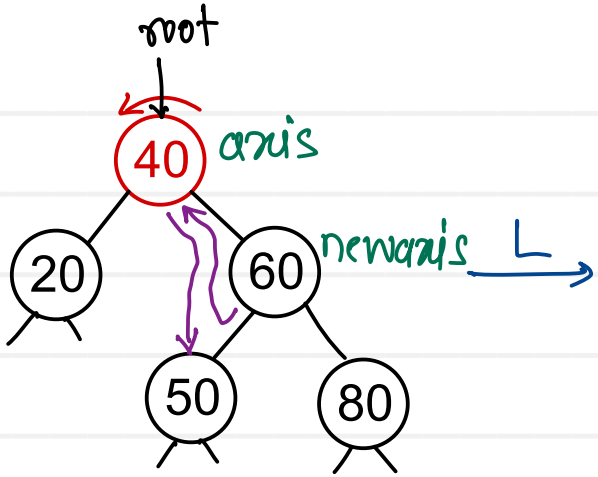


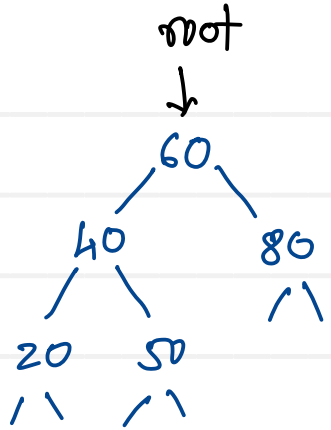




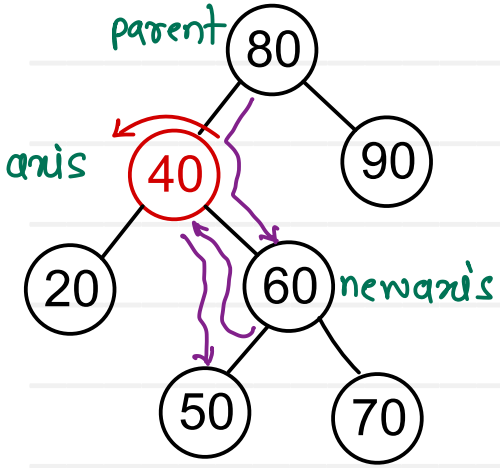



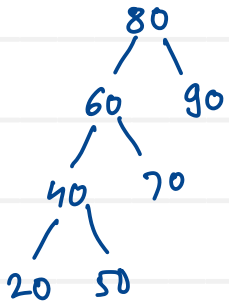
# Left Rotation

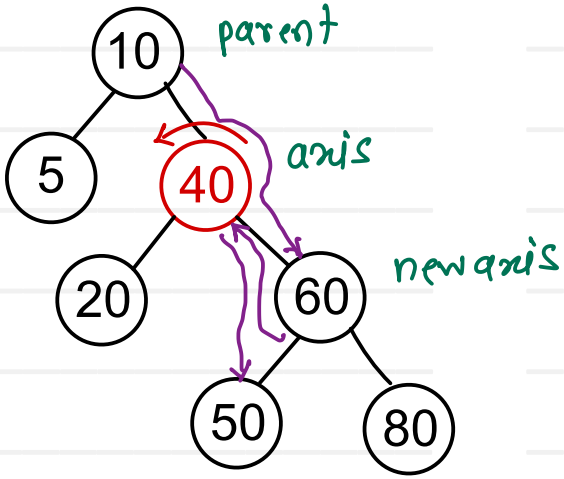


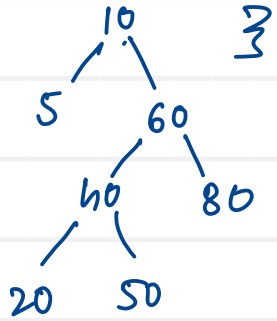




```
leftRotation(axis, parent) {
    newaxis = axis.right;
    axis.right = newaxis.left;
    newaxis.left = axis;
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```

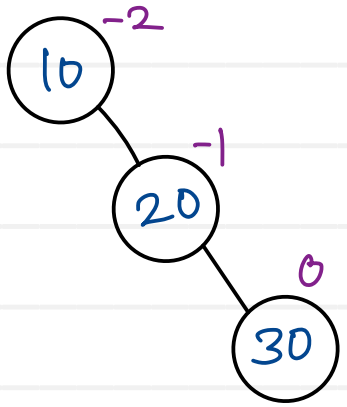


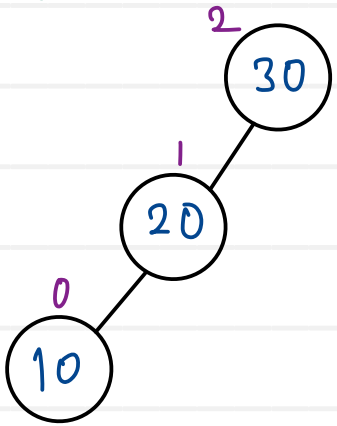



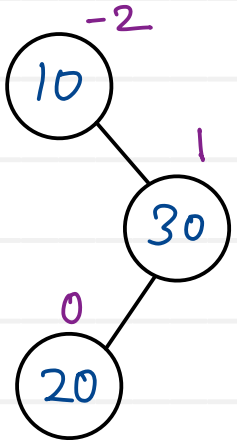
Keys : 10, 20, 30



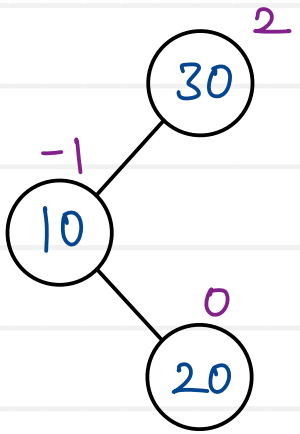

Keys : 30, 20, 10



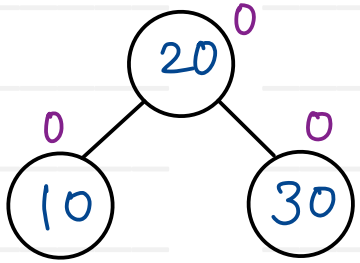

