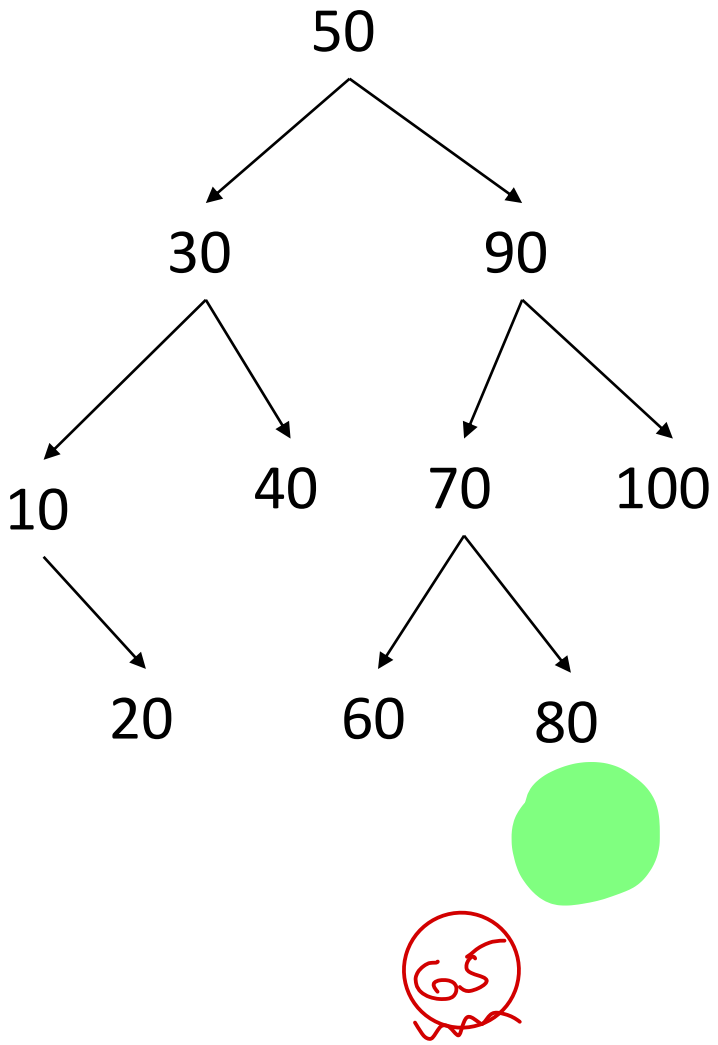
Data Structure & Algorithms

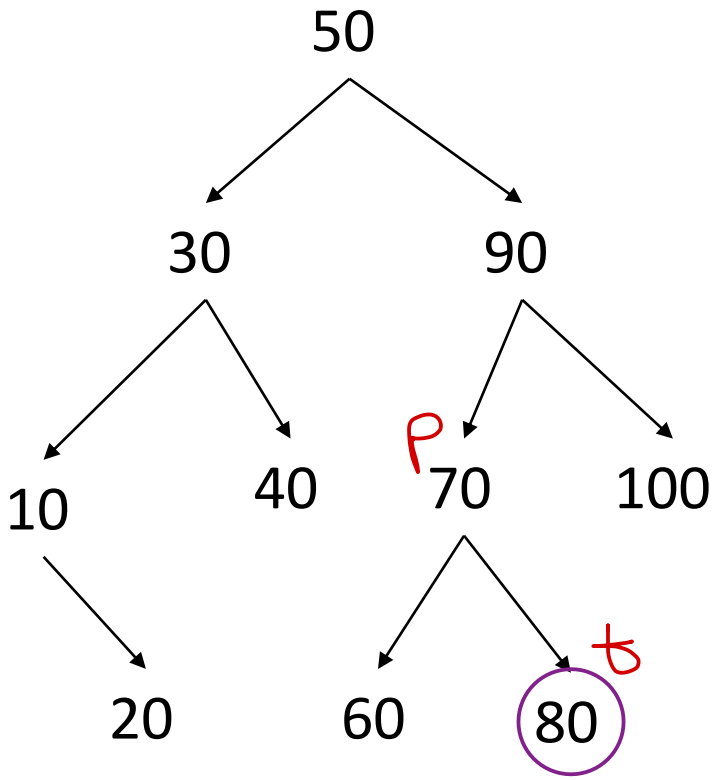
Nilesh Ghule

BST – search




```
trav = root;
while(trav != null)
{   if( key == trav.data)
        return trav;
    if( key < trav.data)
        trav = trav.left;
    else
        trav = trav.right;
}
return null;
```

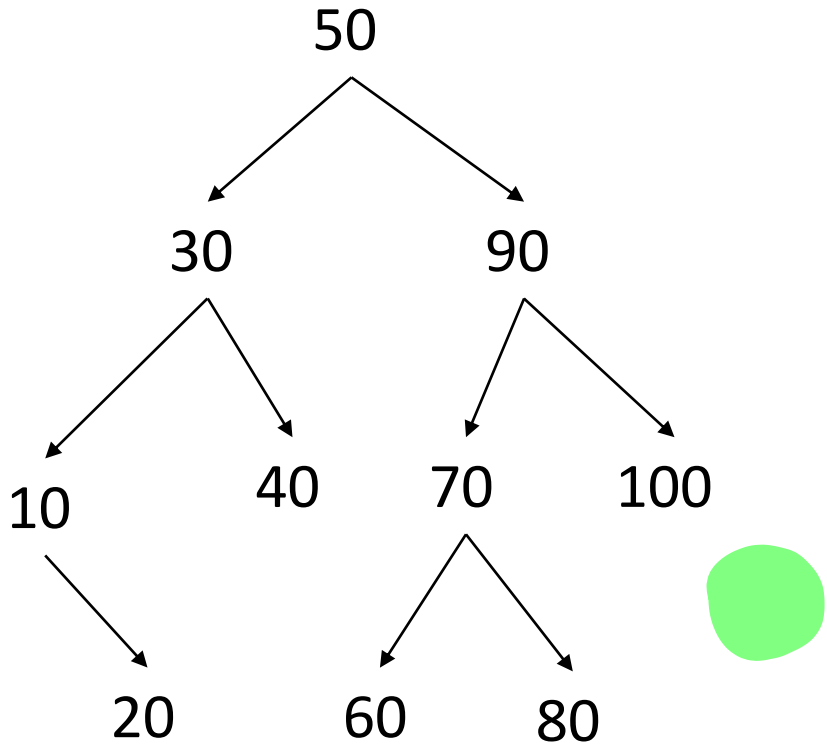
BST – search – with parent




```
Parent = null;
trav = root;
while(trav != null)
{   if( key == trav.data)
        return {trav, parent};
    Parent = trav;
    if( key < trav.data)
        trav = trav.left;
    else
        trav = trav.right;
}
Parent = null;
return {null, null};
```

