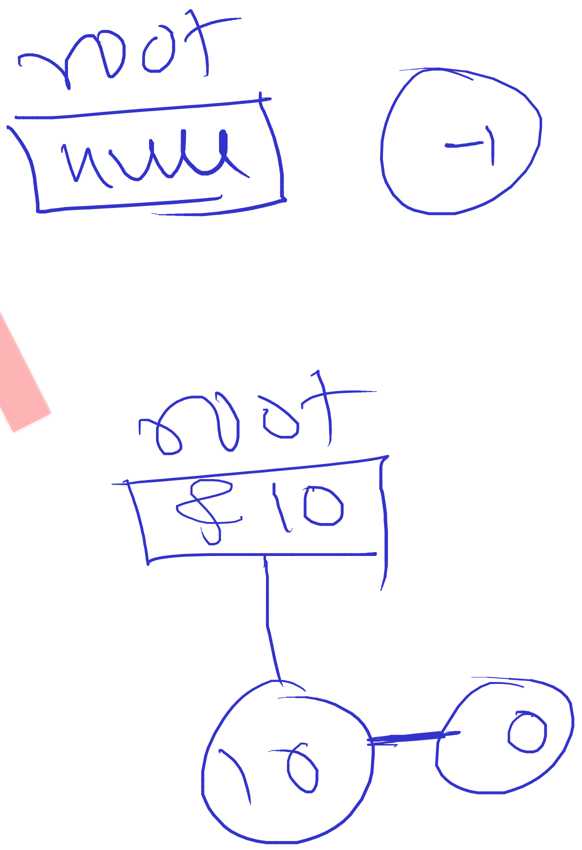
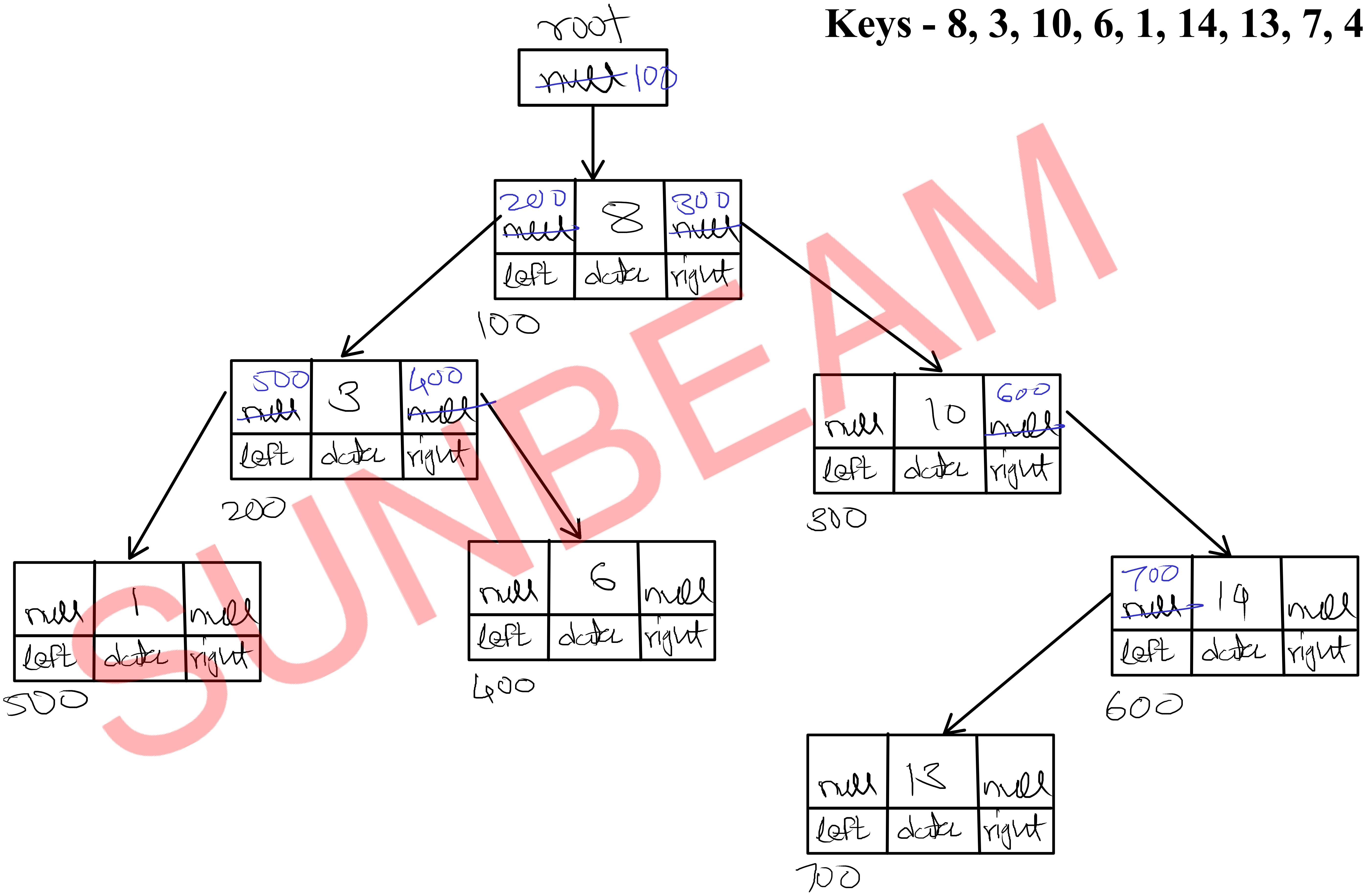
BST - Add Node

```
//1. create a newnode with given data
//2. if tree is empty
    // add newnode into root itself
//3. if tree is not empty
    //3.1 create trav pointer and start at root
    //3.2 if value is less than current node data
        //3.2.1 if left of current node is empty
            // add newnode into left of current node
        //3.2.2 if left of current node is not empty
            // go to left of current node
    //3.3 if value is greater than current node data
        //3.3.1 if right of current node is empty
            // add newnode into right of current node
        //3.3.2 if right of current node is not empty
            // go to right of current node
    //3.4 repeat spep 3.2 and 3.3 till node is not added into BST
```