Keys : 10, 30, 20




Keys : 30, 10, 20




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




Balanced BST

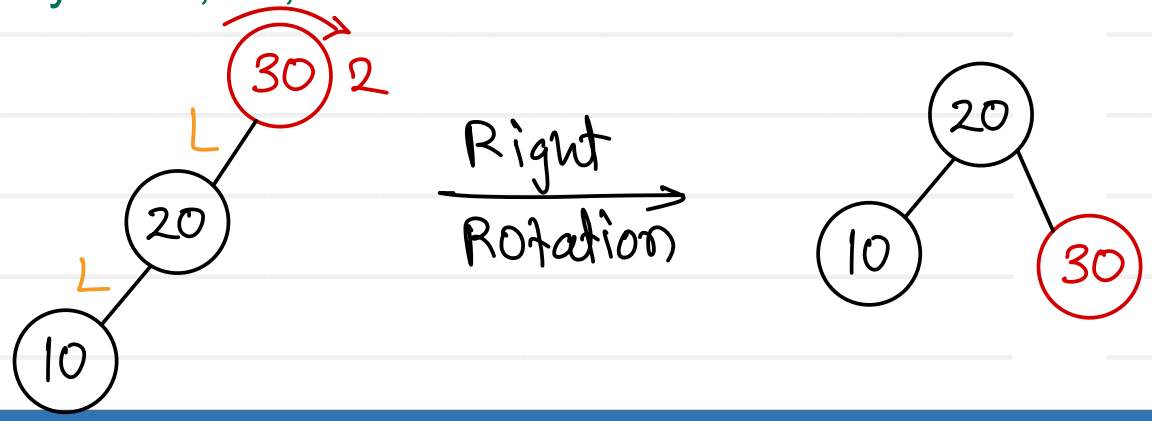
# Rotation cases (Single Rotations)

RR Imbalance : apply left rotation on imbalance node

Keys : 10, 20, 30


LL Imbalance : apply right rotation on imbalance node

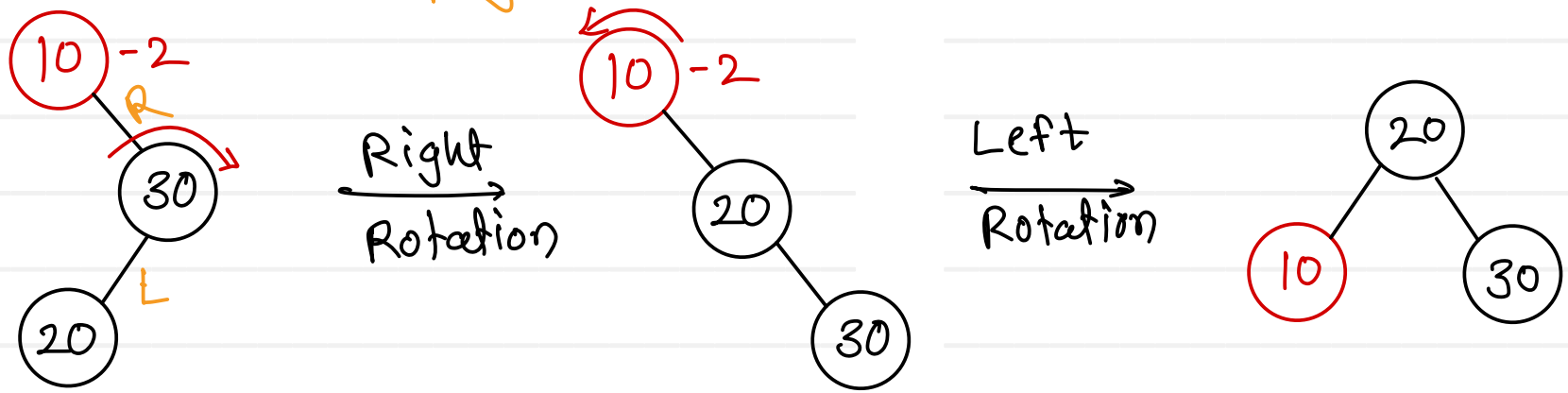
Keys : 30, 20, 10



# Rotation cases (Double Rotations)

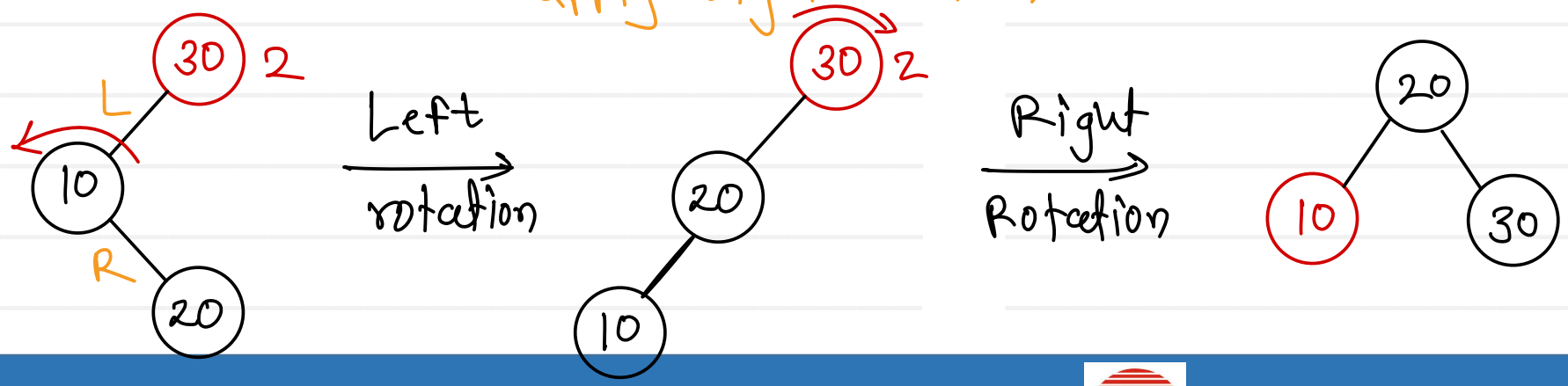
RL Imbalance :- apply right rotation on right of imbalance node
apply left rotation on imbalance node

Keys : 10, 30, 20




LR Imbalance :- apply left rotation on left of imbalance node
apply right rotation on imbalance node

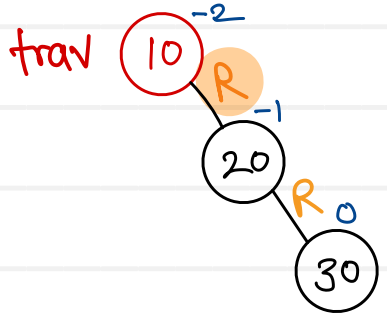