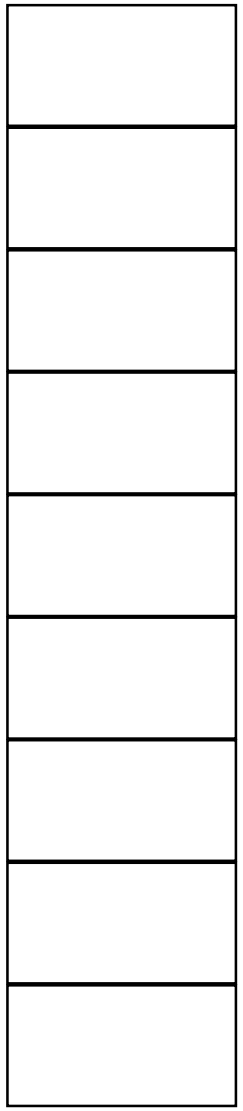
BST – InOrder - non recursive

L - P - R



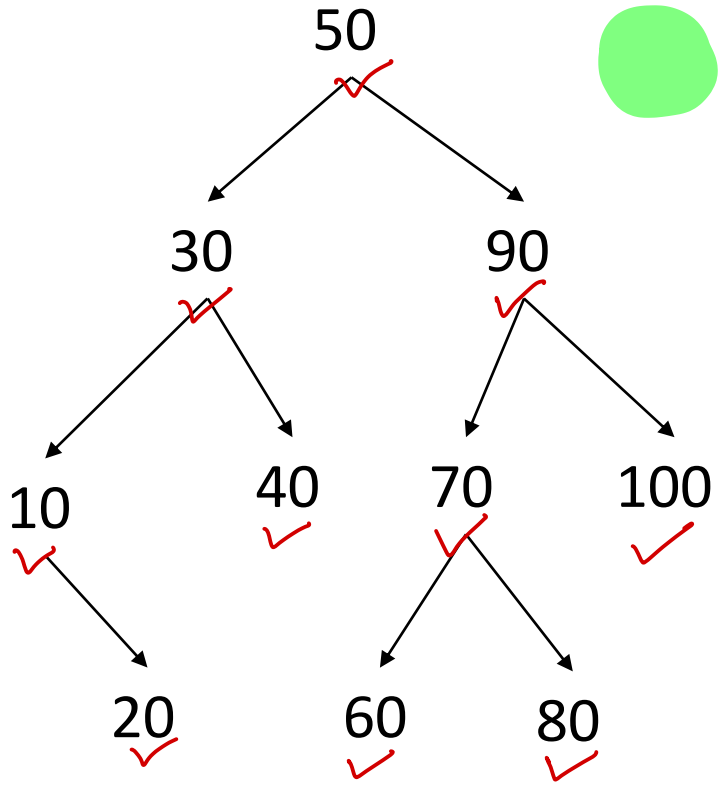
10 20 30 40 50

60 70 80 90 100



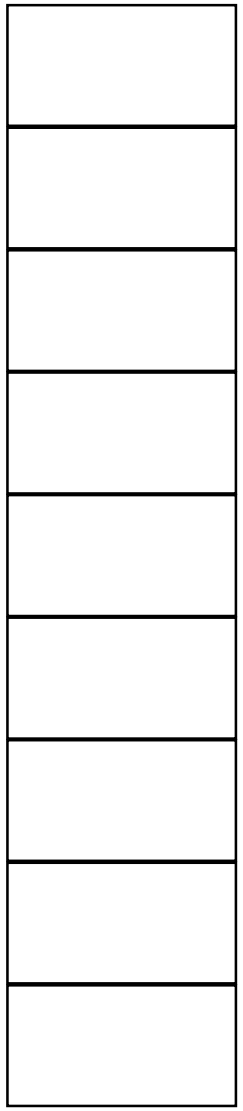
```
trav = root;
while(trav != null || !s.isEmpty()) {
    while(trav != null){
        s.push(trav)
        trav = trav.left;
    }
    if(!s.isEmpty()){
        trav = s.pop();
        print(trav.data);
        trav = trav.right;
    }
}
```

BST –PostOrder - non recursive L R P



20 10 40 30 60

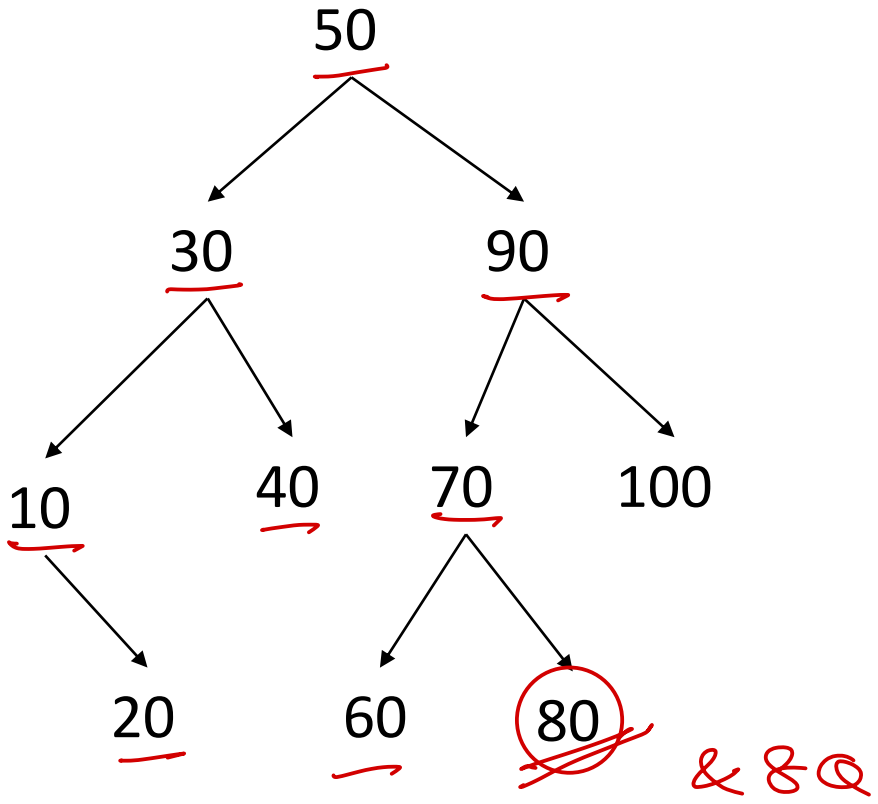
80 70 100 90 50



```
trav = root;
while(trav != null || !s.isEmpty()) {
    while(trav != null) {
        s.push(trav);
        trav = trav.left;
    }
    if(!s.isEmpty()) {
        trav = s.pop();
        if(trav.right == null ||
           trav.right.visited) {
            print(trav.data);
            trav.visited = true;
            trav = null;
        }
        else {
            s.push(trav);
            trav = trav.right;
        }
    }
}
```

BST – Non-Recursive Algorithm – DFS

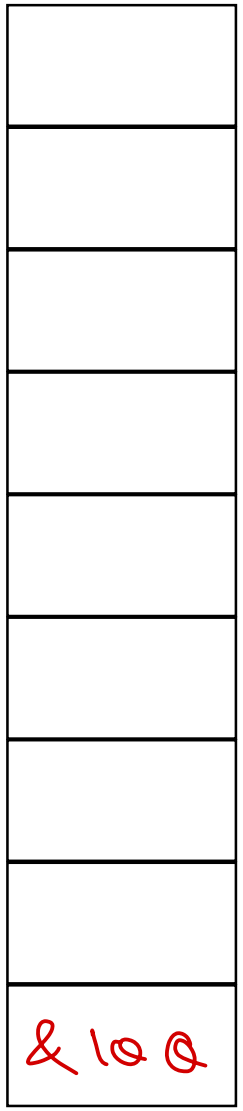
→ Depth First Search



&80

&50 &30 &10 &20

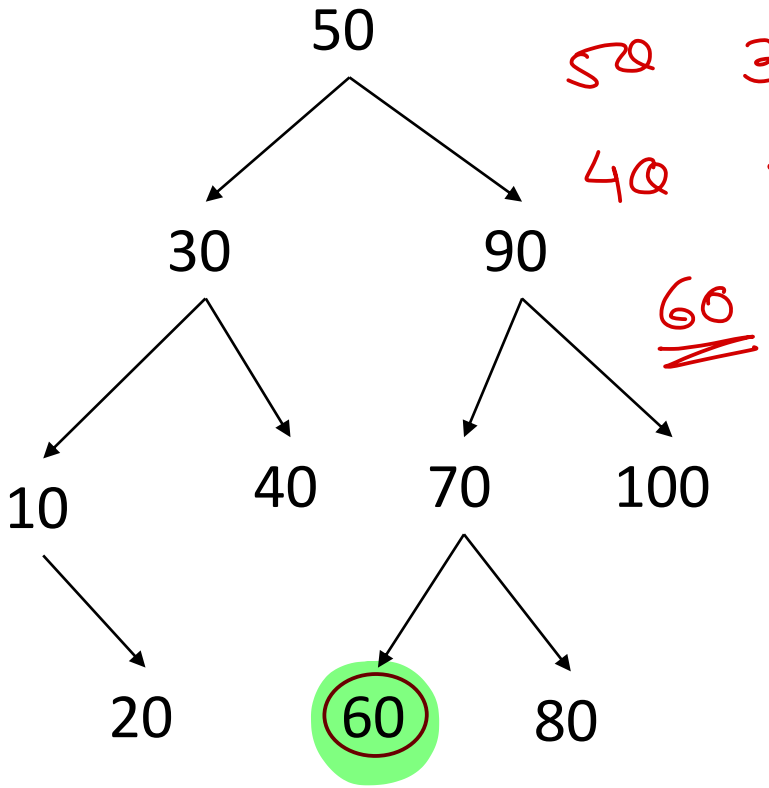
&40 &90 &70 &60



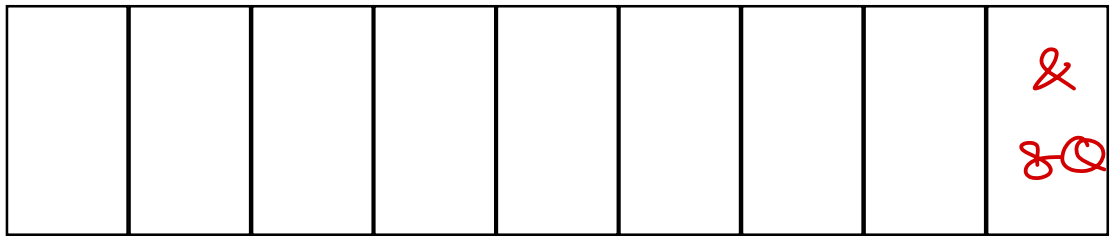

```
push(root)
while( stack not empty){
    pop trav
    if key == trav.data
        return trav;
    if have right child,
        push it on stack;
    if have left child,
        push it on stack;
}
return null;
```

BST – Non-Recursive Algorithm – BFS

level wise search
need queue.