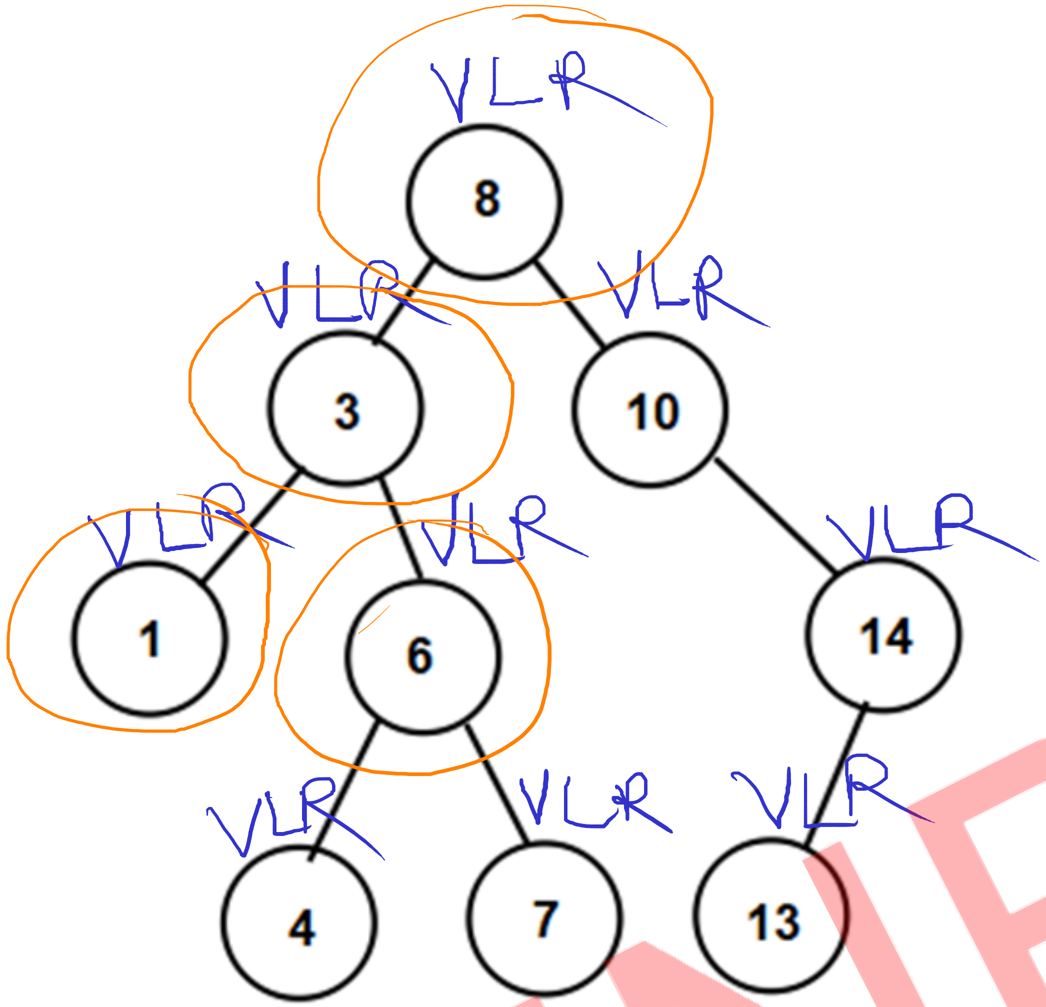


BST - Add Node

Keys - 8, 3, 10, 6, 1, 14, 13, 7, 4



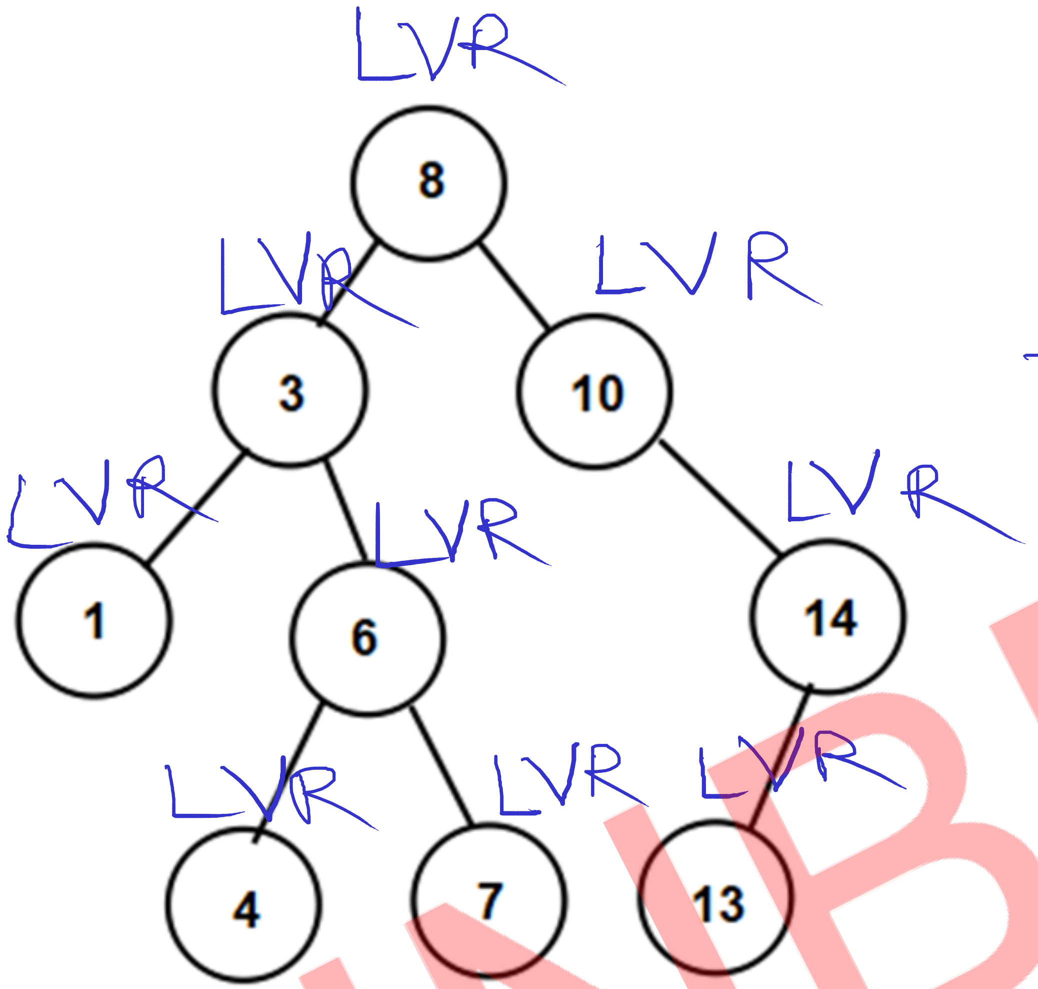
BST - Preorder Traversal



VLR

Traversal : 8, 3, 1, 6, 4, 7, 10, 14, 13

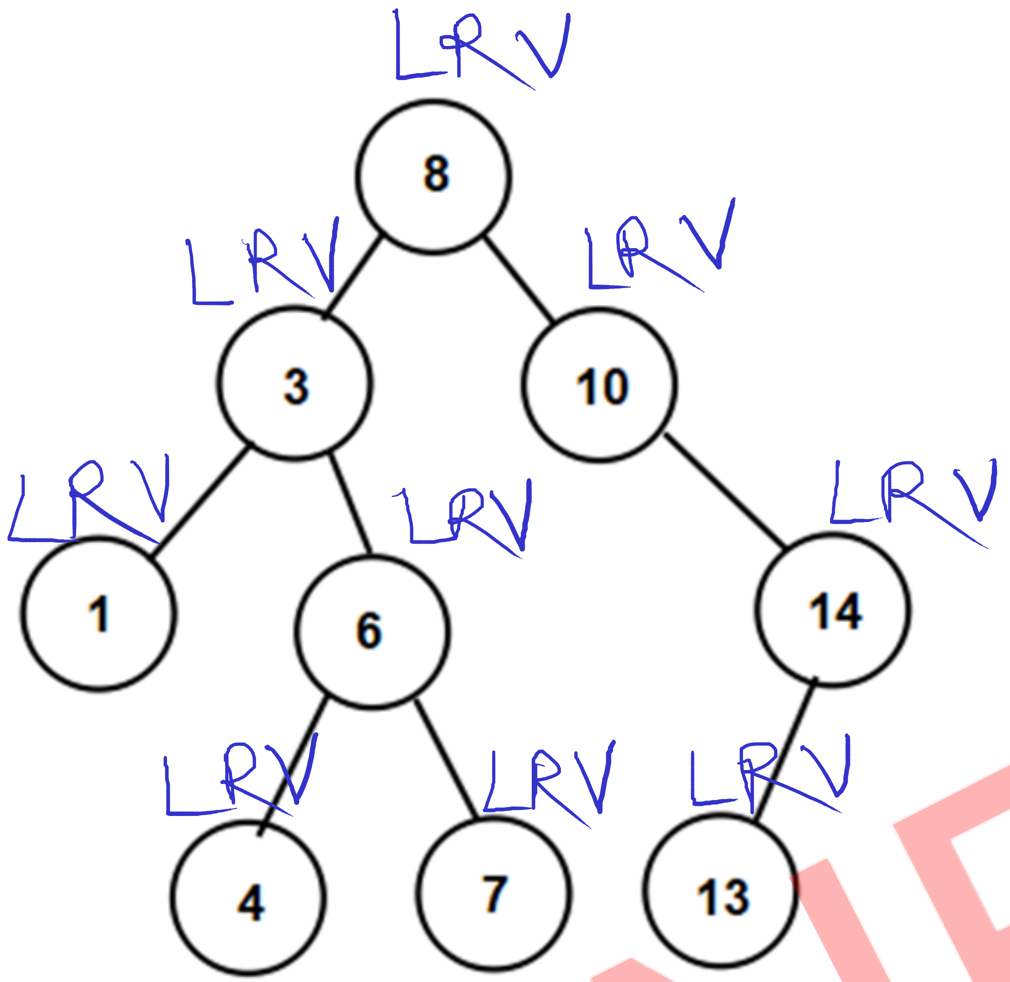
BST - Inorder Traversal



LVR

Traversal: 1, 3, 4, 6, 7, 8, 10, 13, 14

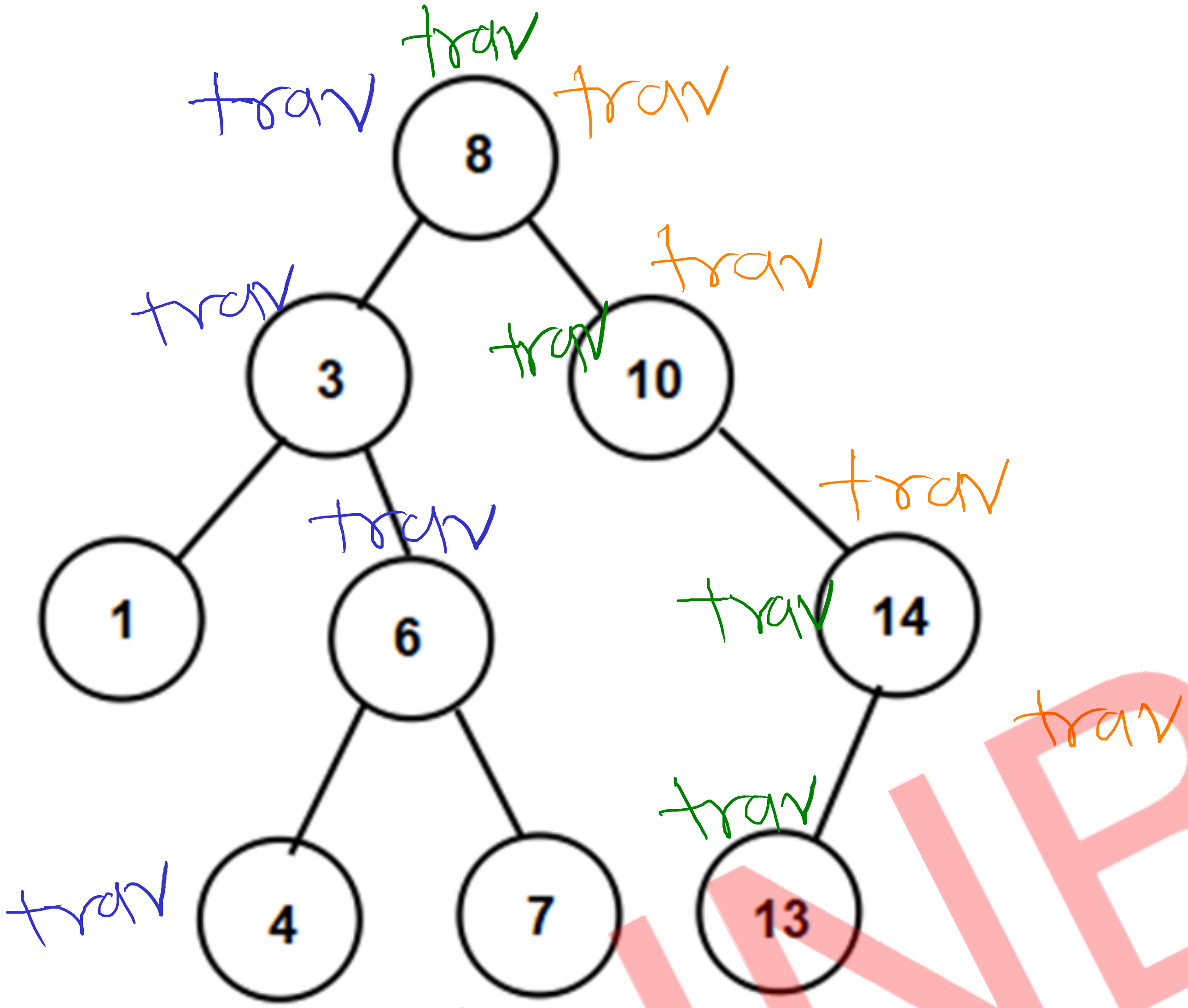
BST - Postorder Traversal



LRV