Keys : 30, 10, 20



BF < -1

RR Imbalance

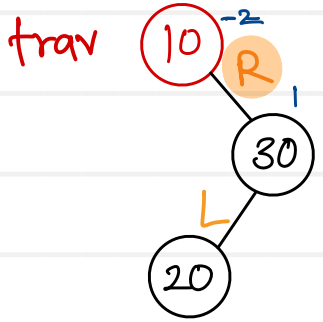
Keys : 10, 20, 30




value > trav.right.data

RL Imbalance

Keys : 10, 30, 20

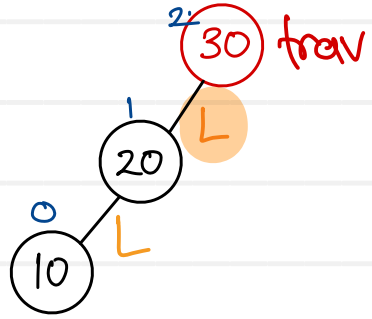



value < trav.right.data

BF > 1

LL Imbalance

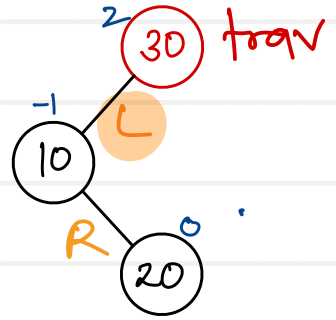
Keys : 30, 20, 10




value < trav.left.data

LR Imbalance

Keys : 30, 10, 20

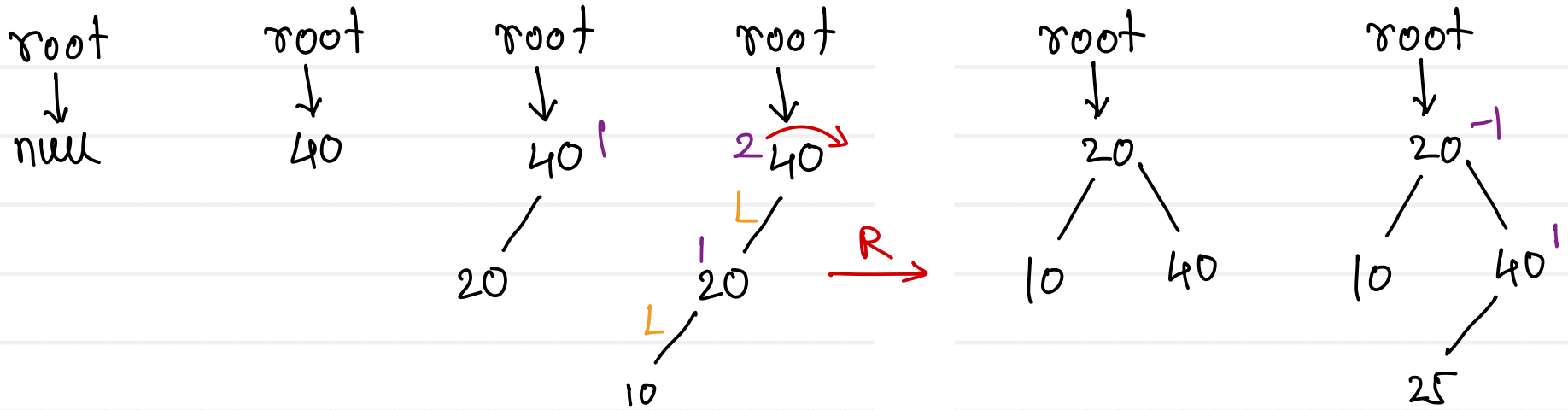



value > trav.left.data

# AVL Tree

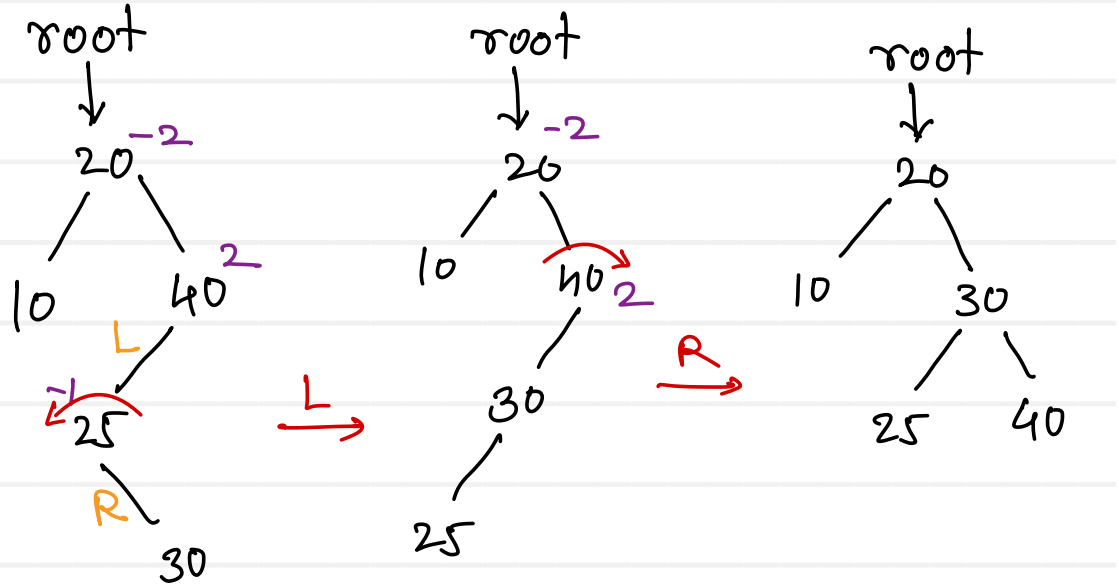
- self balancing binary search tree
- on every insertion and deletion of a node, tree is getting balanced by applying rotations on imbalance nodes
