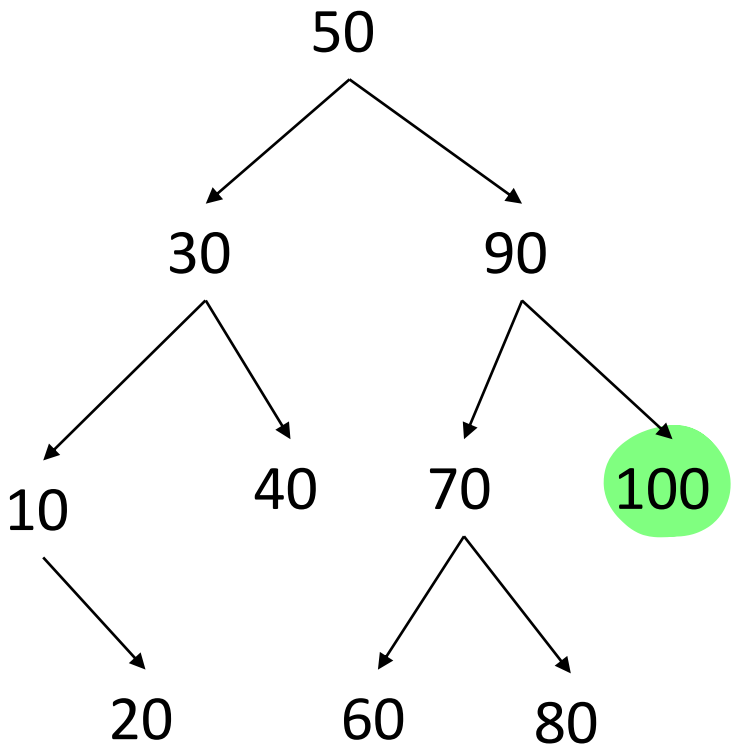
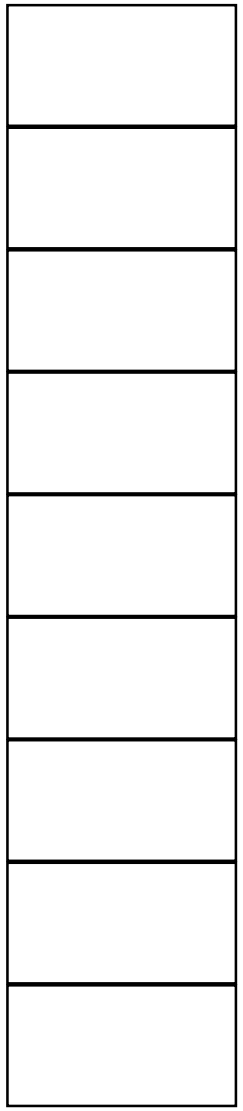
Data Structure & Algorithms