Traversal : 1, 4, 7, 6, 3, 13, 14, 10, 8

BST - Binary Search



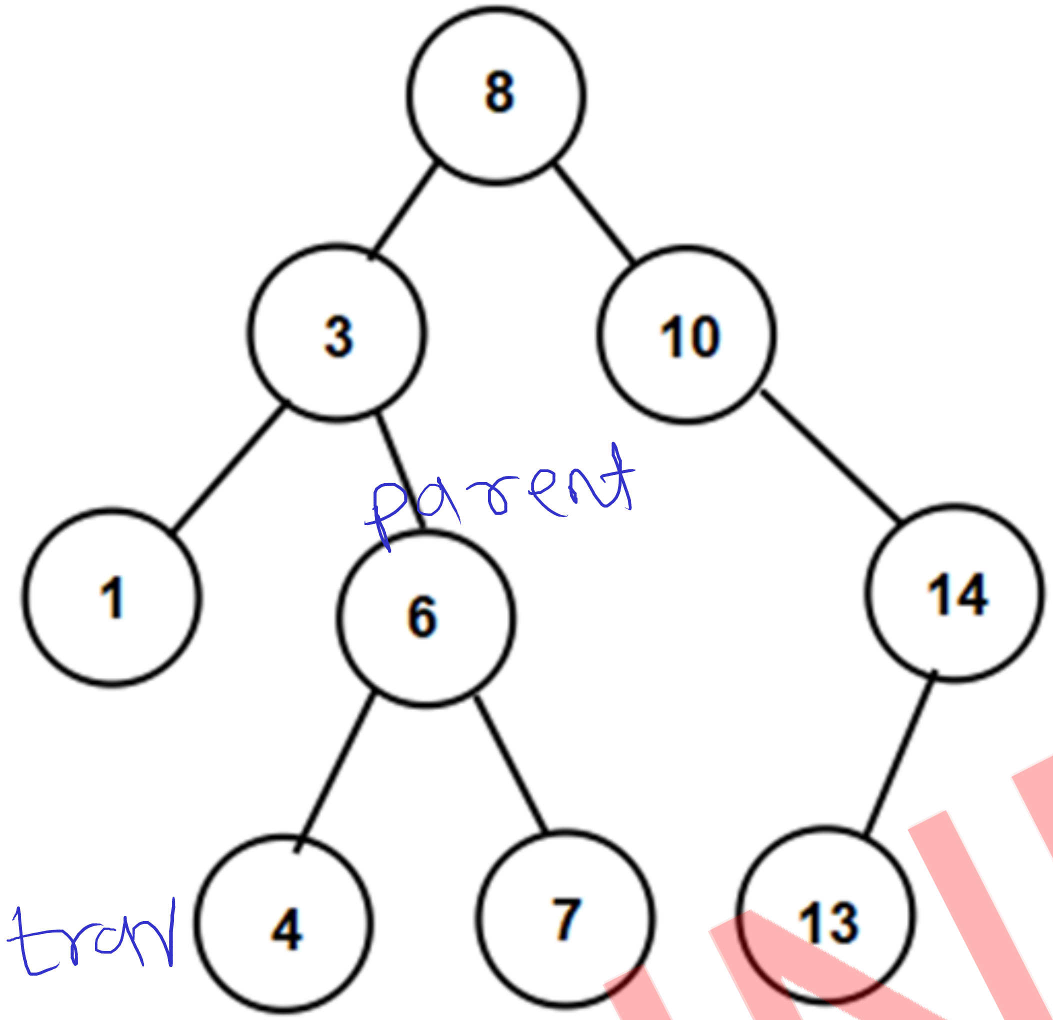
```
//1. start from root
//2. if key is equal to current data
        //return current node
//3. if key is less than current data
        // search key into left of current node
//4. if key is greater than current data
        // search key into right of current node
//5. repeat step 2 to 4 till leaf nodes
```