- The difference between heights of left and right sub trees can not be more than one for all nodes
- Balance factors of all the nodes are either -1 , 0 or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

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

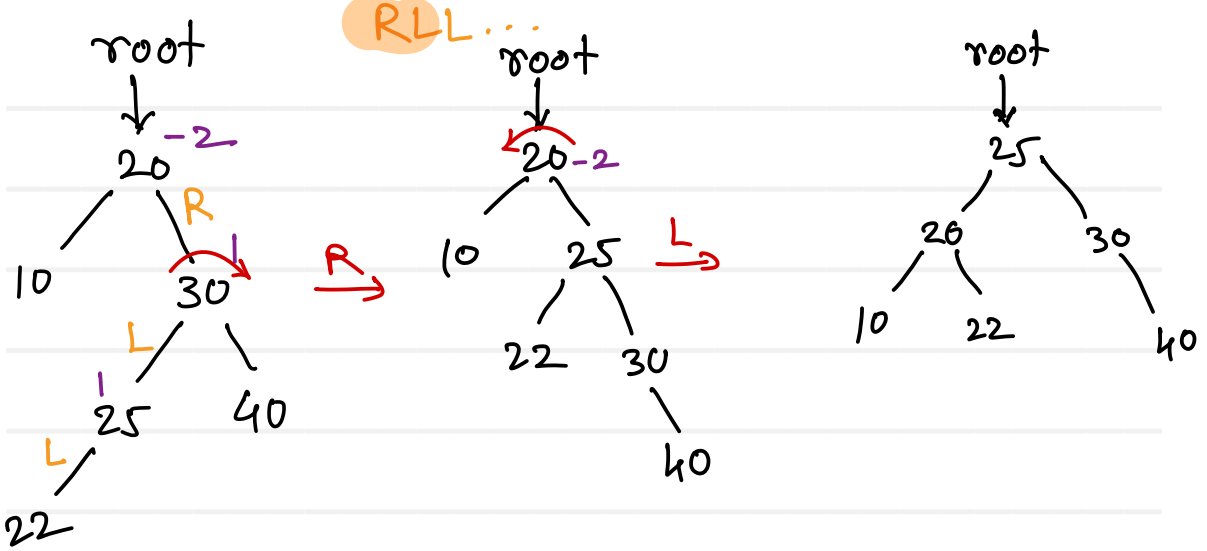


# AVL Tree

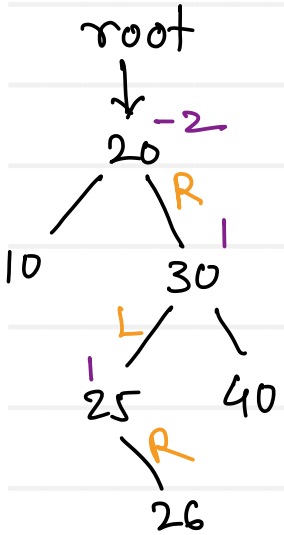
Keys : 40, 20, 10, 25, 30, 22, 50




RLL...

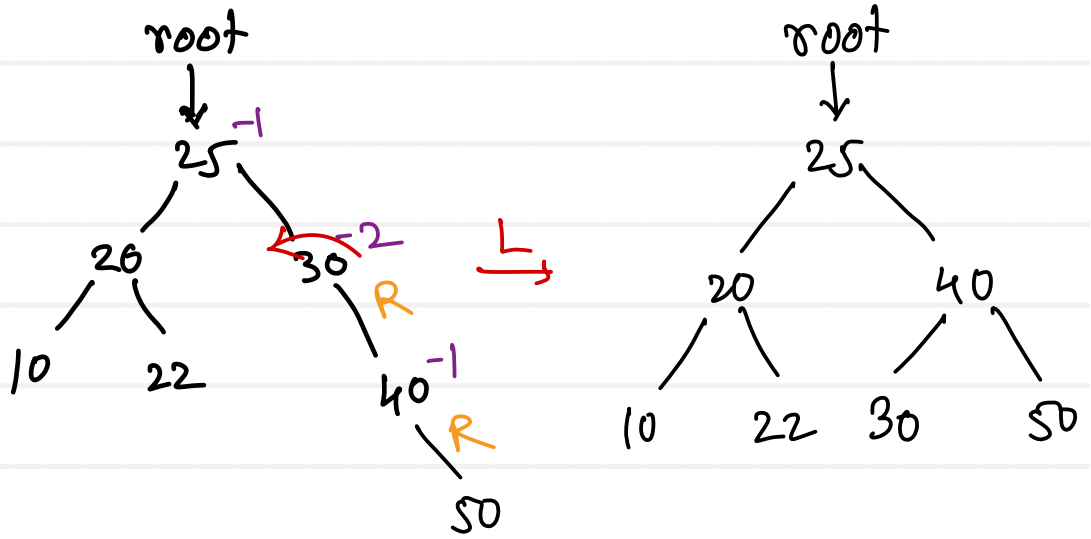



RLR...



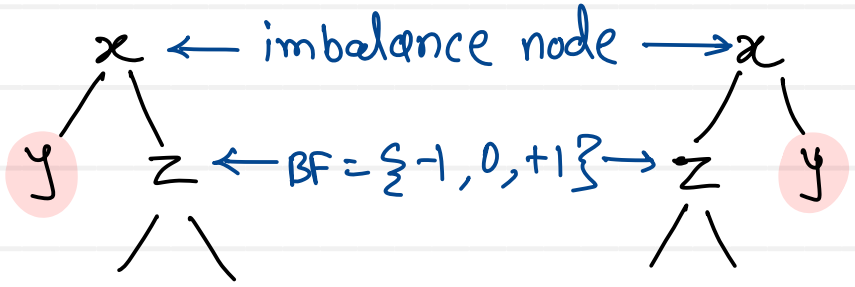
# AVL Tree

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

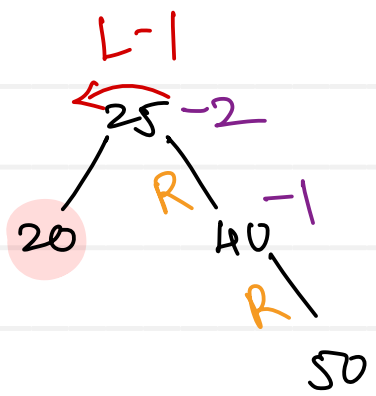


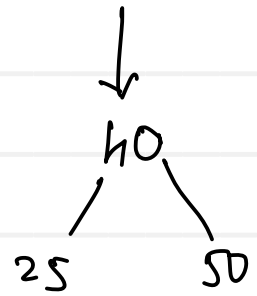

L-Deletion
BF < -1

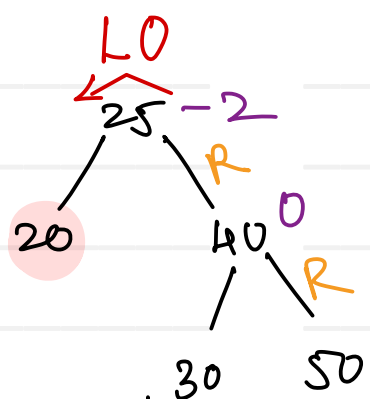
R-Deletion
BF > +1