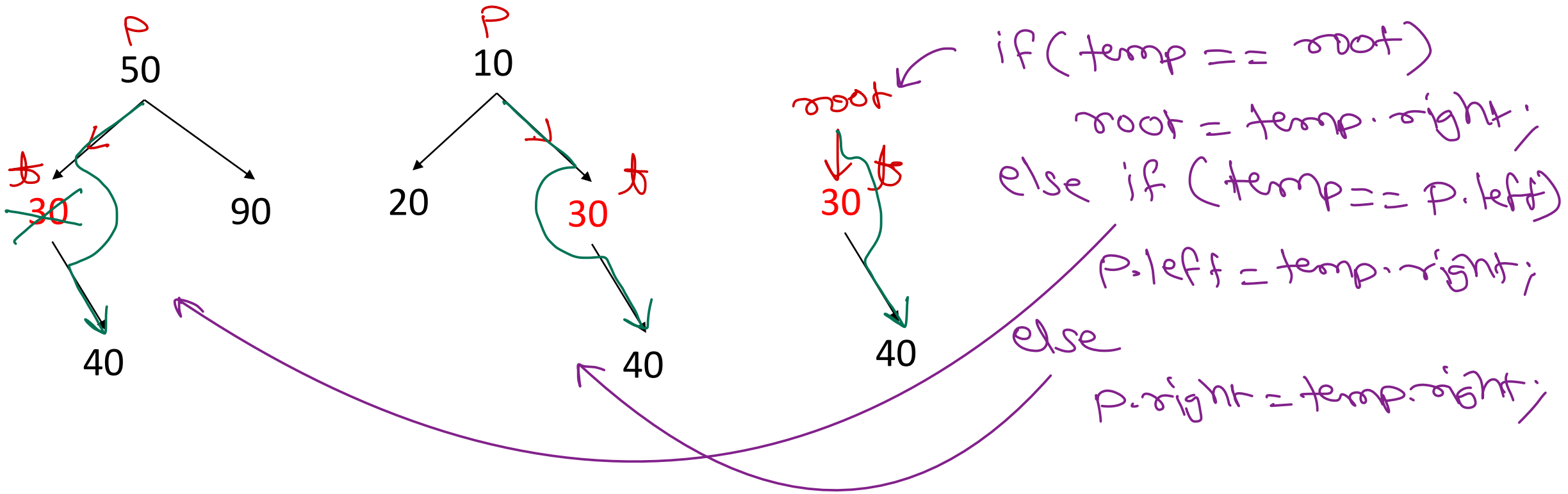
50 30 90 10
40 70 100 20
60




```
q.offer(root);
while(!q.isEmpty()){
    trav = q.poll();
    if(key == trav.data)
        return trav;
    if(trav.left!=null)
        q.offer(trav.left);
    if(trav.right!=null)
        q.offer(trav.right);
}
return null;
```

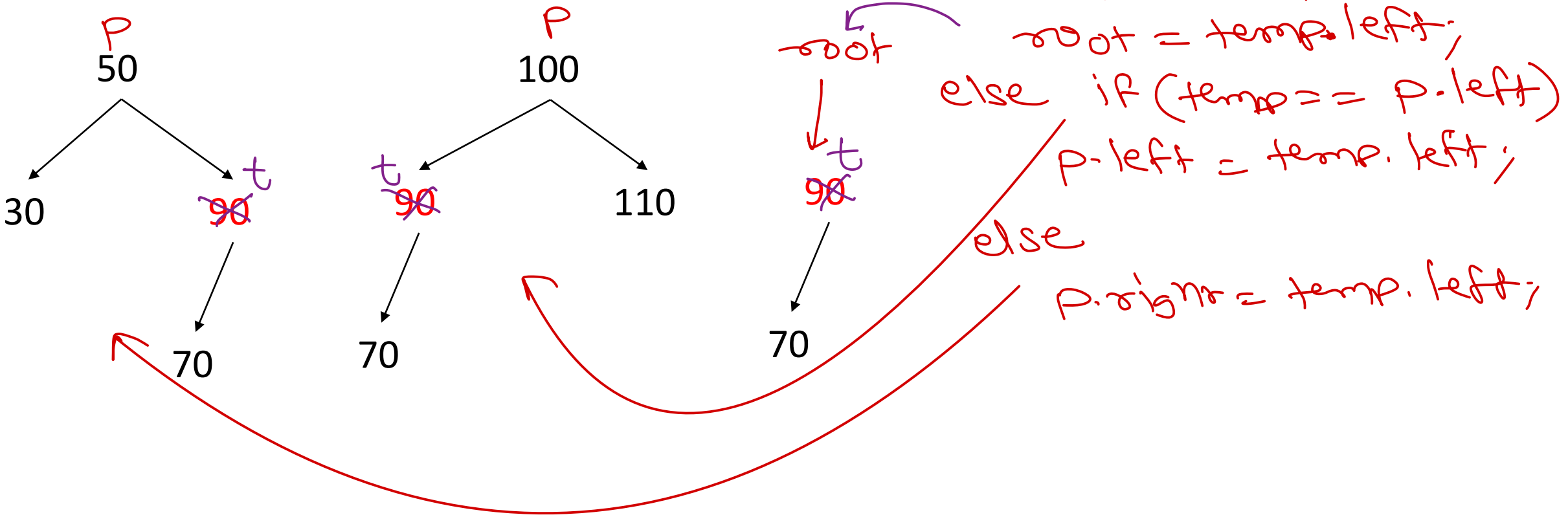
BST – Delete Node

temp.left == null



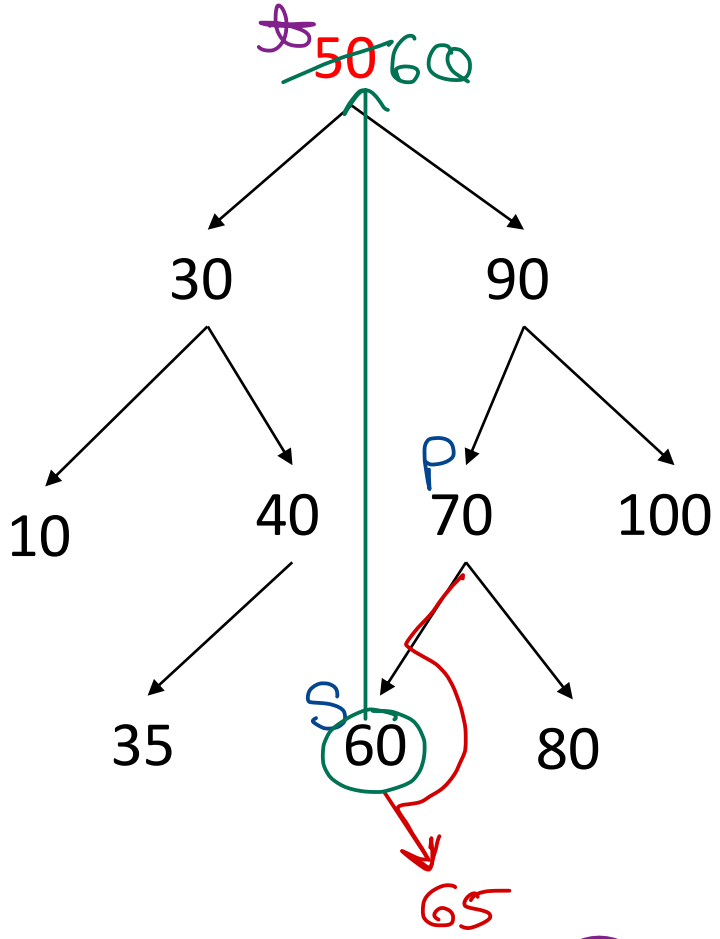
BST – Delete Node

temp.right == null



BST – Delete Node

temp.left != null && temp.right != null




```
parent = temp;
succ = temp.right;
while( succ.left != null){
    parent = succ;
    succ = succ.left;
}
3
temp.data = succ.data;
parent.left = succ.right;
```