Nilesh Ghule

BST – Non-Recursive Algorithm – DFS



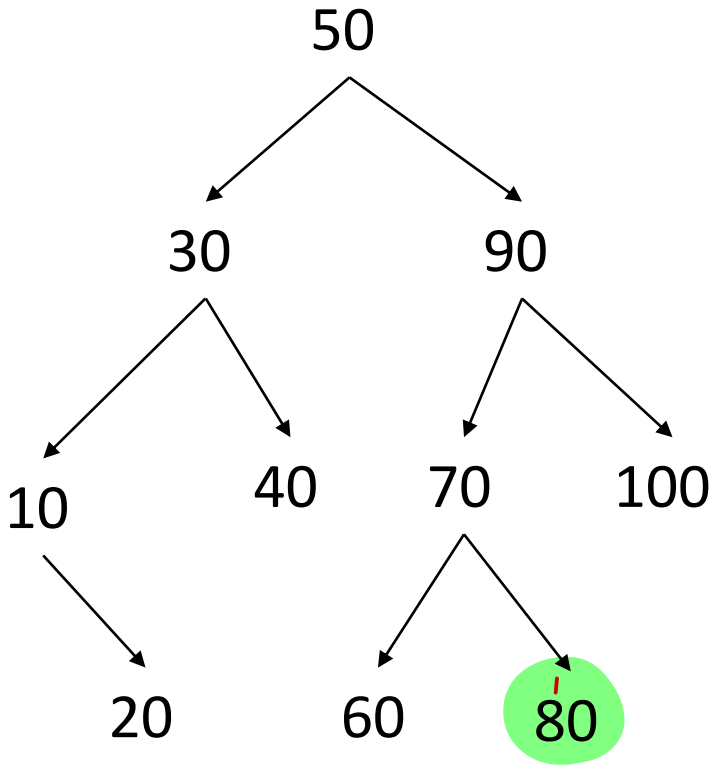


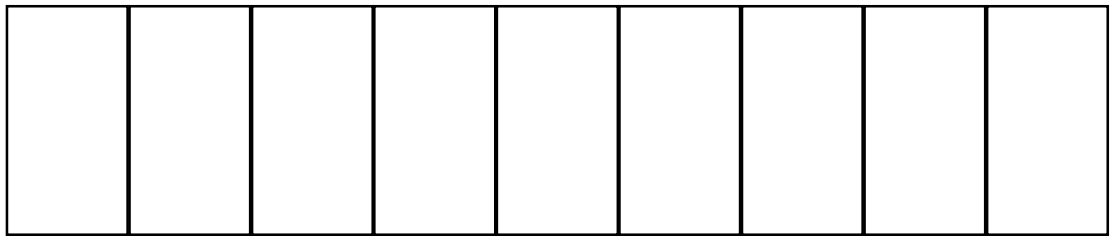
50 30 10 20 40

90 70 60 80 100

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

BST – Non-Recursive Algorithm – BFS - level wise search

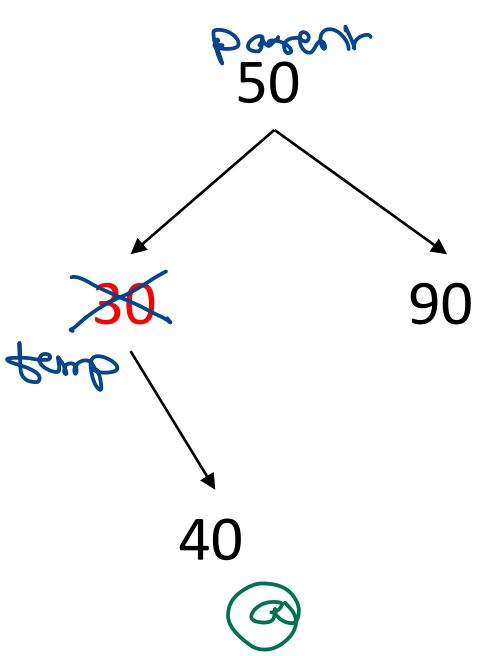


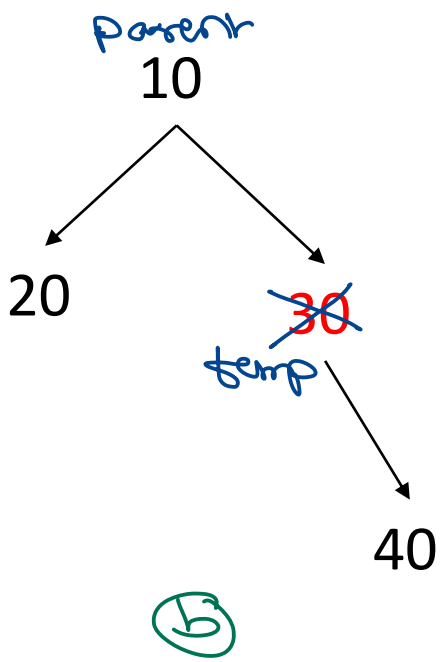


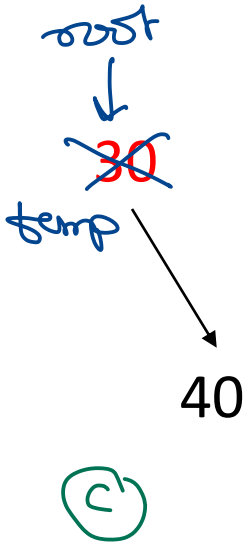
```
q.push(root);
while(!q.isEmpty()){
    trav=q.pop();
    print(trav.data);
    if(trav.left!=null)
        q.push(trav.left);
    if(trav.right!=null)
        q.push(trav.right);
}
```