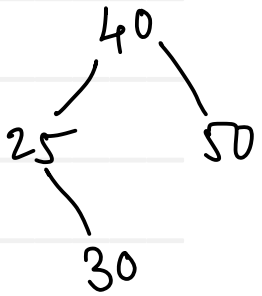


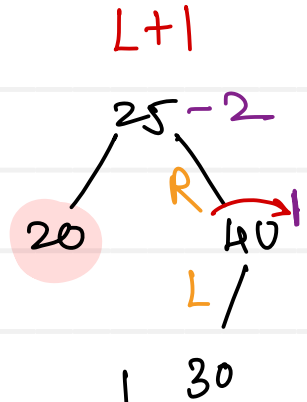
L - Deletion


L-1
25
-2
20
R
40
-1
R
50

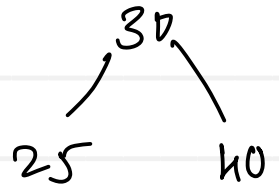

40
25
50


L0
25
-2
R
20
40
0
R
30
50

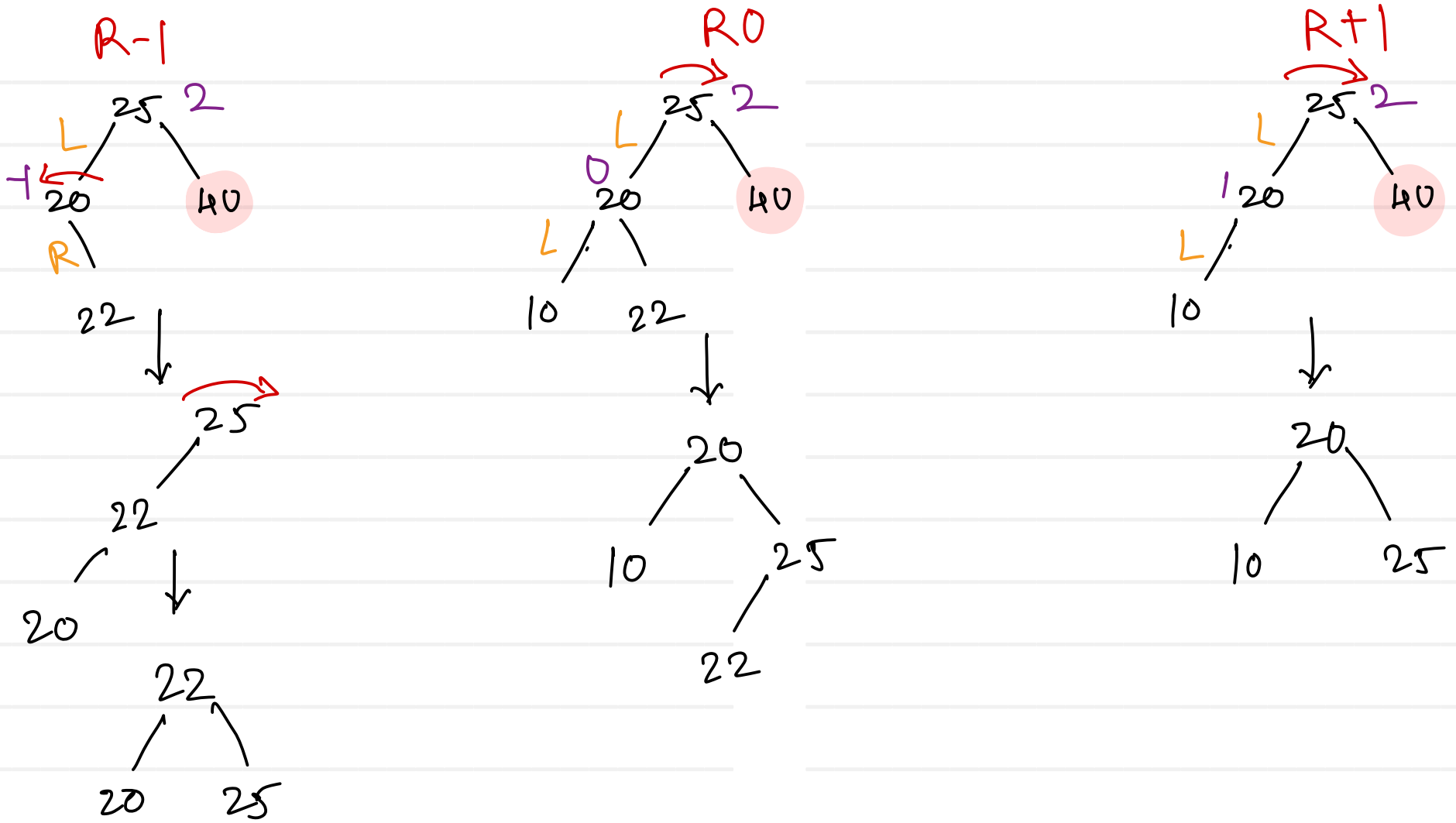

40
25
50
30


L+1
25
-2
R
20
40
1
L
30

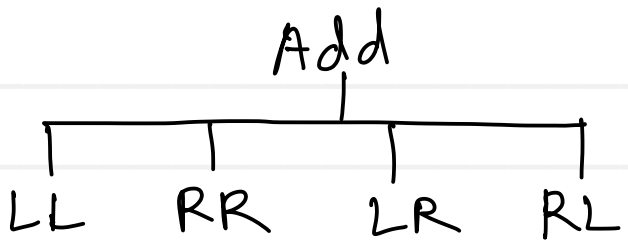

25
30
40

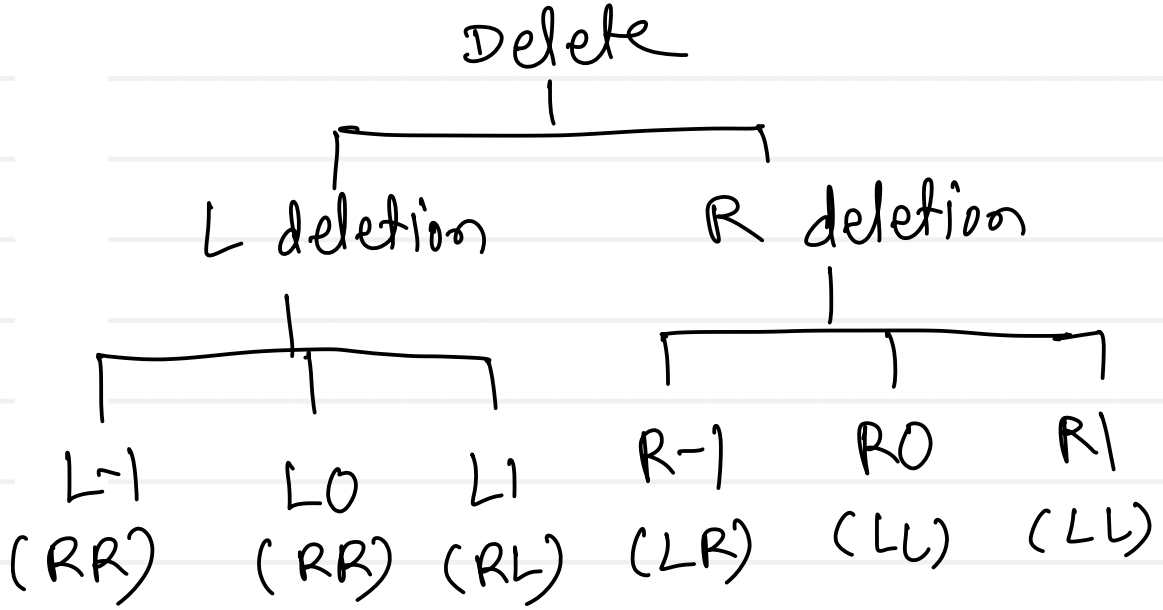

30
25
40

# R - Deletion

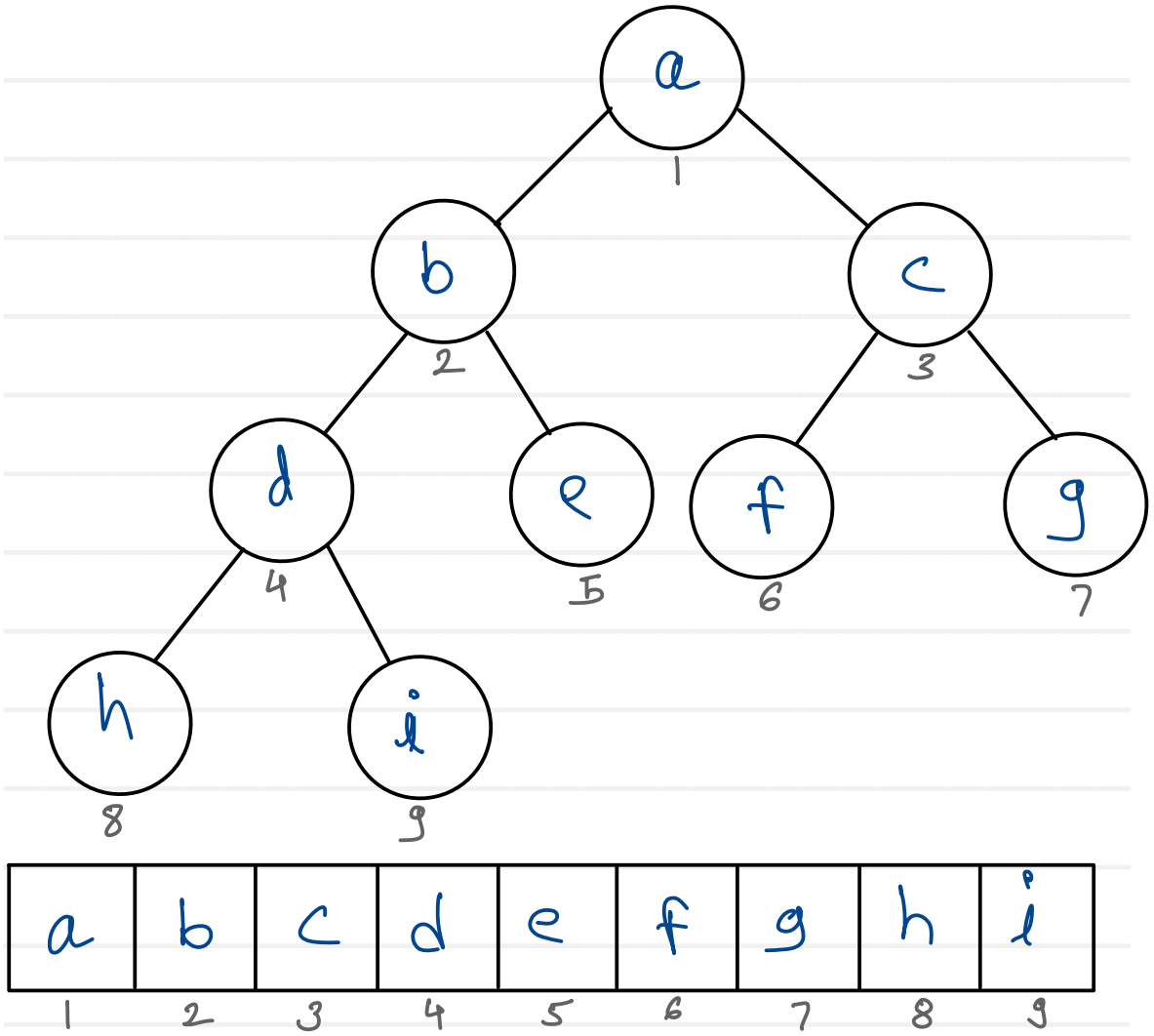

R-1
25
2
L
+1
20
40
R
22
25
22
20
22
20
25
R0
25
2
L
0
20
40
L
10
22
20
10
25
22
R+1
25
2
L
1
20
40
L
10
20
10
25

# AVL Tree


Add
LL
RR
LR
RL


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

# 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

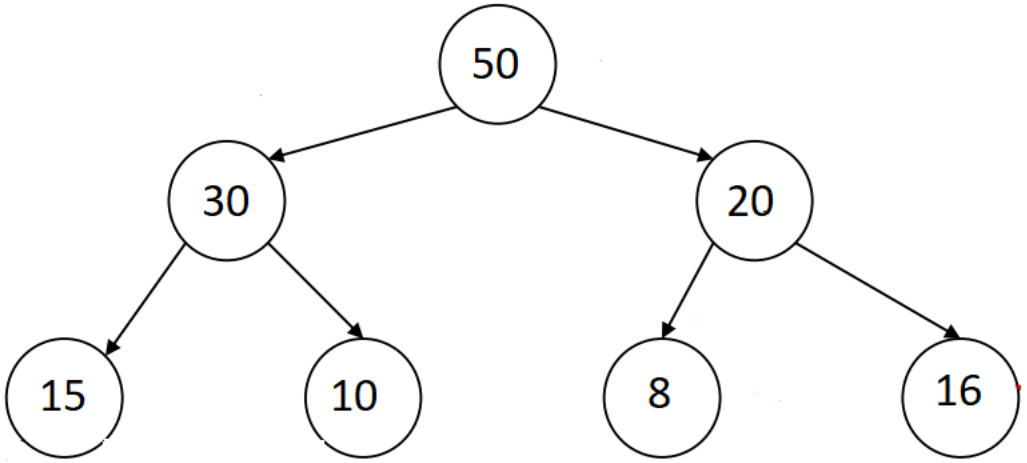