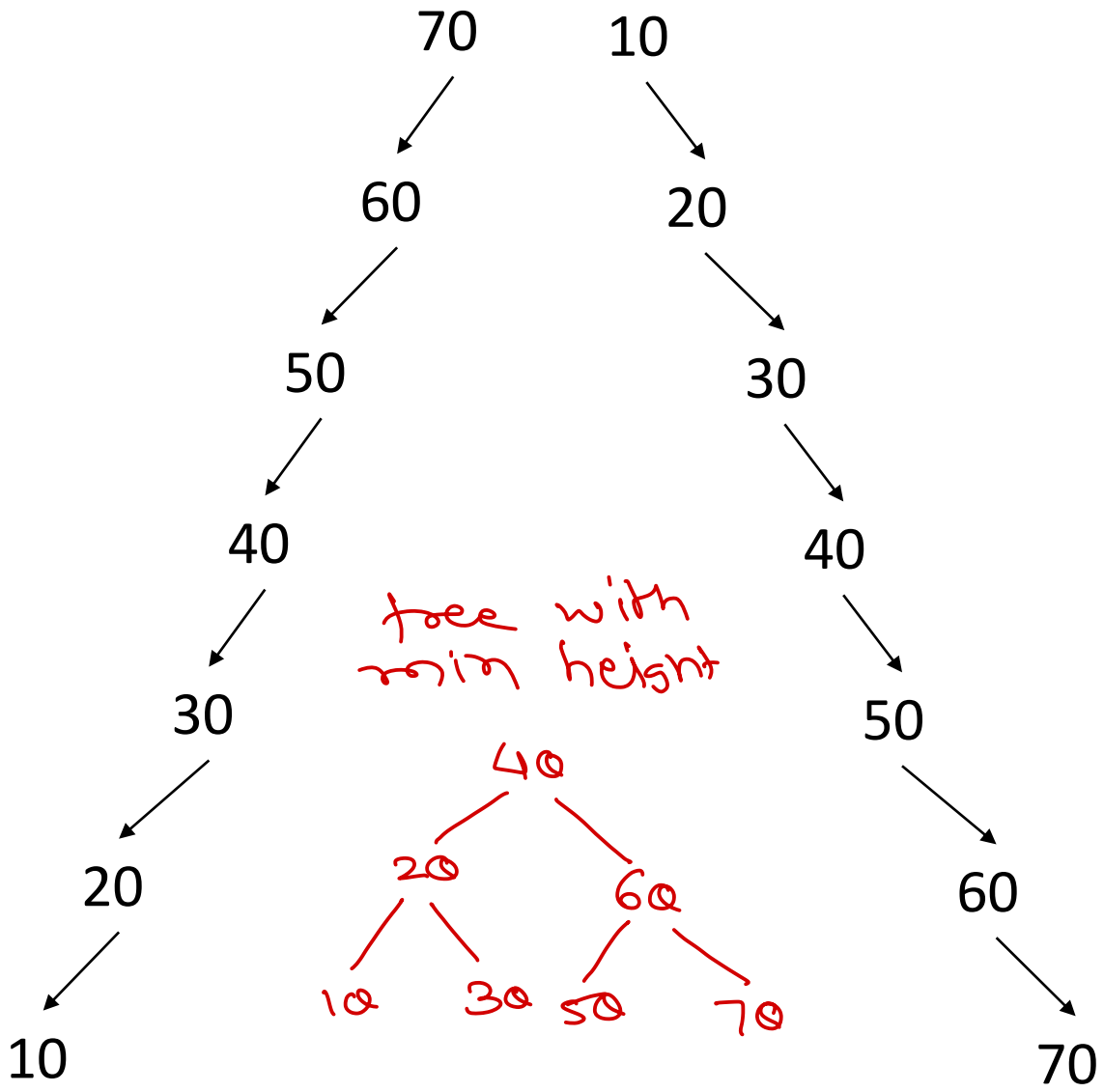
10 30 35 40 50 60

65 70 80 90 100

40 → inorder predecessor

60 → inorder successor ✓

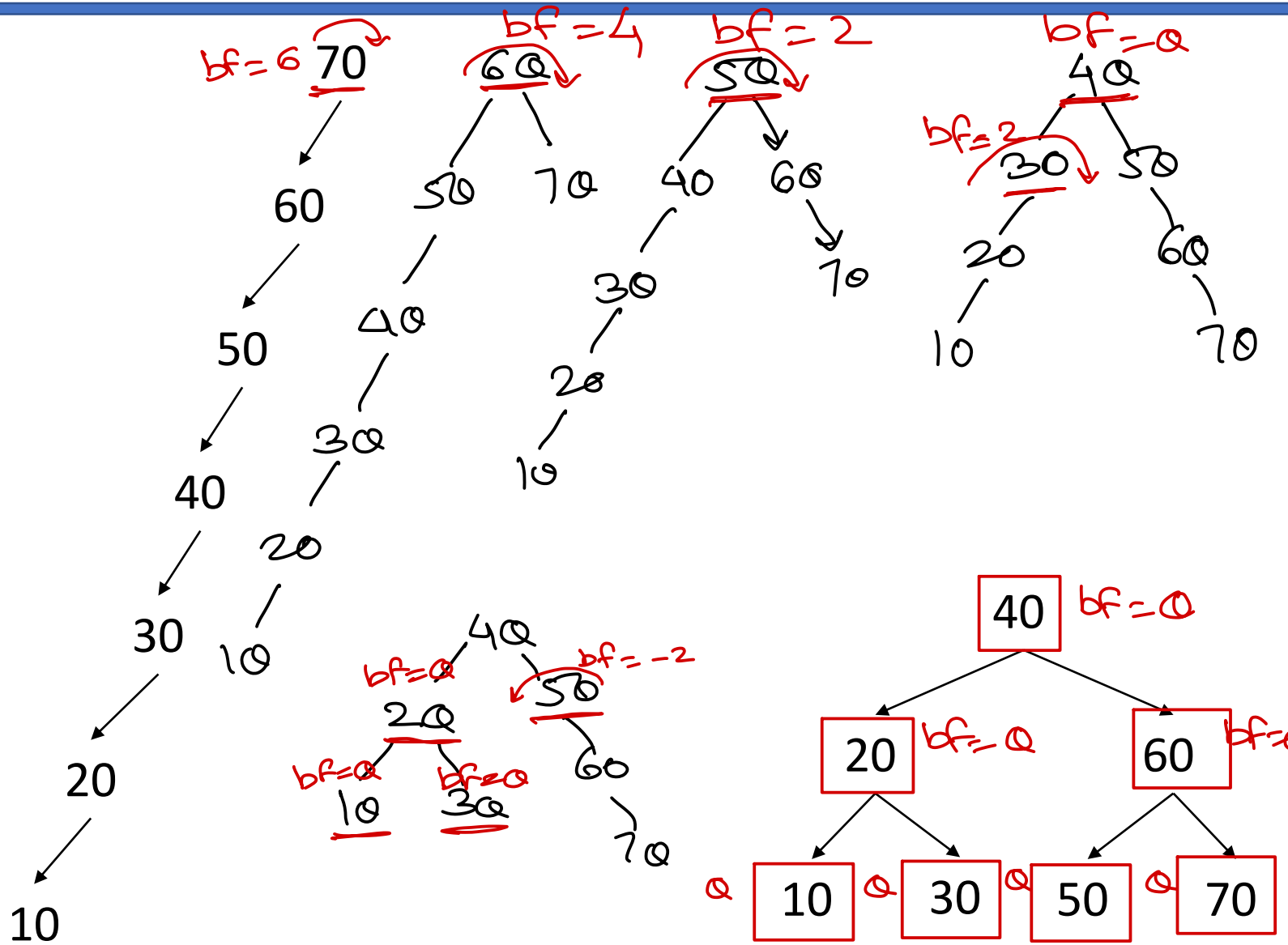
Skewed Binary Tree



- In Binary tree if only left or only right links are used, tree grows only on one side. Such tree is called as skewed binary tree.
  - Left skewed binary tree
  - Right skewed binary tree
- Time complexity of any BST is O(h).
- Such tree have maximum height i.e. same as number of elements.
- Time complexity of searching in skewed BST is O(n).