50 30 90 10 40 70 100 20 60 80

BST – Delete Node

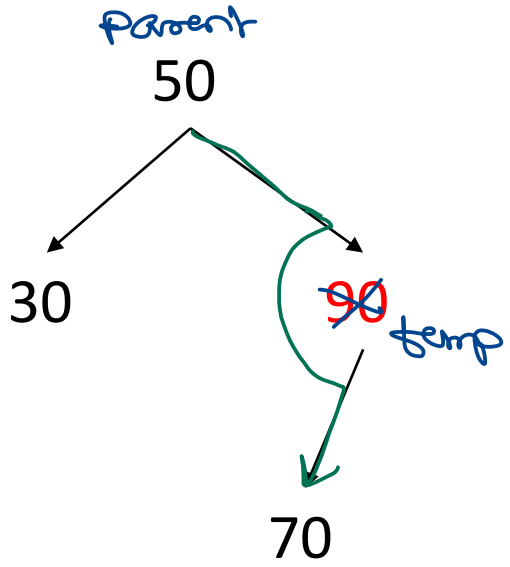


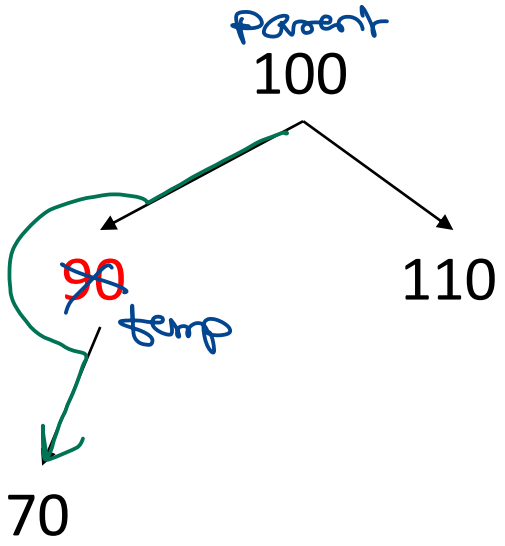


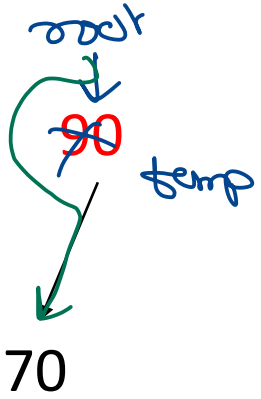



```
if (temp.left == null) {
    if (temp == root)
        c) root = temp.right;
    else if (temp == parent.left)
        a) parent.left = temp.right;
    else // parent.right
        b) parent.right = temp.right;
}
```

BST – Delete Node

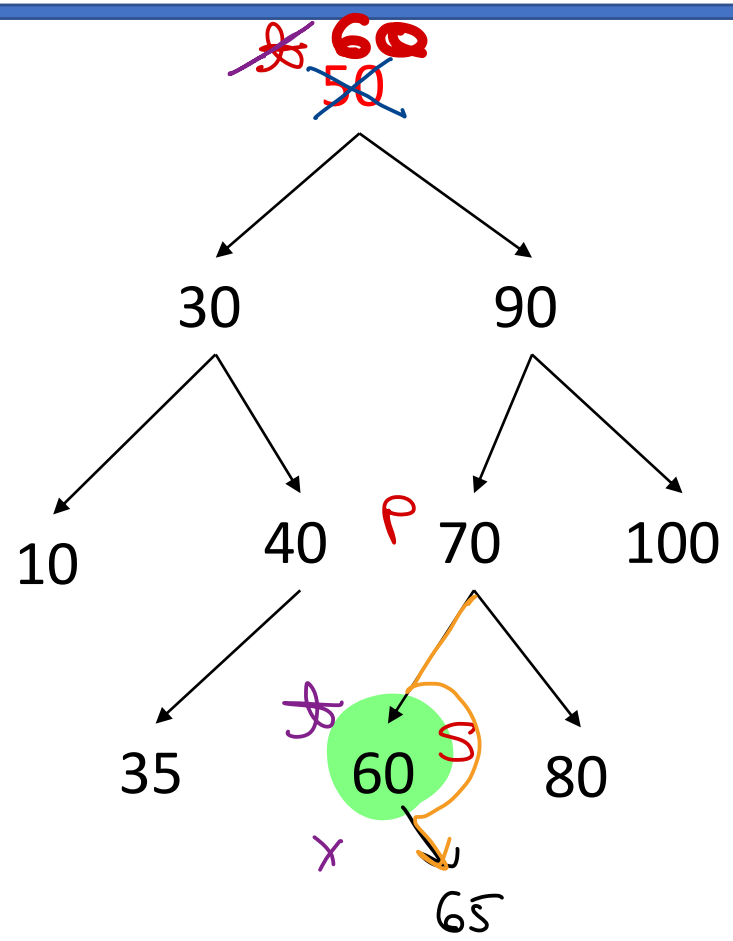







```
if(temp.right == null) {
    if (temp == root)
        root = temp.left;
    else if (temp == parent.left)
        parent.left = temp.left;
    else // parent.right
        parent.right = temp.left;
}
```