parent - $i/2$ index

left child - $i*2$ index

right child - $i*2+1$ index

# Heap Types – Max and Min

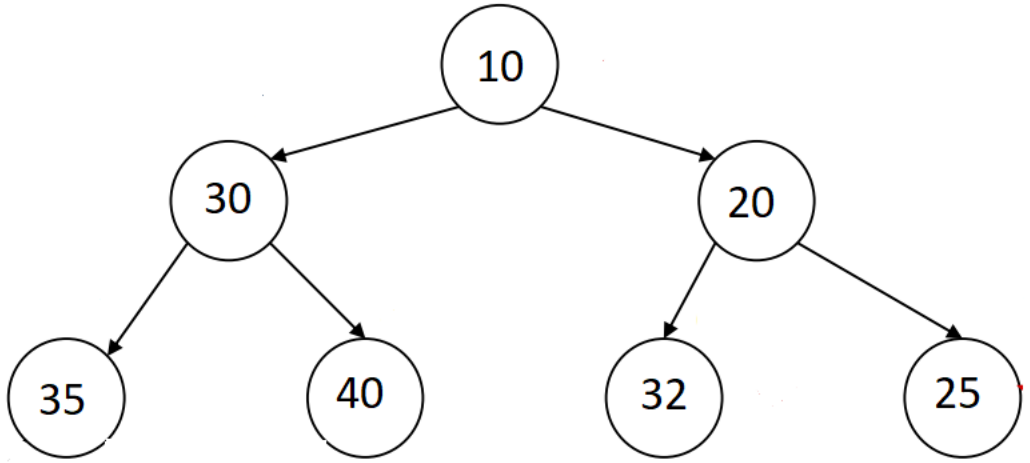
## Max Heap




| 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.

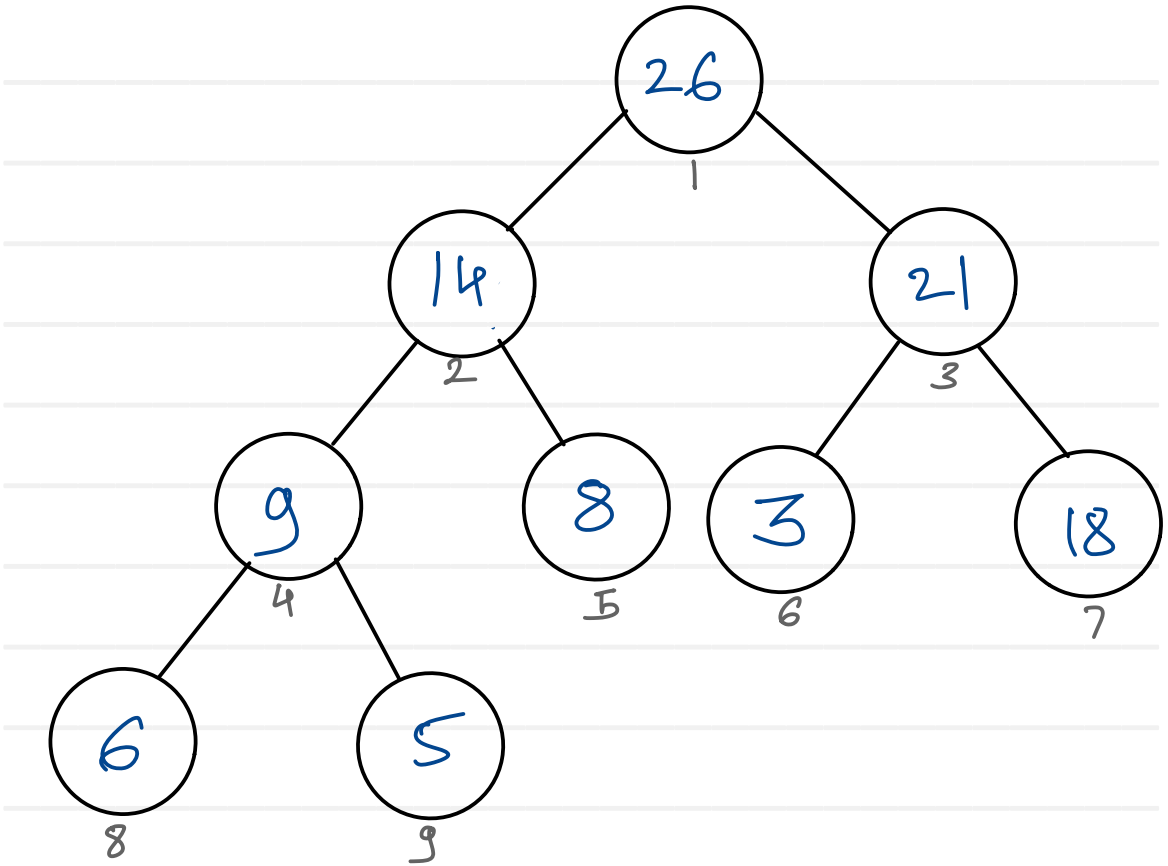
## Min Heap




| 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 )




| 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

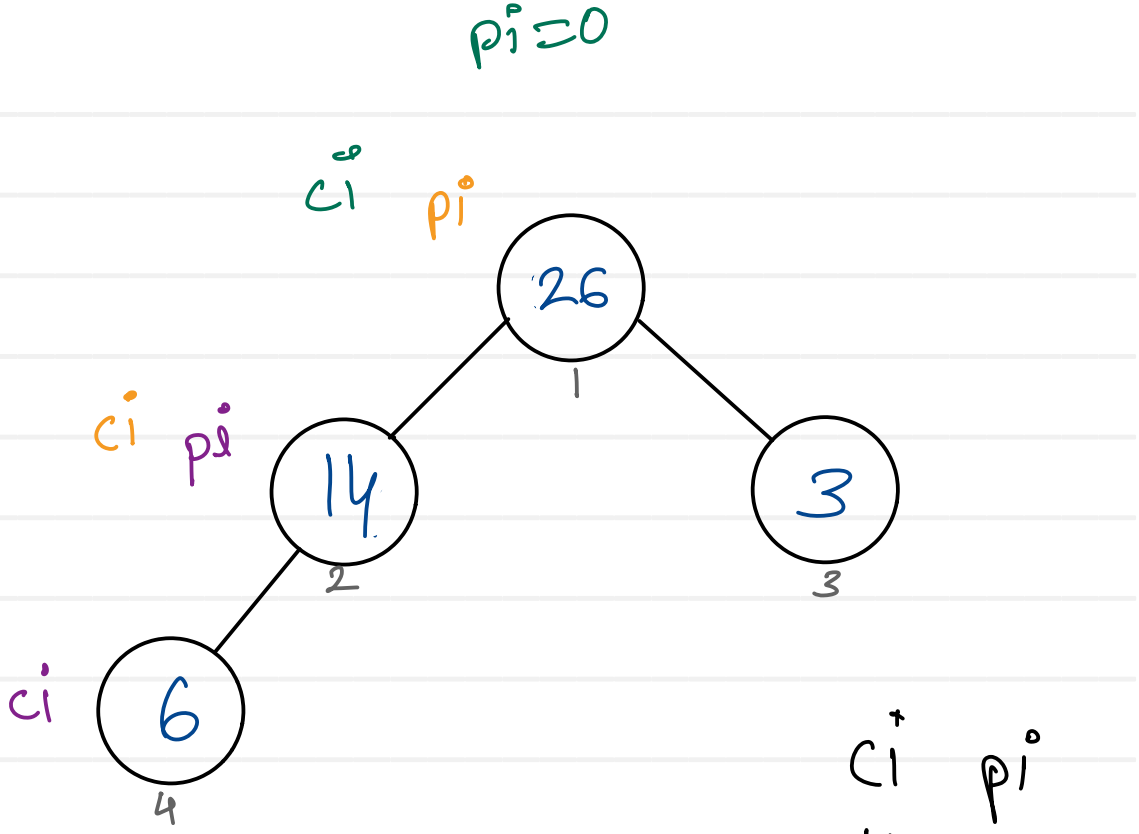
Algorithm :

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

- to add value, need to traverse from leaf position to root position

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

# Heap - Add

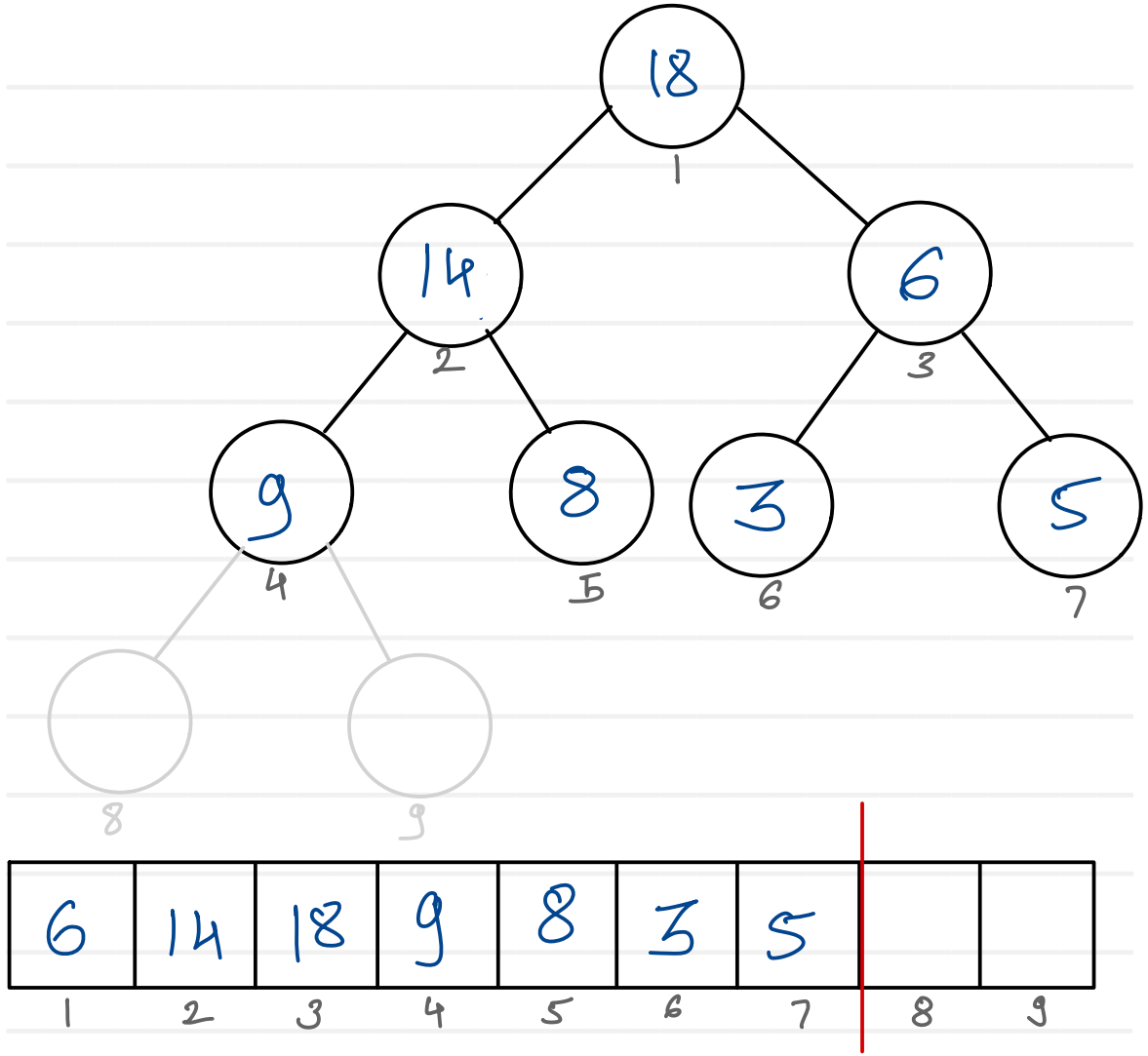