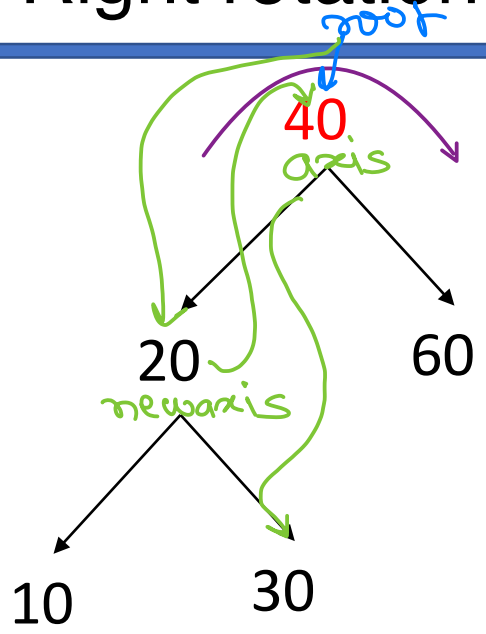
del()
add()
find()
height

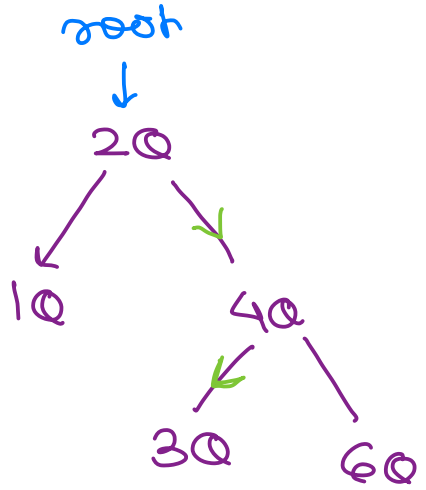
Balanced BST



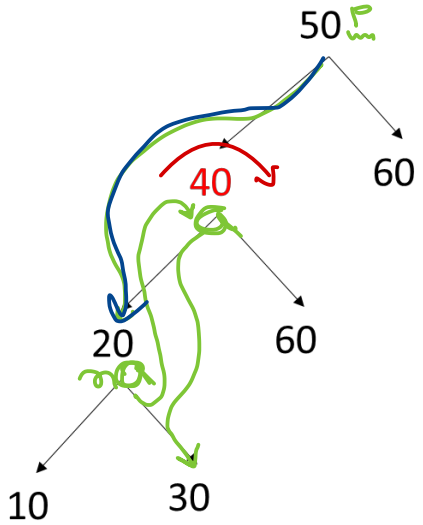
- To speed up searching, height of BST should minimum as possible.
- If nodes in BST are arranged so that its height is kept as less as possible, is called as Balanced BST.
- Balance factor of node
  - = Height of left sub tree – Height of ~~left~~ right sub tree
- In balanced BST, BF of each node is -1, 0 or +1.
- A tree can be balanced by applying series of left or right rotations on unbalanced nodes.

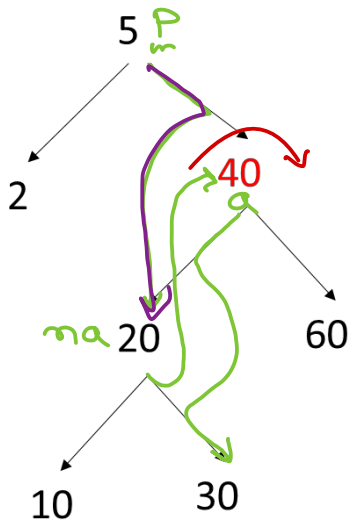
Right rotation



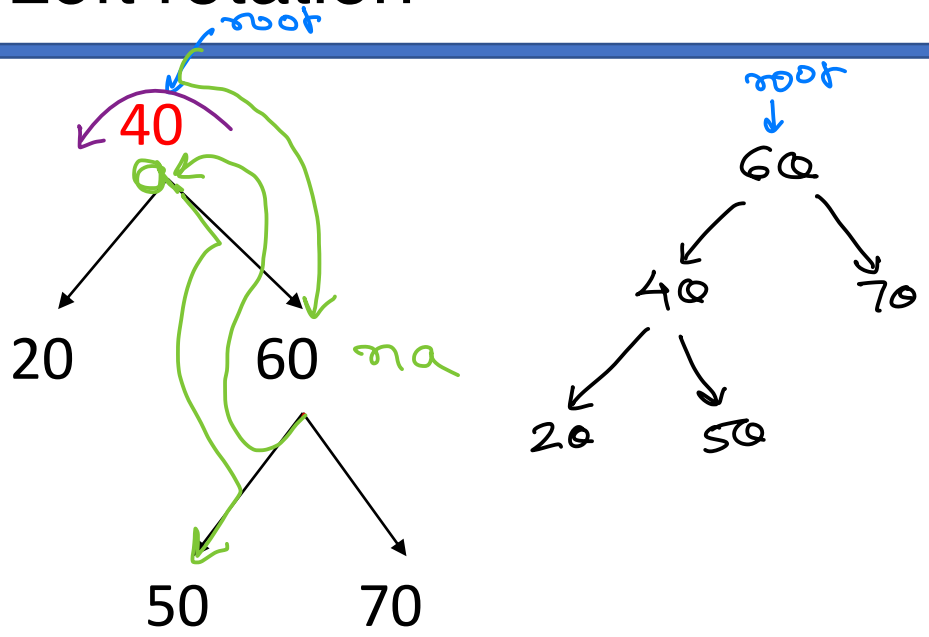



```
newaxis = axis.left;
axis.left = newaxis.right;
newaxis.right = axis;
if(axis == root)
    root = newaxis;
else if (axis == p.left)
    p.left = newaxis;
else
    p.right = newaxis;
```