key = 4 - key found

key = 15 - key not found

key = 13 - key found

BST - Binary Search with Parent



key=4

trav	parent
&8	null
&3	&8
&6	&3
&4	&6

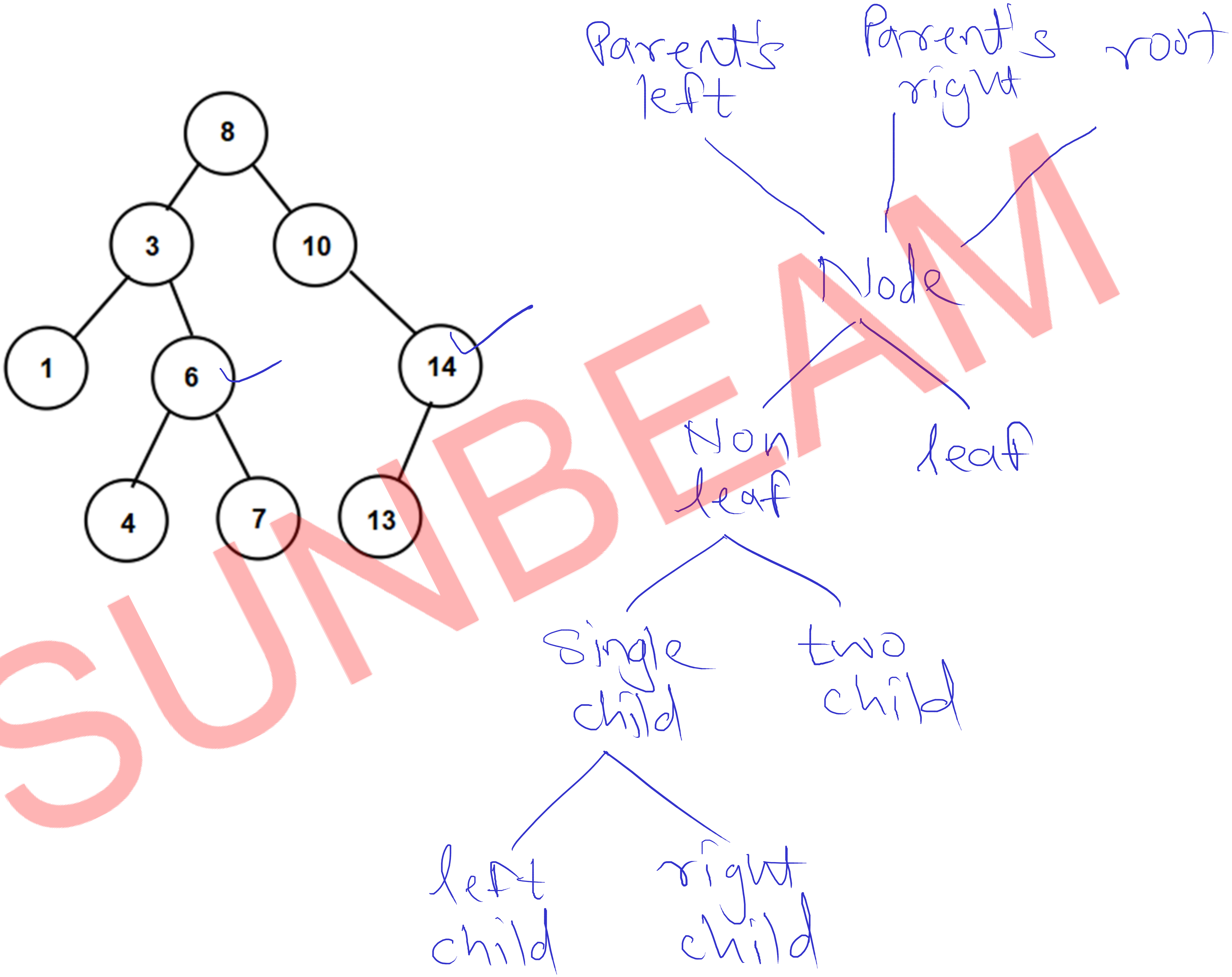
key=15

trav	parent
&8	null
&10	&8
&14	&10
null	&14

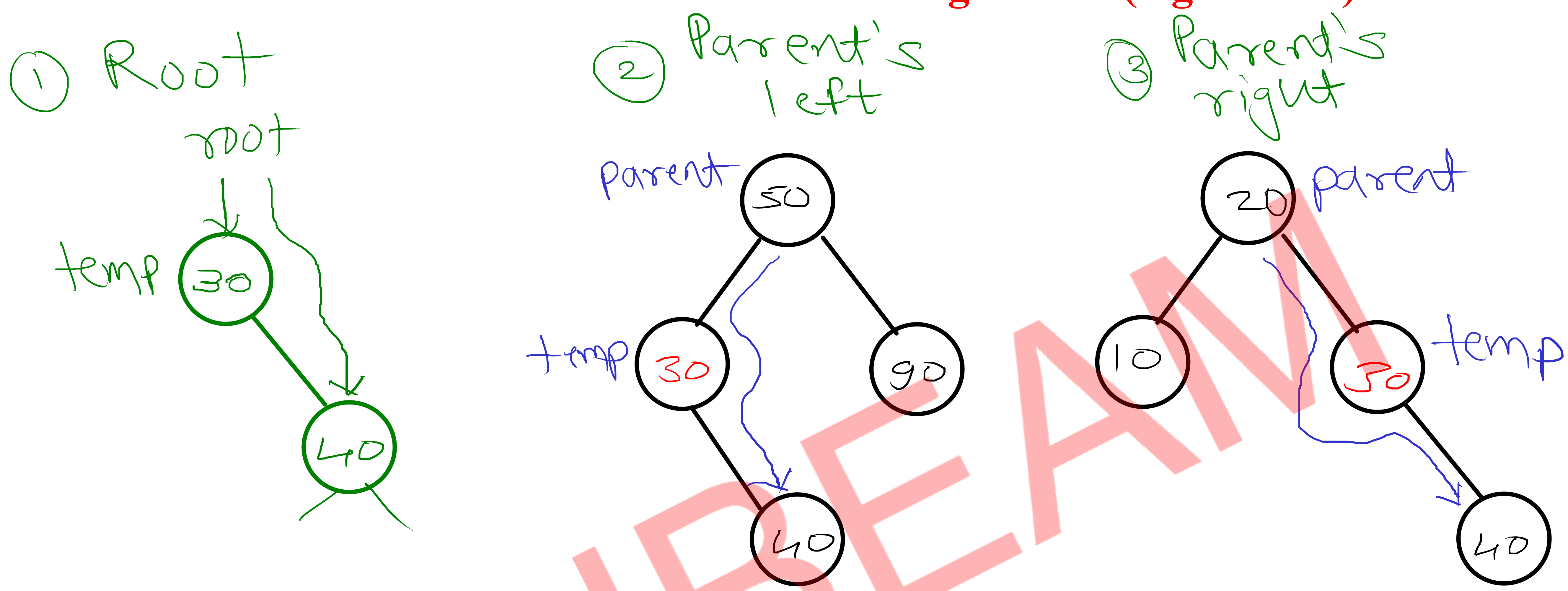
key=8

trav	parent
&8	null

BST - Delete Node



BST - Delete node which has single child (right child)



```
if(temp.left == null){
    if(temp == root)
        root = temp.right;
    else if(temp == parent.left)
        parent.left = temp.right;
    else if(temp == parent.right)
        parent.right = temp.right;
}
```

① ② ③

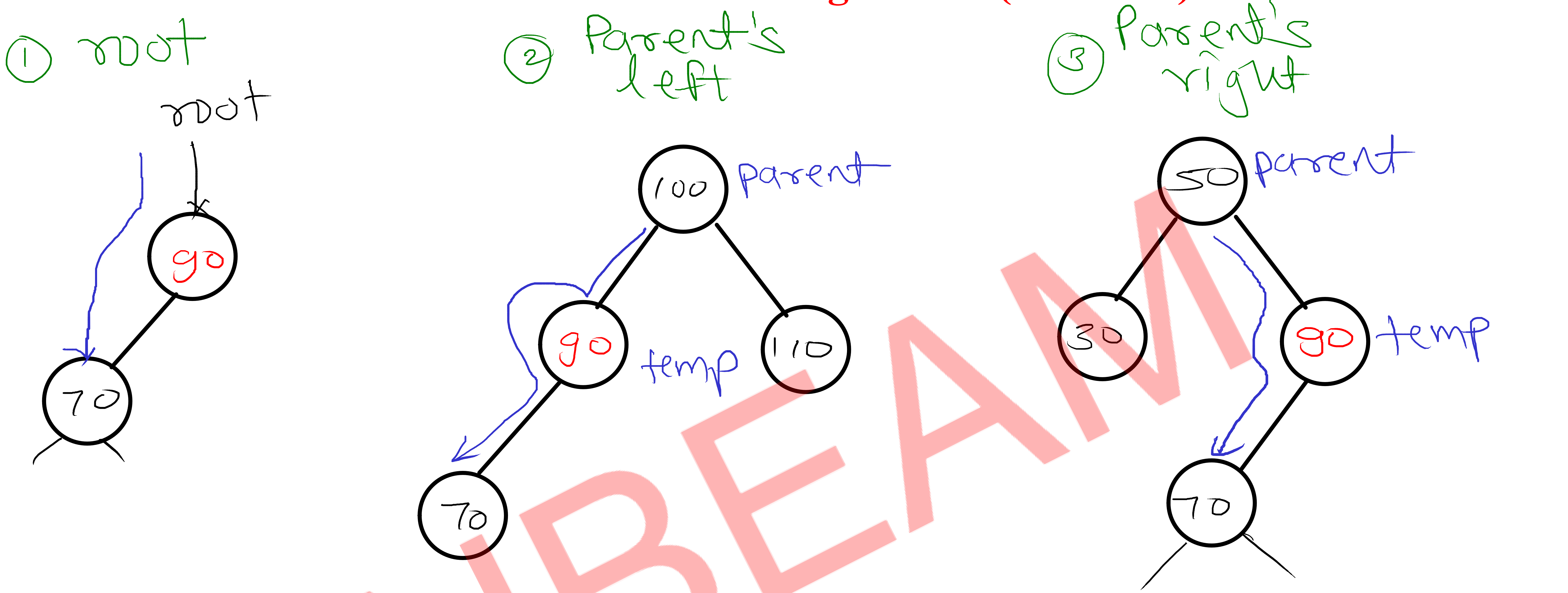
→ singl child (left missing)

→ root node

→ parent's left child

→ Parent's right child

BST - Delete node which has single child (left child)



```
if(temp.right == null){
        if(temp == root)
①               root = temp.left;
        else if(temp == parent.left)
②               parent.left = temp.left;
        else if(temp == parent.right)
③               parent.right = temp.left;
}
```

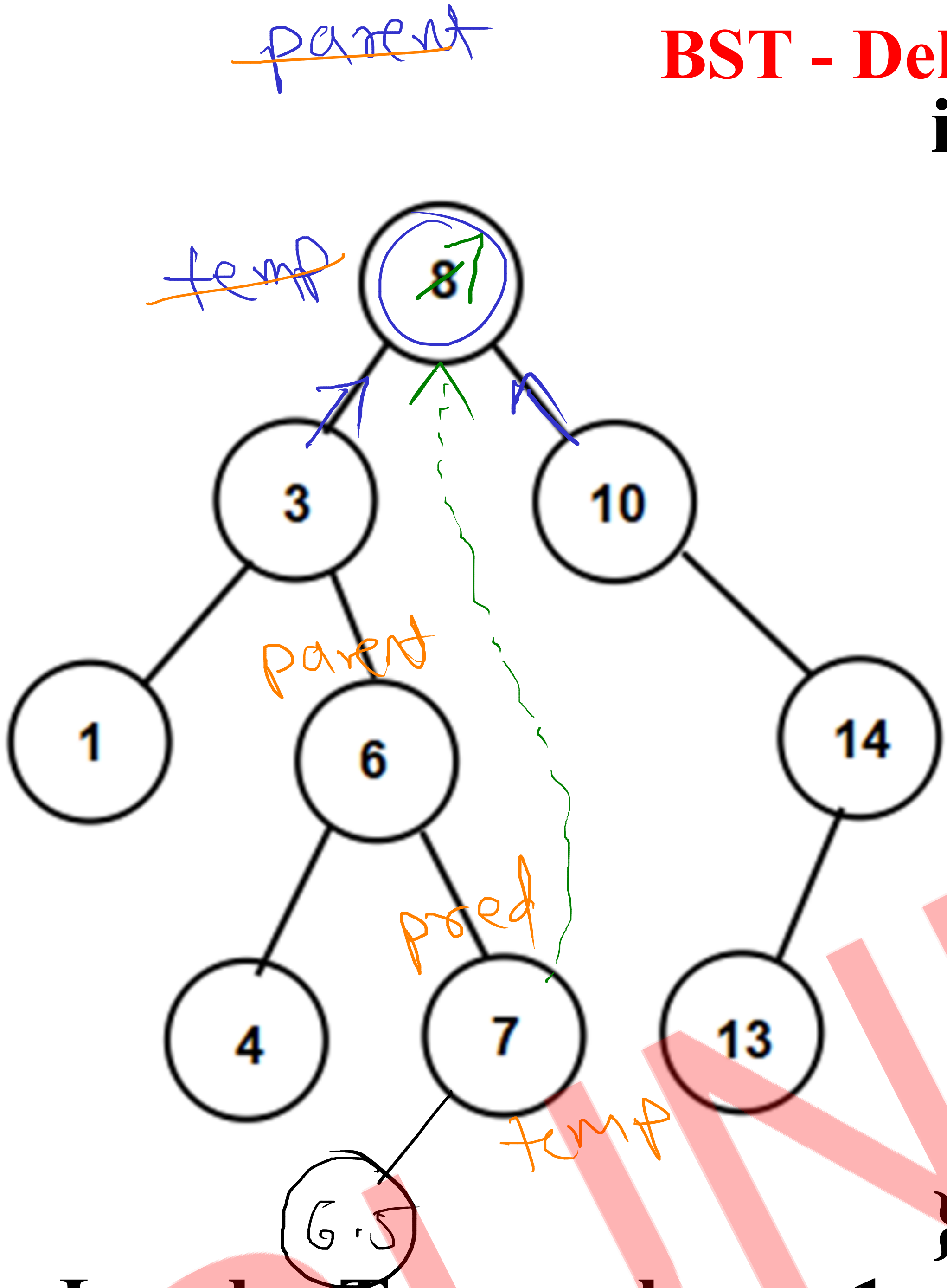
→ singl child (right missing)