```
int arr[10];
int SIZE = 0;
```

| 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

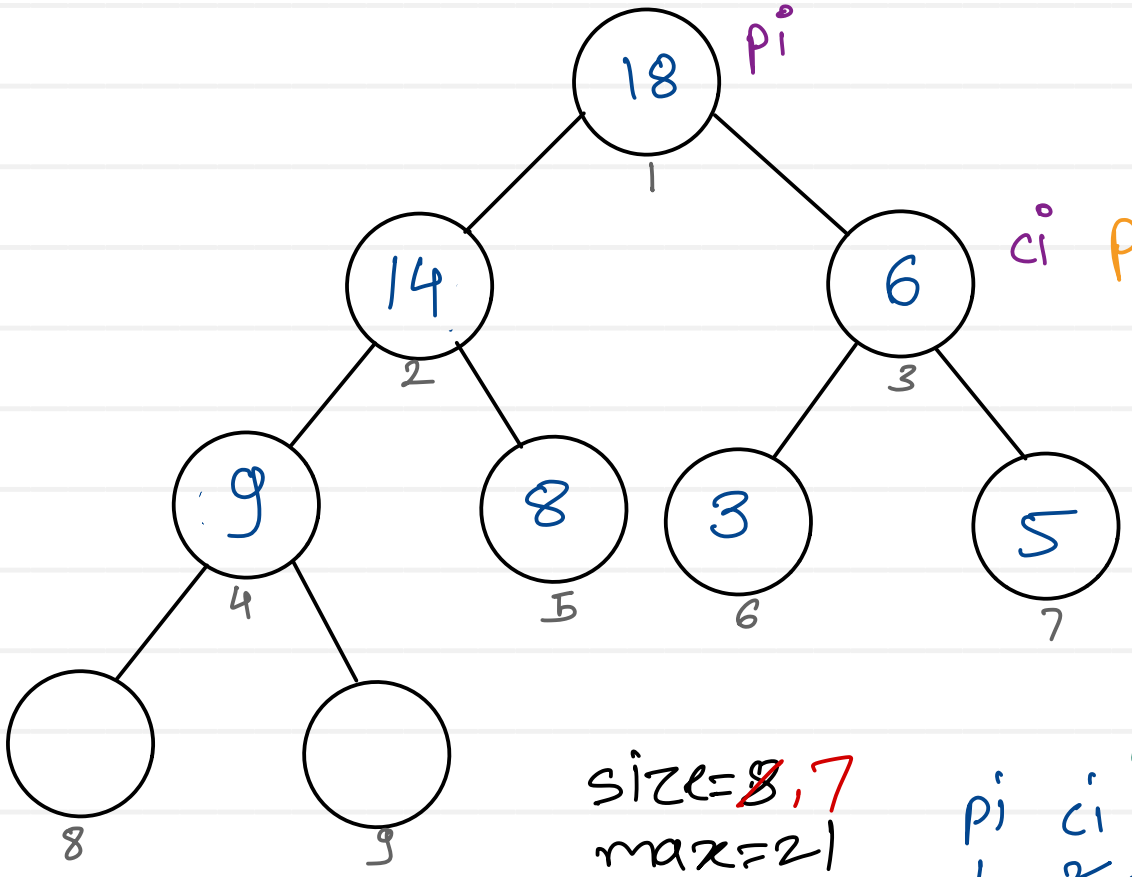
Algorithm:

1. place last element of heap at root position
2. adjust position of new root by comparing with all it's descendents one by one

- to adjust position need to traverse from root position to leap position

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

# Heap - Delete

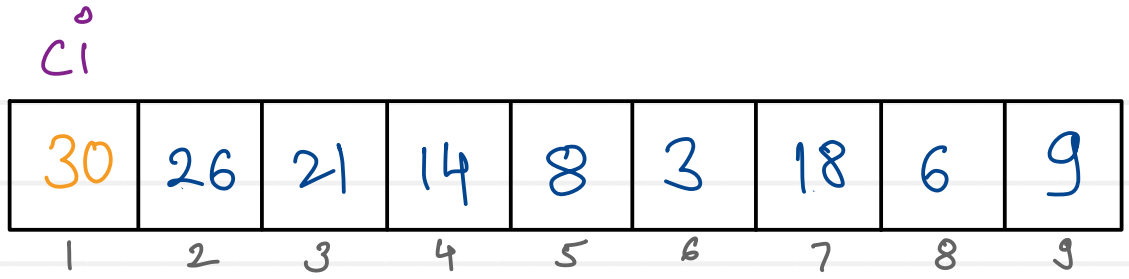



size=8,7
max=21

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;
```