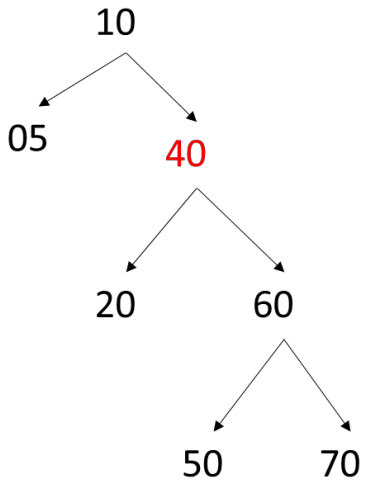


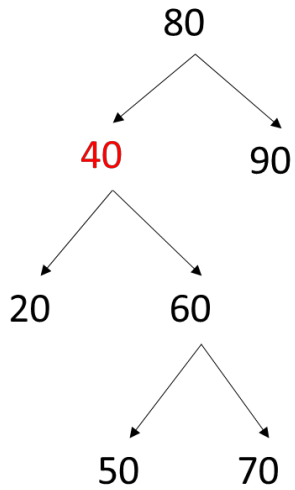


Left rotation

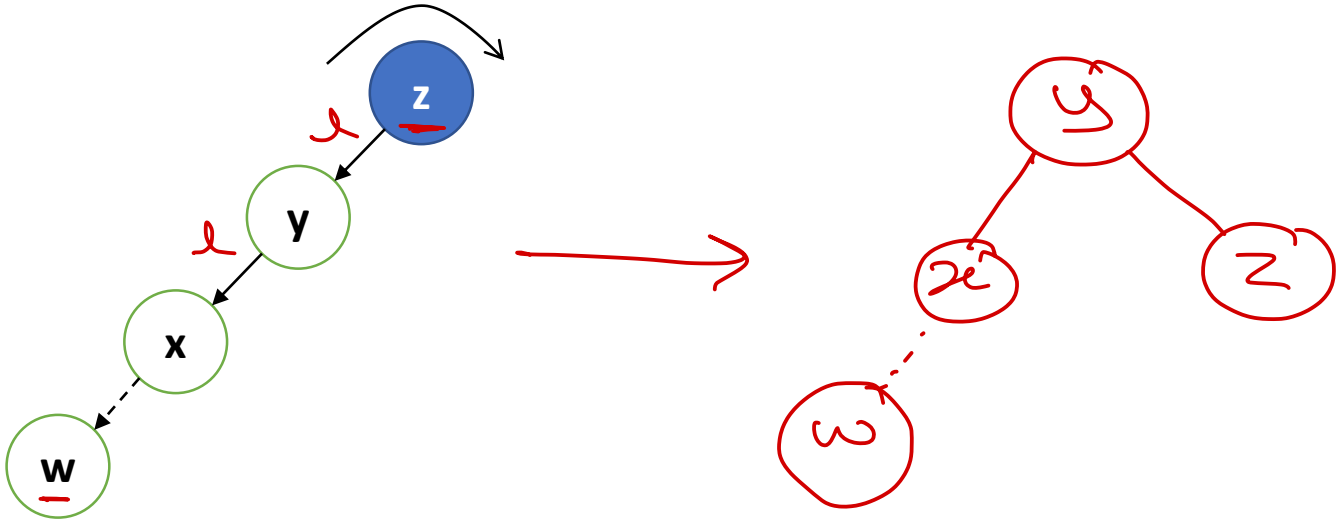



```
na = a->right;
a->right = na->left;
na->left = a;
if(a == root)
    root = na;
else if (a == p->left)
    p->left = na;
else
    p->right = na;
```