→ root node

→ parent's left child

→ Parent's right child

BST - Delete node which has two childs



```
if(temp.left != null && temp.right != null){
        //1. find predecessor of temp.data
        Node pred = temp.left;
        parent = temp;
        while(pred.right != null){
                parent = pred;
                pred = pred.right
        }
        //2. replace temp's data by predecessor's data
        temp.data = pred.data;
        //3. delete predecessor
        temp = pred;
}
```

Inorder Traversal : 1 3 4 6 7 8 10 13 14

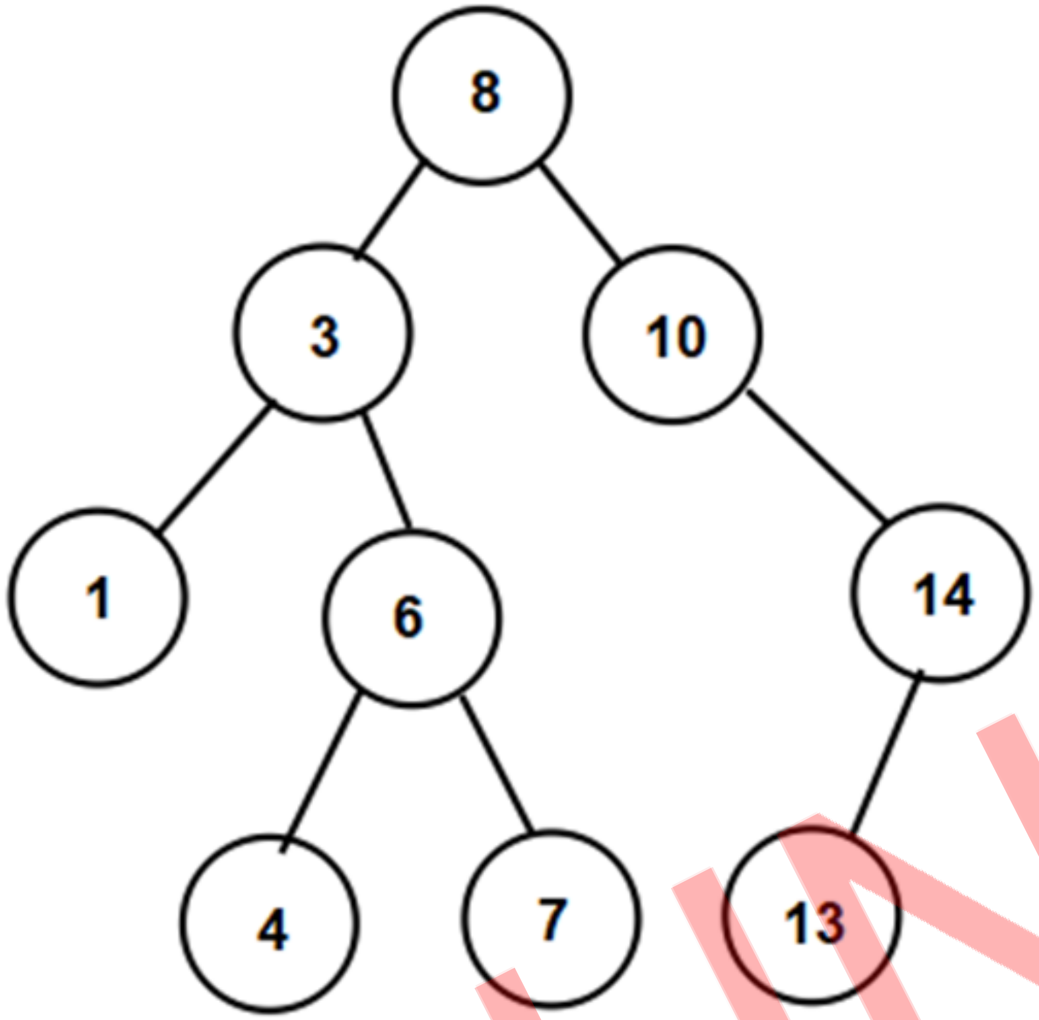
inorder predecessor

left extreme right

inorder successor

right extreme left

BST - DFS (Depth First Search)



stack

~~13~~

~~14~~

~~4~~

~~7~~

~~1~~

~~6~~

~~3~~

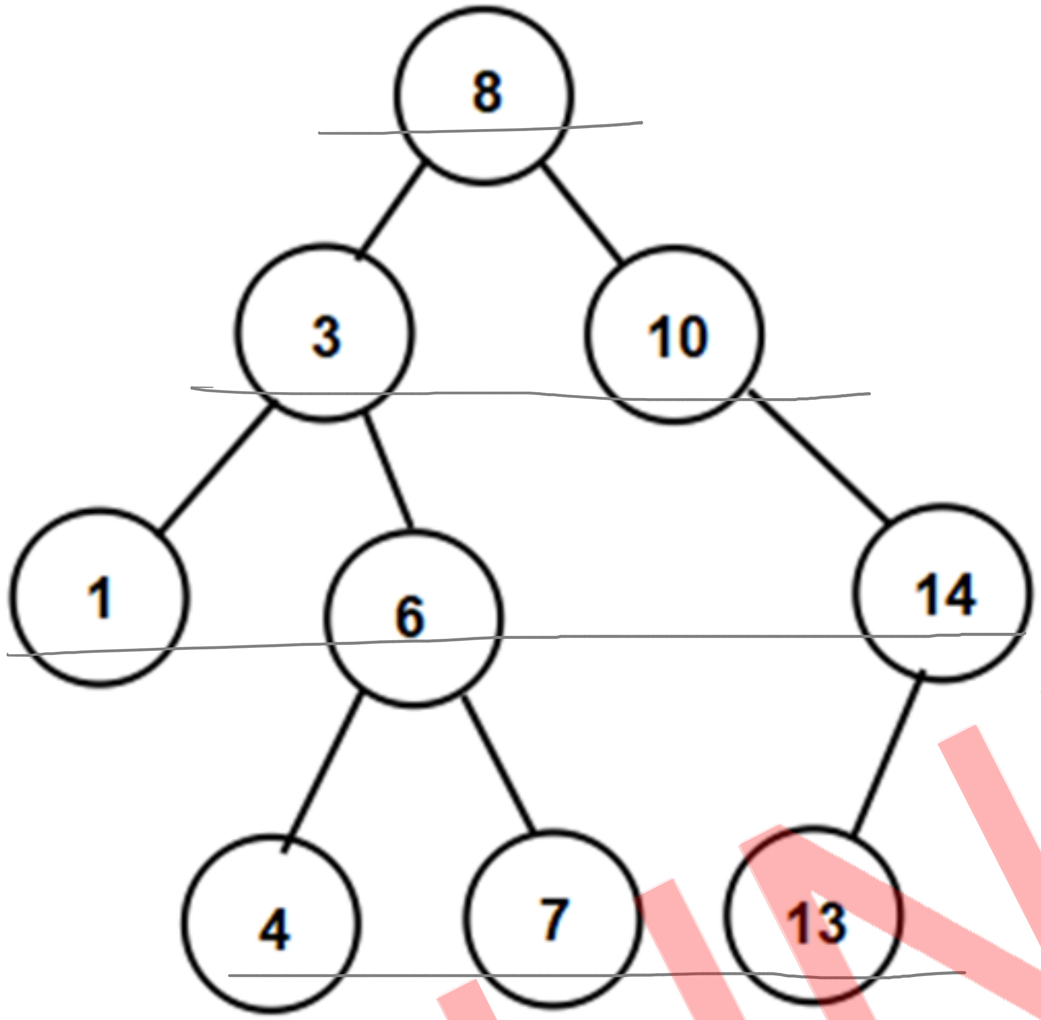
~~10~~

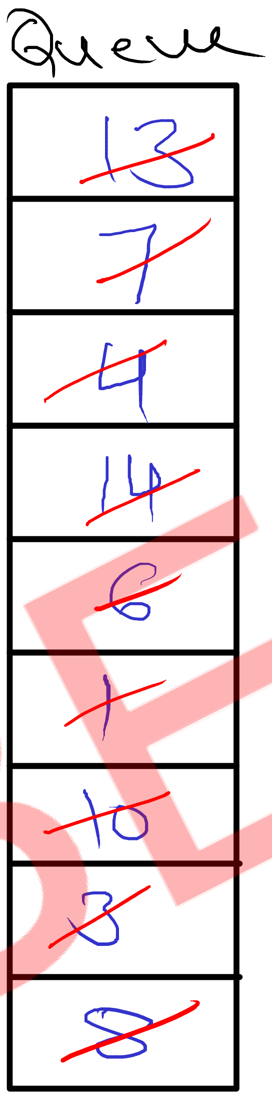
~~8~~

```
//1. push root on stack
//2. pop one node from stack
//3. visit(print) node
//4. if right exist, push it on stack
//5. if left exist, push it on stack
//6. while stack is not empty
    //repeat ste 2 to 5
```