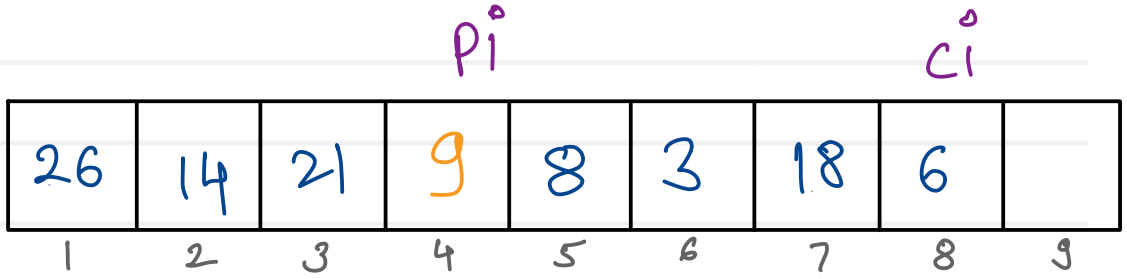



SIZE = 9

| ci | pi |
|---|---|
| 9 | 4 |
| 4 | 2 |
| 2 | 1 |
| 1 | 0 — stop |

ci = pi

pi = ci/2




max = 30

SIZE = 8

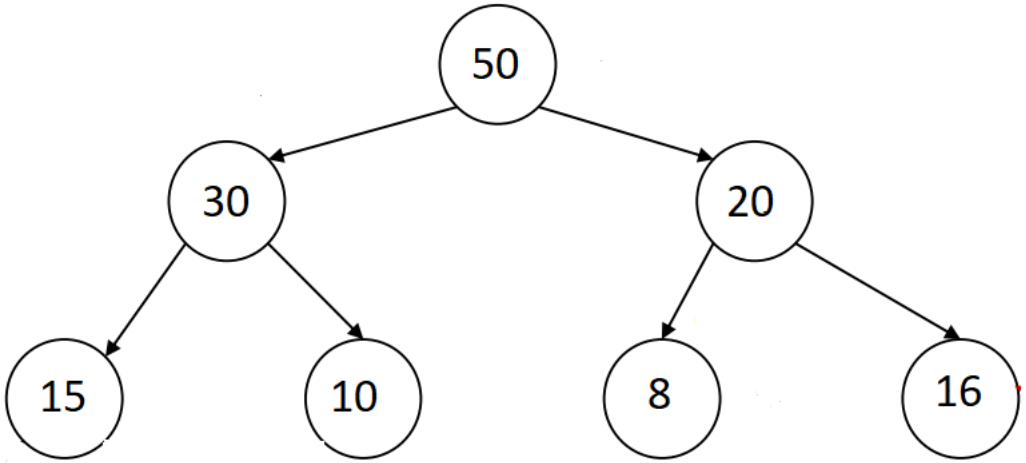
| pi | ci |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 4 | 8 |

pi = ci

ci = pi * 2

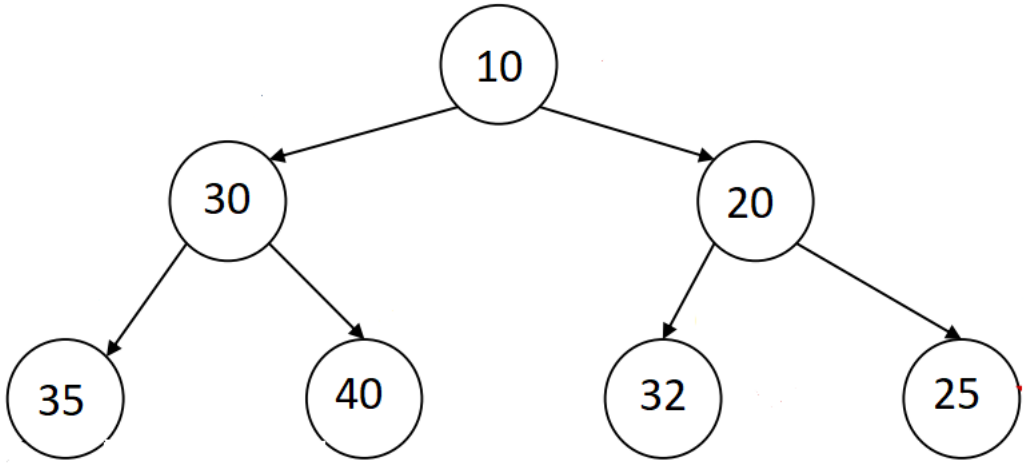
# Priority Queues

**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


# Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com