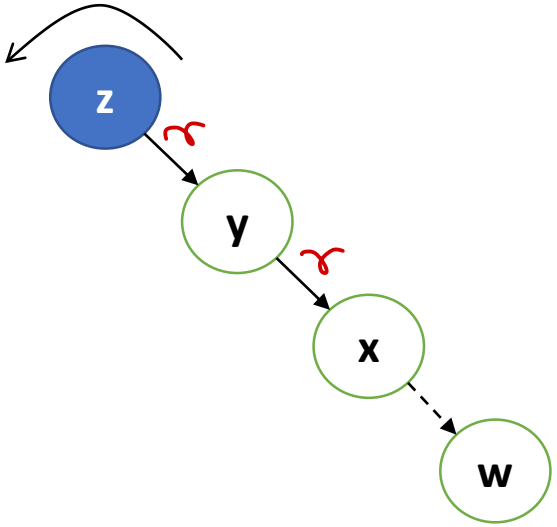


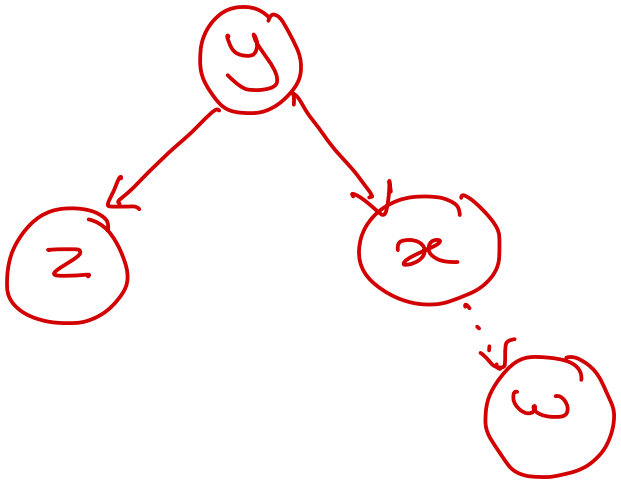
Rotation cases



Left-Left case

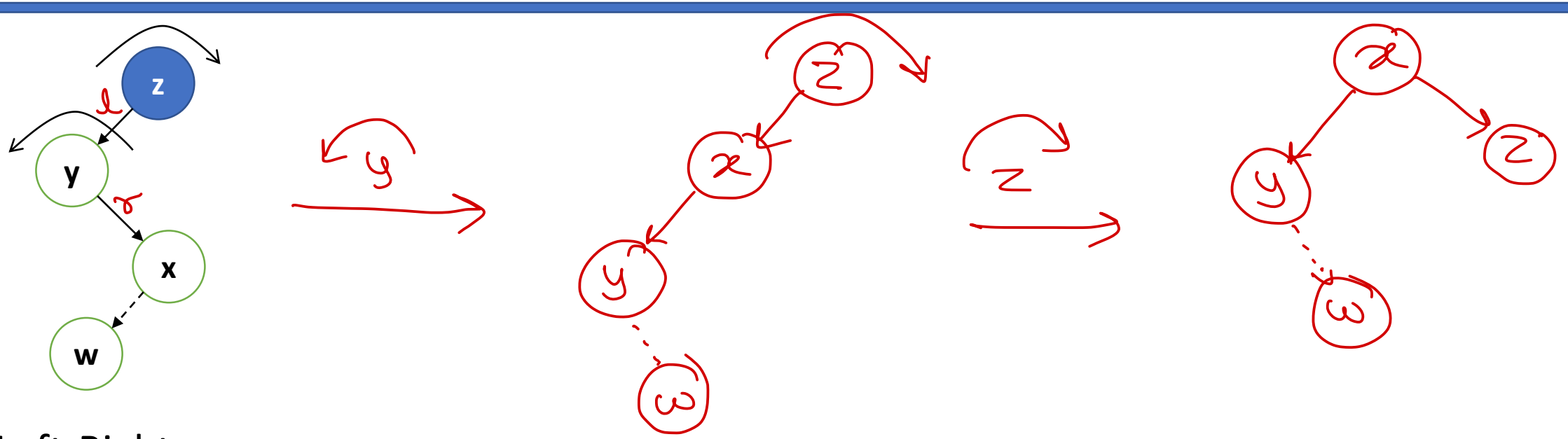
Rotation cases





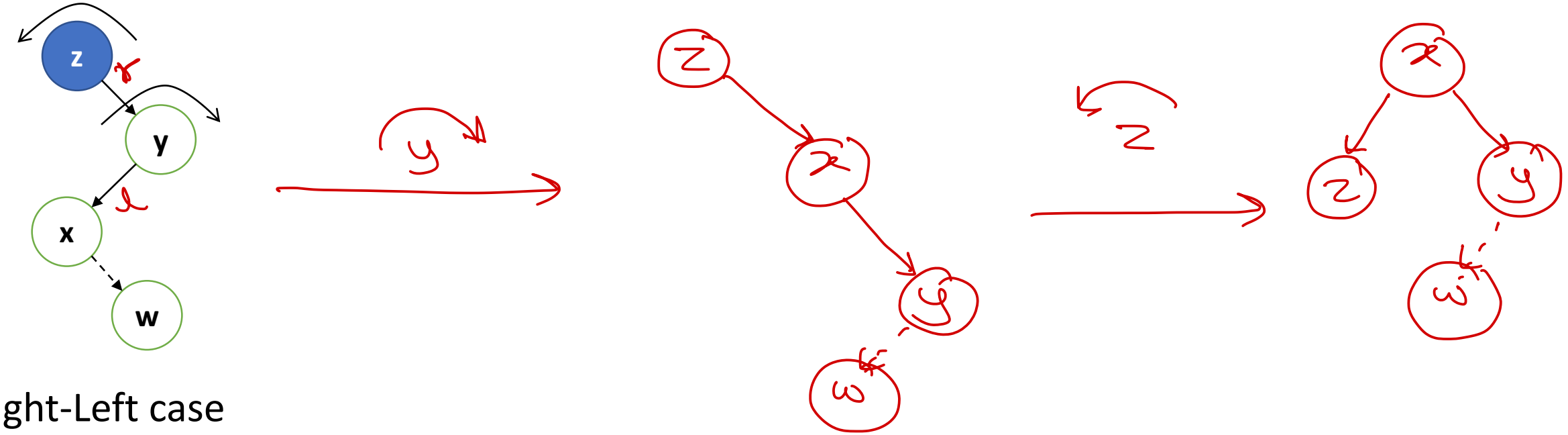
Right-Right case

Rotation cases



Left-Right case

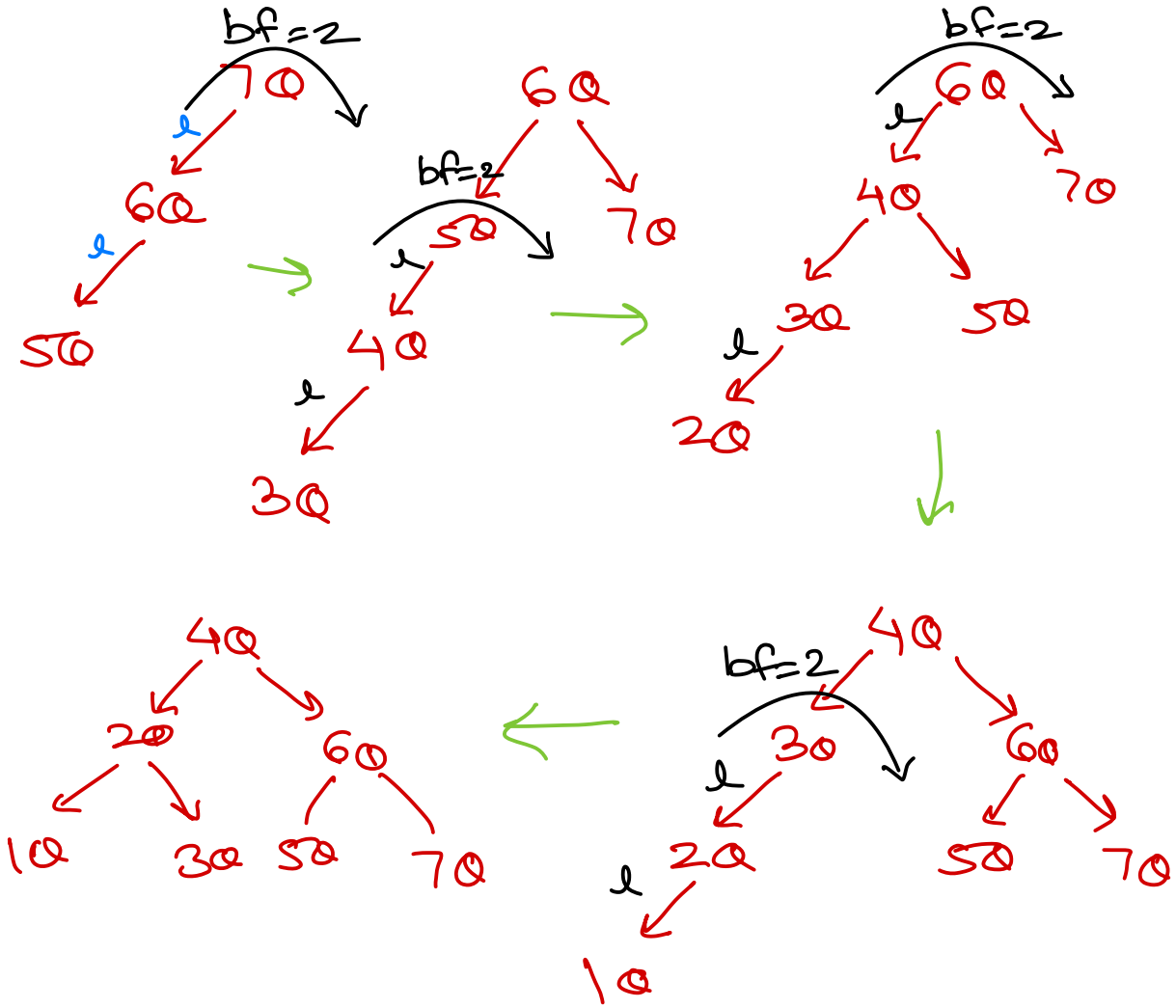
Rotation cases



Right-Left case

AVL Tree

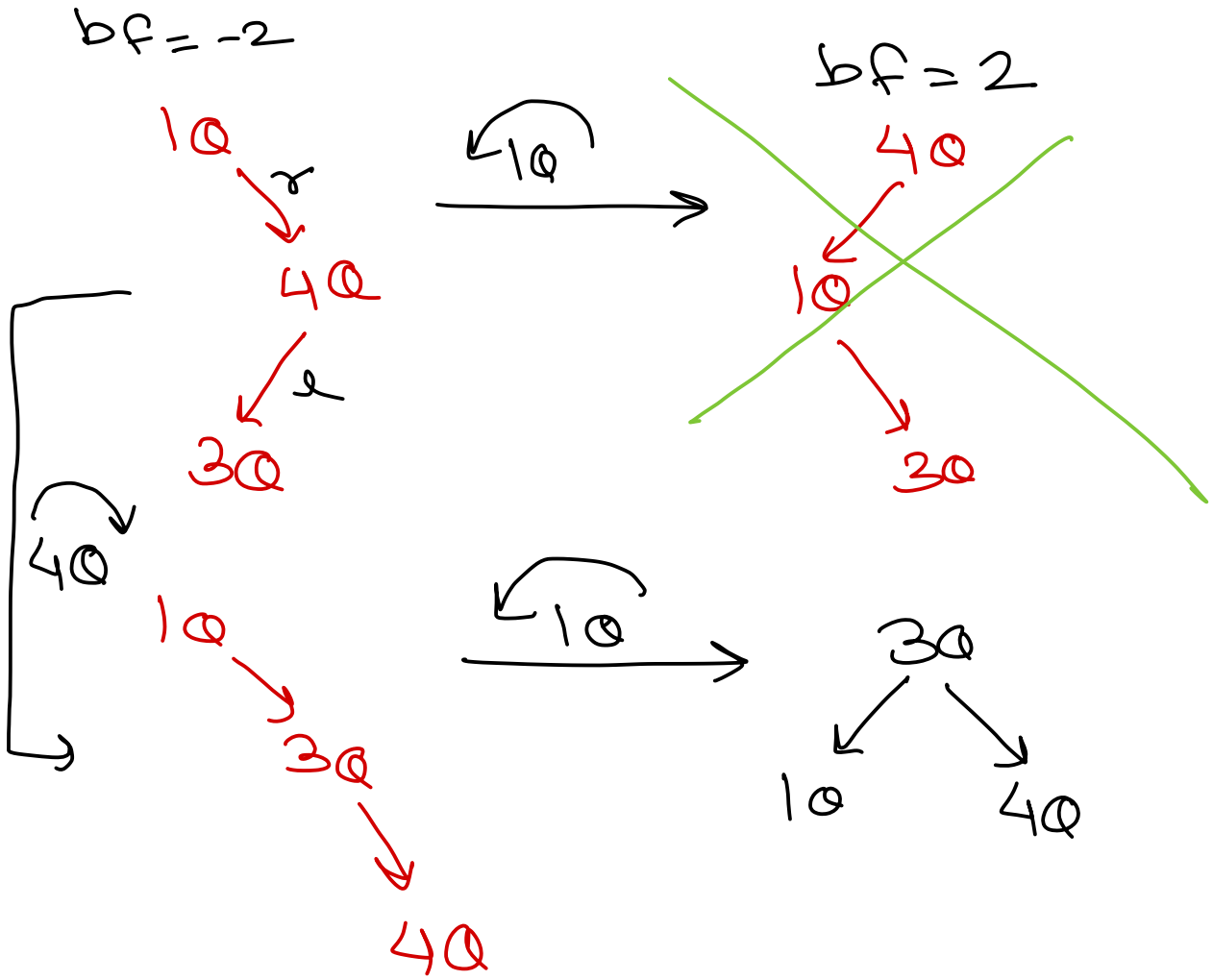
- AVL tree is a self-balancing Binary Search Tree (BST).
- The difference between heights of left and right subtrees cannot be more than one for all nodes.
- Most of BST operations are done in O(h) i.e. O(log n) time.
- Nodes are rebalanced on each insert operation and delete operation.
- Need more number of rotations as compared to Red & Black tree.