Traversal : 8, 3, 1, 6, 4, 7, 10, 14, 13

BST - BFS (Bredth First Search)



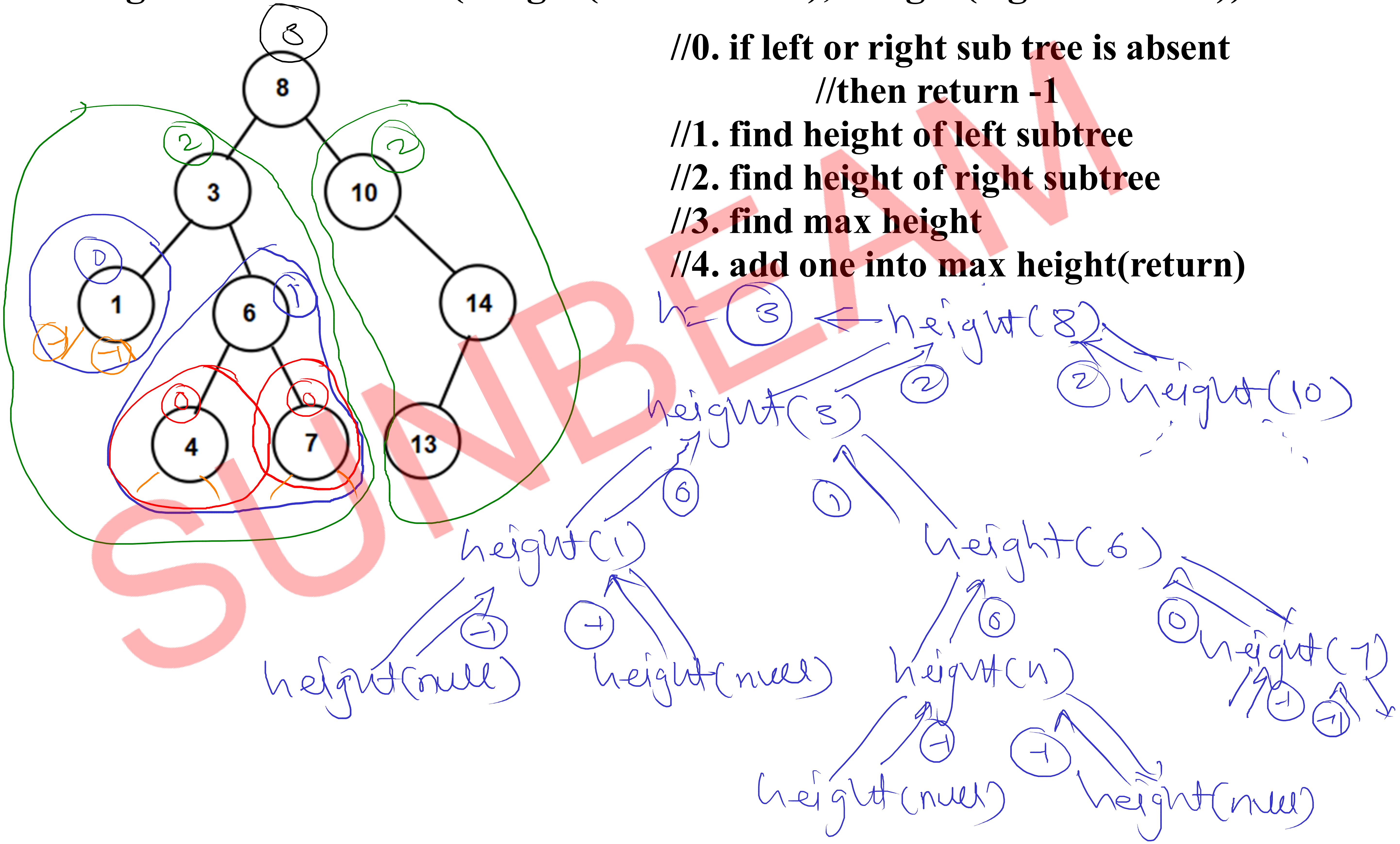


```
//1. push root on queue
//2. pop one node from queue
//3. visit(print) node
//4. if left exist, push it on queue
//5. if right exist, push it on queue
//6. while queue is not empty
    //repeat ste 2 to 5
```

Traversed : 8, 3, 10, 1, 6, 14, 4, 7, 13

BST - Height

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



//0. if left or right sub tree is absent
//then return -1
//1. find height of left subtree
//2. find height of right subtree
//3. find max height
//4. add one into max height(return)

BST - Time complexity of operations

No. of nodes = n

height = h

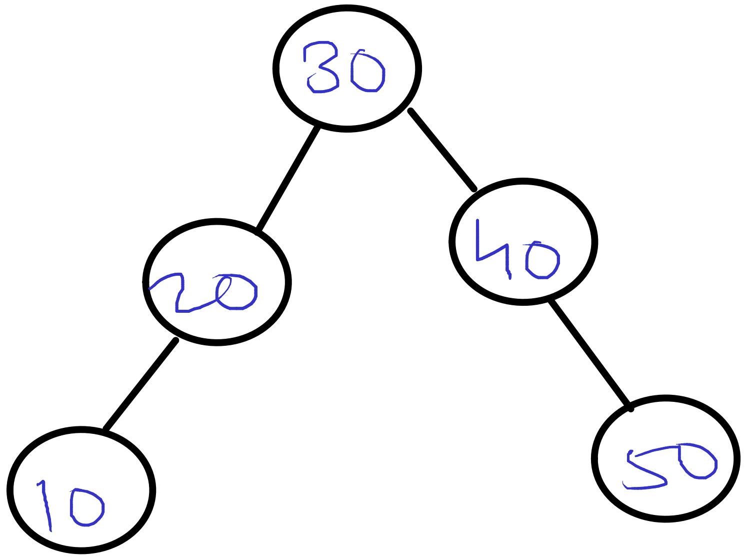
$$\boxed{n = 2^{h+1} - 1}$$

h	n
0	1
1	3
2	7
3	15

ADD	:	$T(h) = O(h)$	$T(n) = O(\log n)$
Search	:	$T(h) = O(h)$	$T(n) = O(\log n)$
DELETE	:	$T(h) = O(h)$	$T(n) = O(\log n)$
Traversel	:		$T(n) = O(n)$

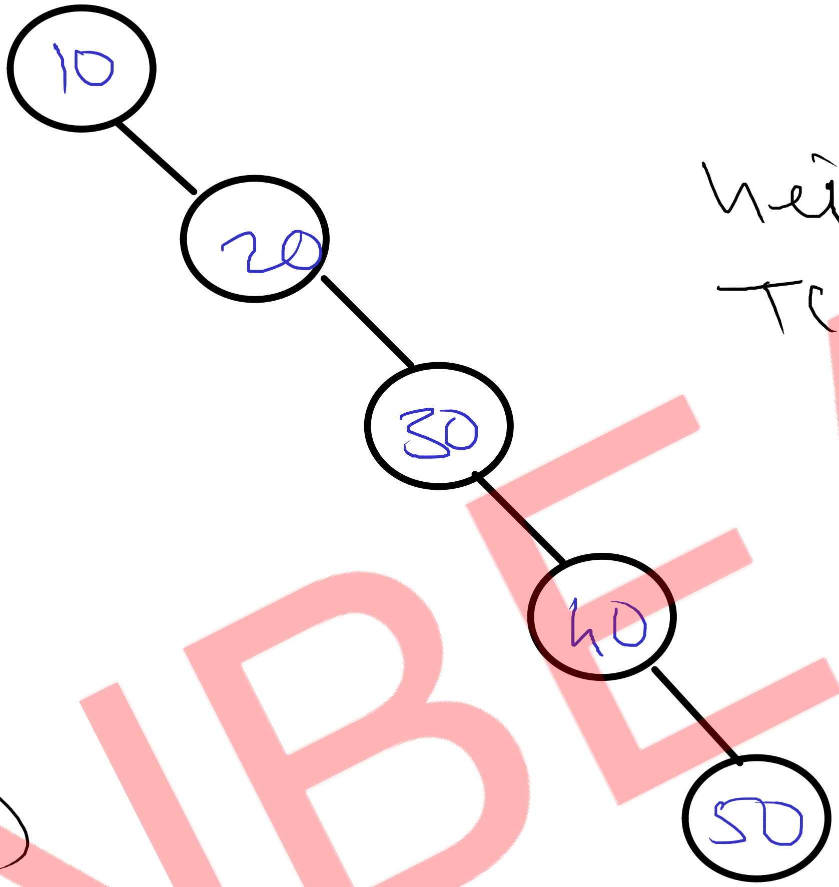
Skewed BST

Keys : 30, 40, 20, 50, 10



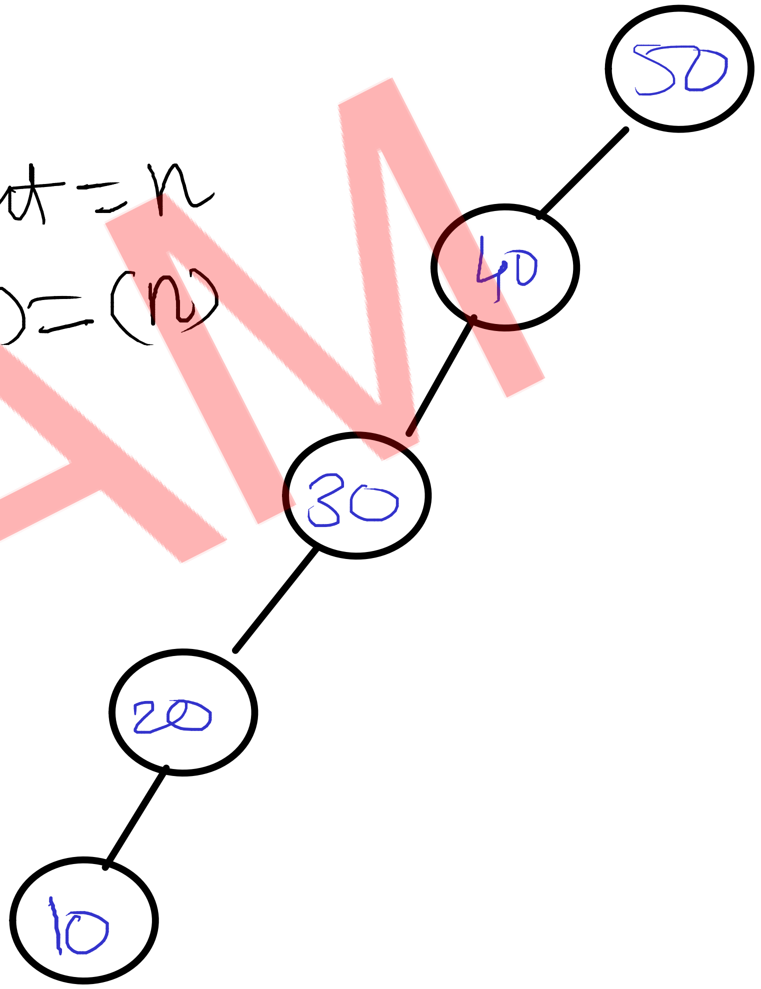
height = log n
T(n) = O(log n)

Keys : 10, 20, 30, 40, 50



height = n
T(n) = (n)

Key : 50, 40, 30, 20, 10



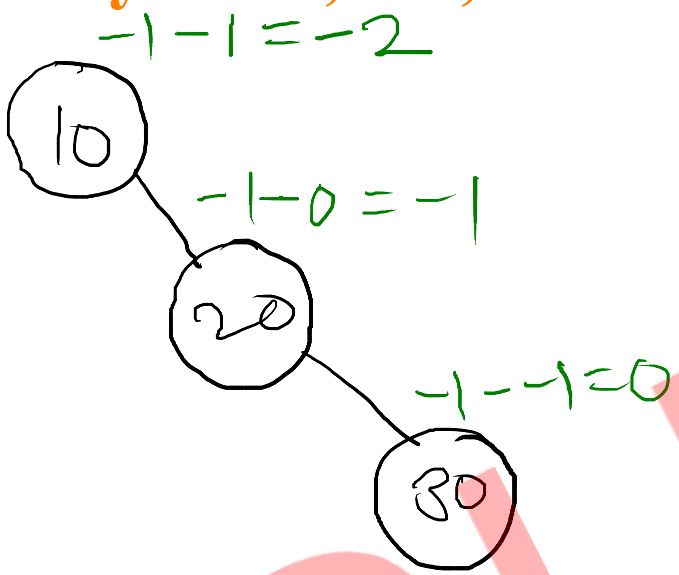
- tree is growing in only one direction either left or right
- if tree is growing in only left direction : left skewed BST
- if tree is growing in only right direction : right skewed BST

Balanced BST

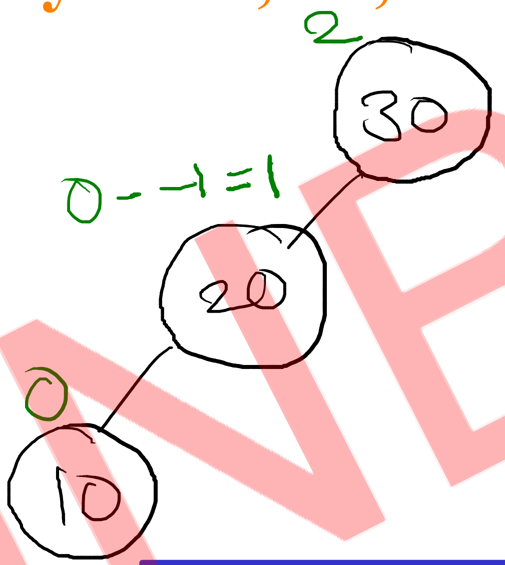
Balance Factor = height(left sub tree) - height(right sub tree)

- tres is balanced if balance factor of all the nodes is either -1, 0 or +1
- balance factor = {-1, 0, +1}