AVL Tree

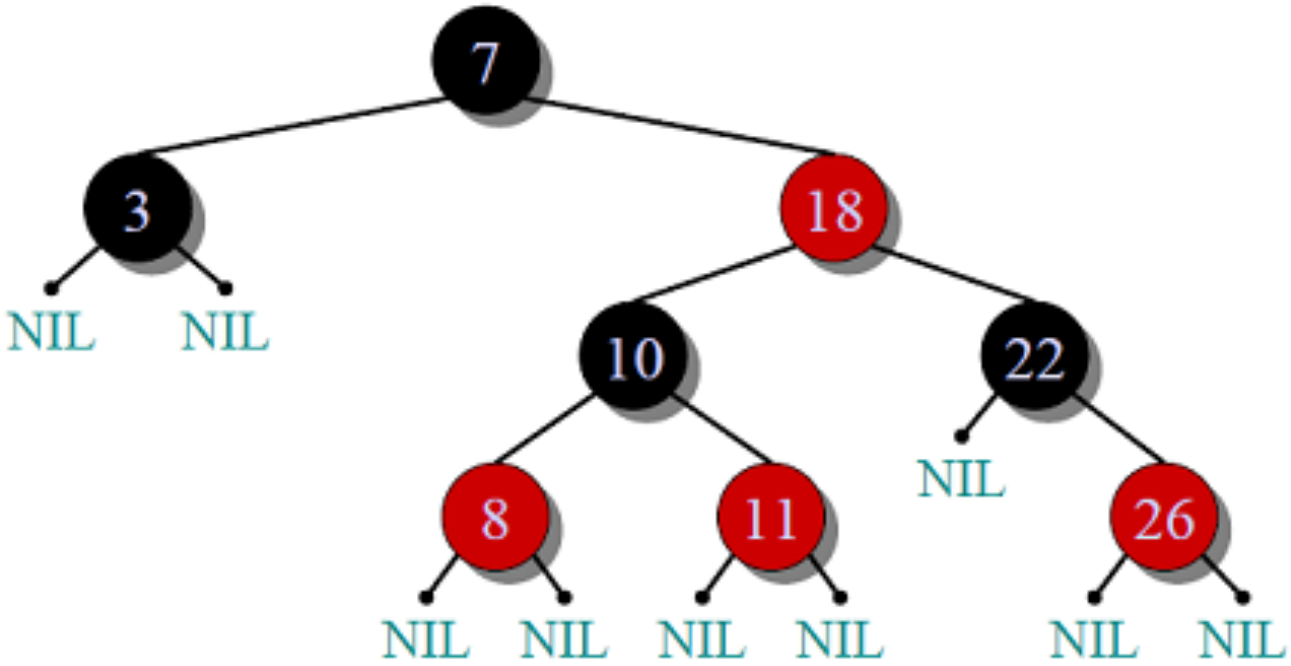
10 40 30 20

- AVL tree is a self-balancing Binary Search Tree (BST).
- The difference between heights of left and right subtrees cannot be more than one for all nodes.
- Most of BST operations are done in O(h) i.e. O(log n) time.
- Nodes are rebalanced on each insert operation and delete operation.
- Need more number of rotations as compared to Red & Black tree.

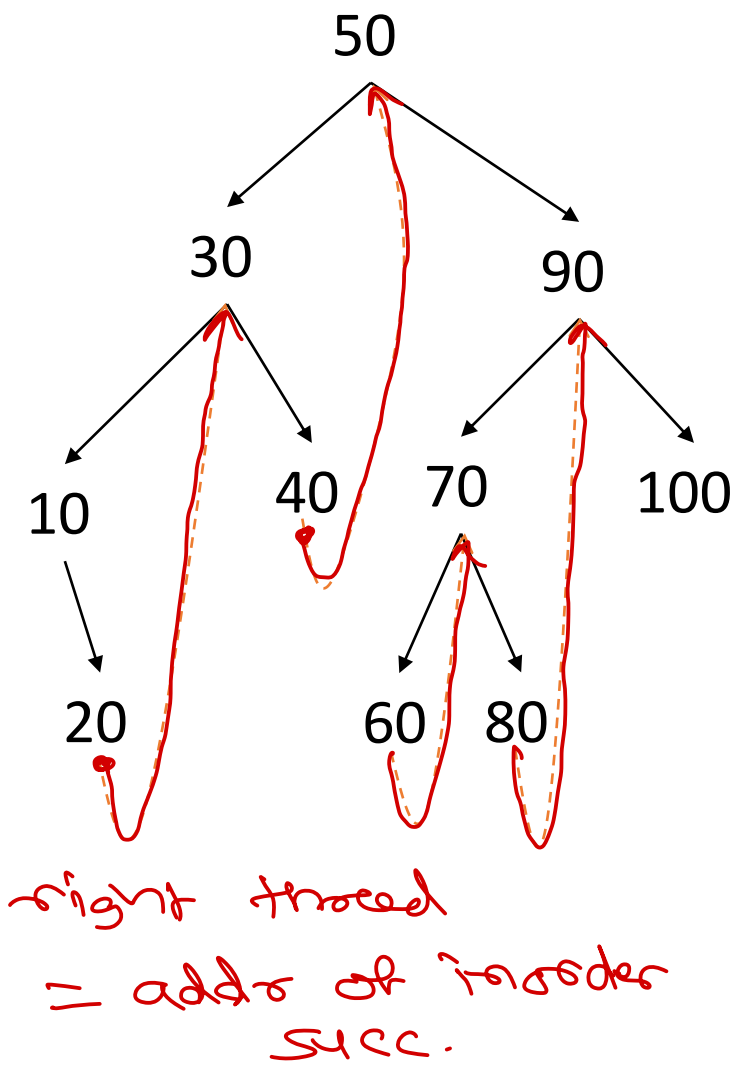


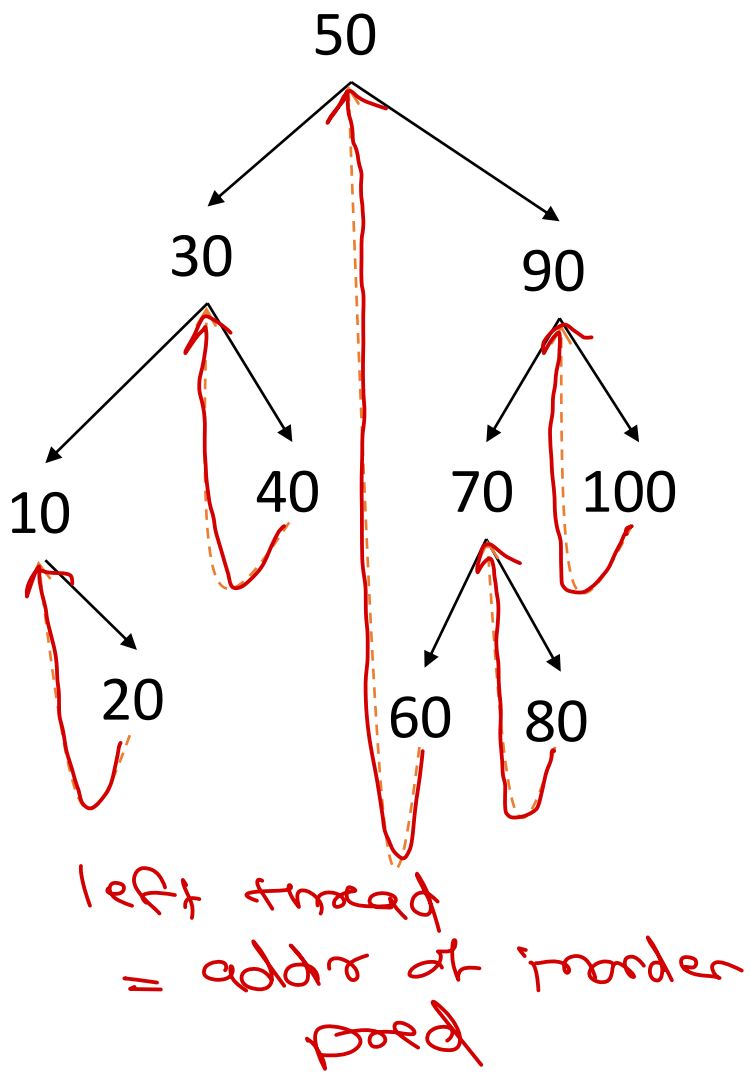
Red & Black tree

- Red & Black tree is a self-balancing Binary Search Tree (BST).
- Each node follows some rules:
  - Every node has a color either red or black.
  - Root of tree is always black.
  - Two adjacent cannot be red nodes (Parent color should be different than child).
  - Every path from a node (including root) to any of its descendant NULL node has the equal number of black nodes.
- Most of BST operations are done in O(h) i.e. O(log n) time.
- For frequent insert/delete, RB tree is preferred over AVL tree.



Threaded BST





- Typical BST in-order traversal involves recursion or stack. It slows execution and also need more space.
- Threaded BST keep address of in-order successor or predecessor addresses instead of NULL to speed up in-order traversal (using a loop).
- Left threaded BST
- Right threaded BST
- In-threaded BST
  = left thread + right thread.

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>