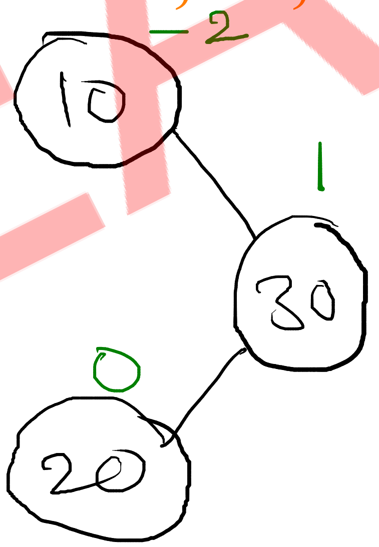
Keys : 10, 20 ,30



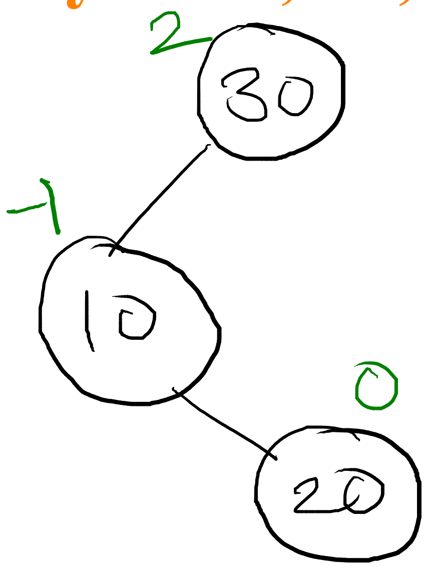
Keys : 30, 20, 10



Keys : 10, 30, 20

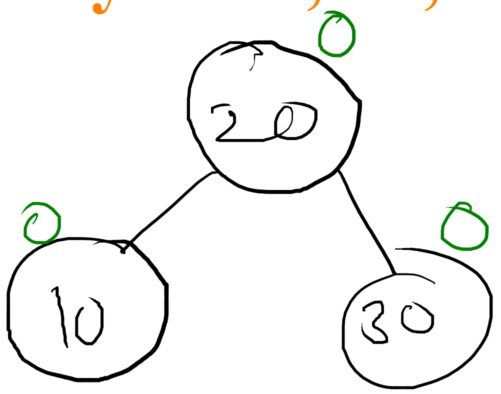


Keys : 30, 10, 20



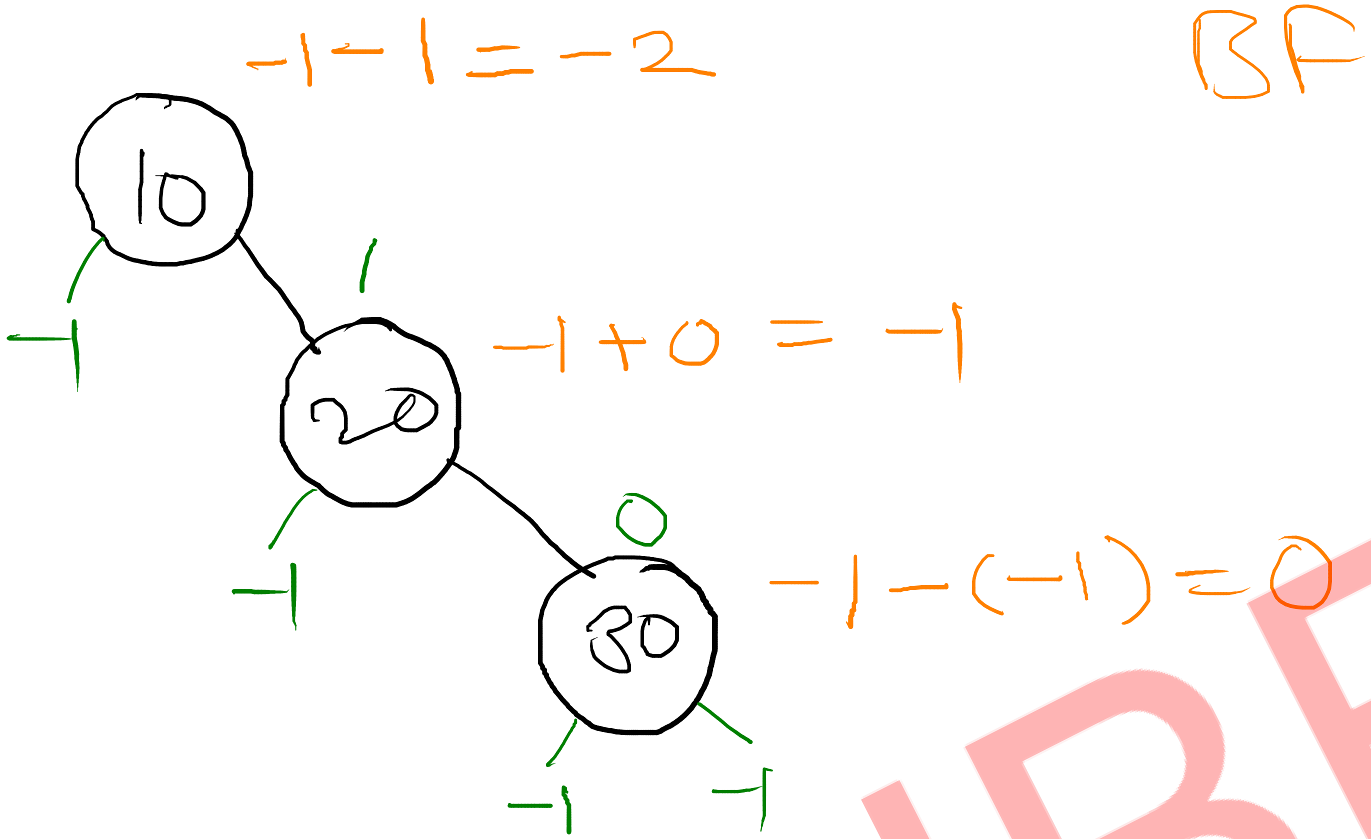
Keys : 20, 10, 30

Keys : 20, 30, 10



Balanced BST

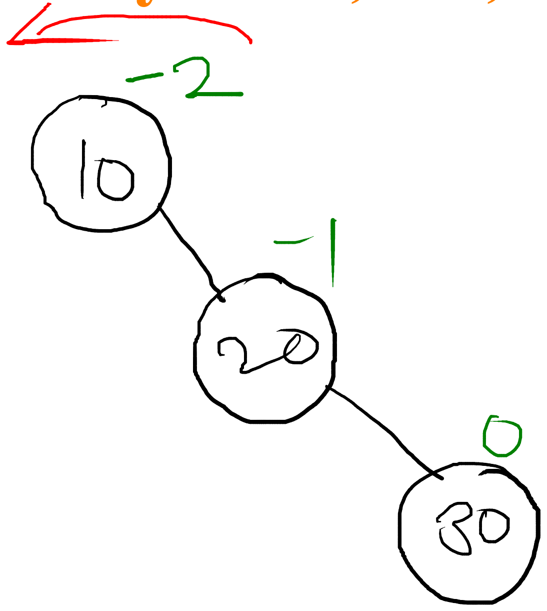
BF = HL - HR



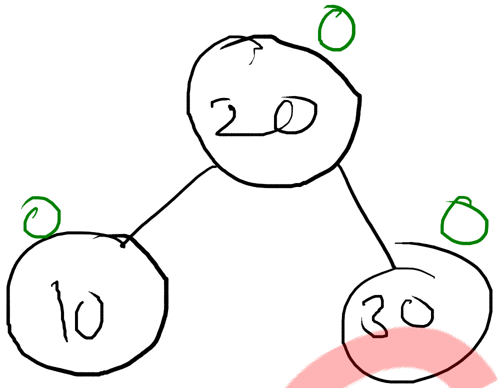
Rotations

RR Imbalance

Keys : 10, 20 ,30

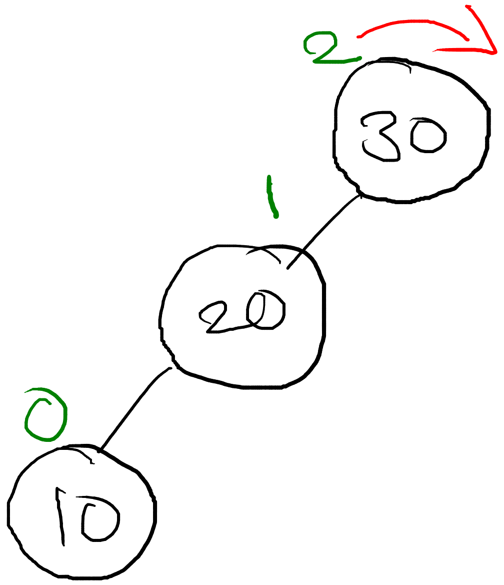


Left Rotation

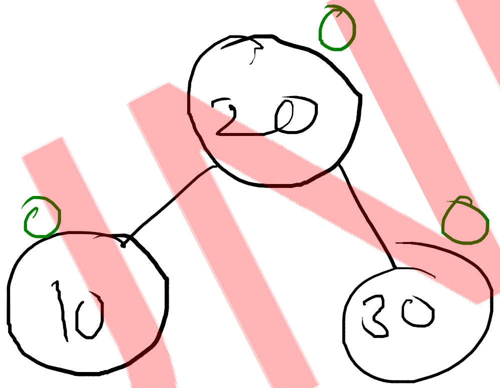


LL Imbalance

Keys : 30, 20, 10



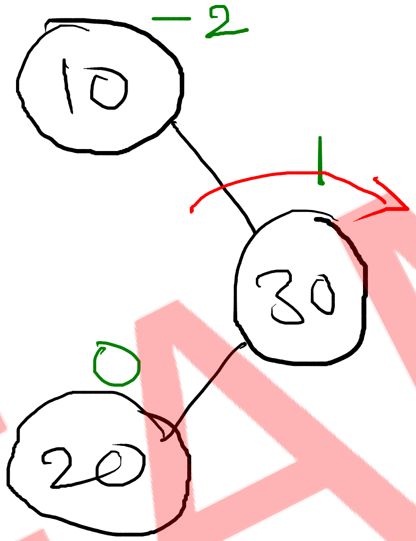
Right Rotation



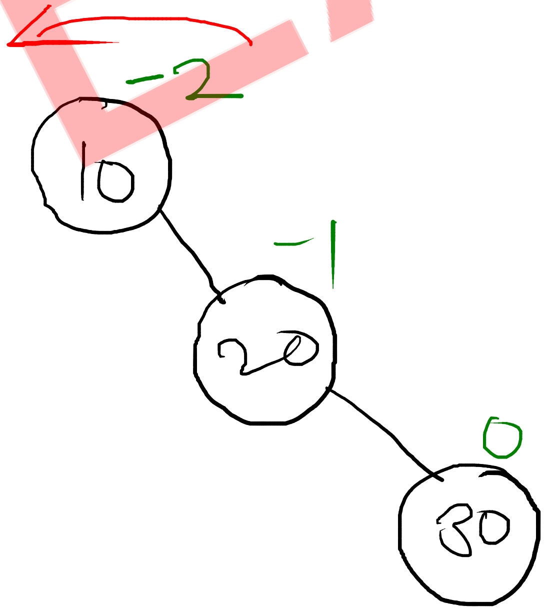
Single Rotation

RL Imbalance

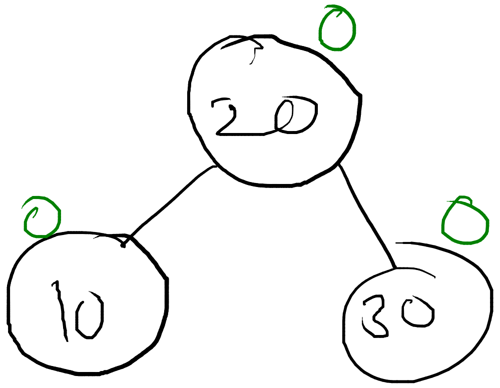
Keys : 10, 30, 20



Right Rotation

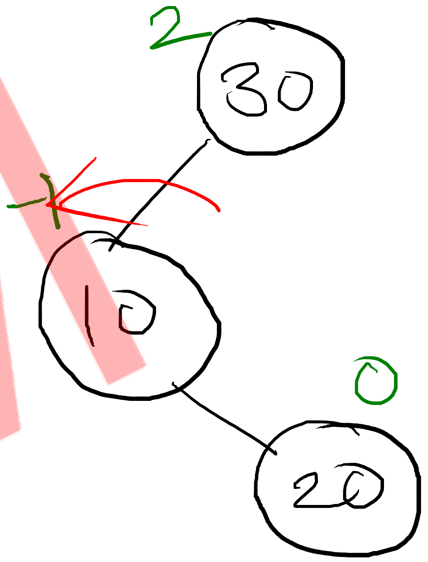


Left Rotation

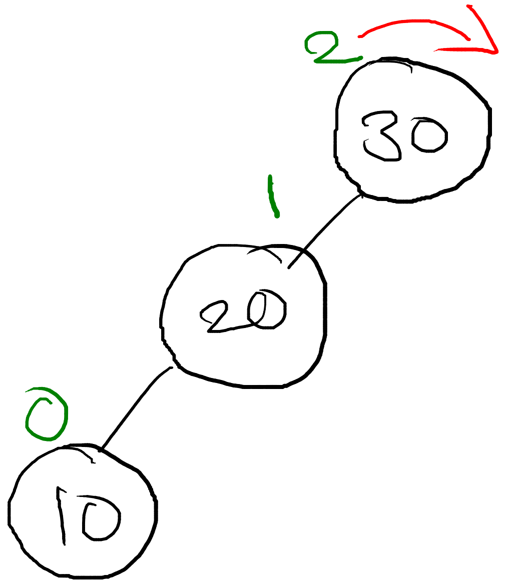


LR Imbalance

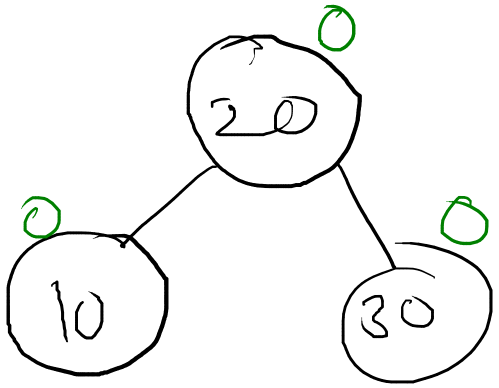
Keys : 30, 10, 20



Left Rotation



Right Rotation



Double rotations