BST – Delete Node

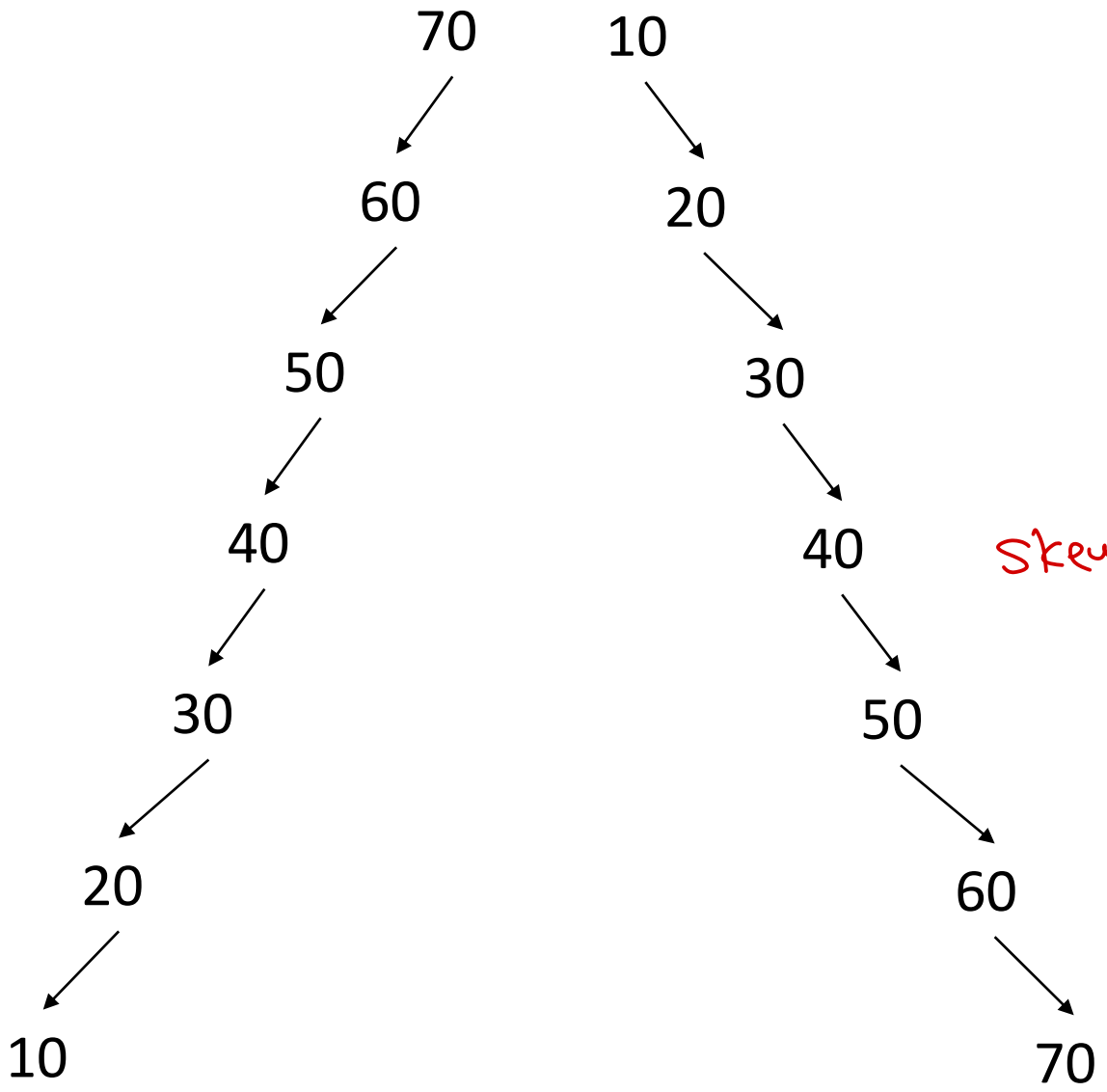



```
parent = temp;
succ = temp.right;
while(succ.left != null) {
    parent = succ;
    succ = succ.left;
}
temp.data = succ.data;
temp = succ;
parent.left = temp.right;
```

10 30 35 40 ~~50~~ 60

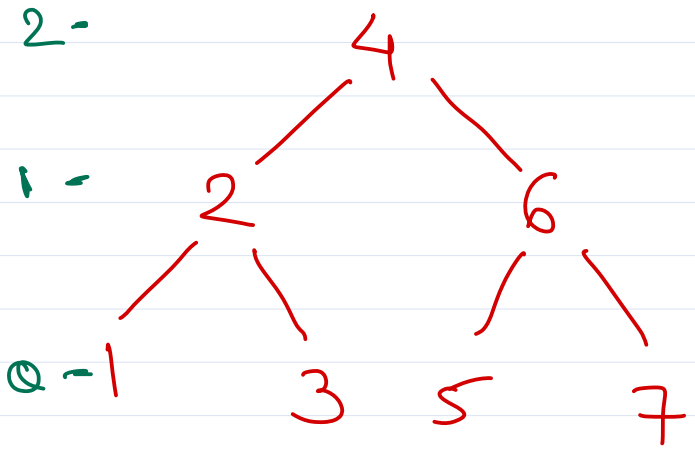
65 70 80 90 100

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).

Binary Search → O(n)



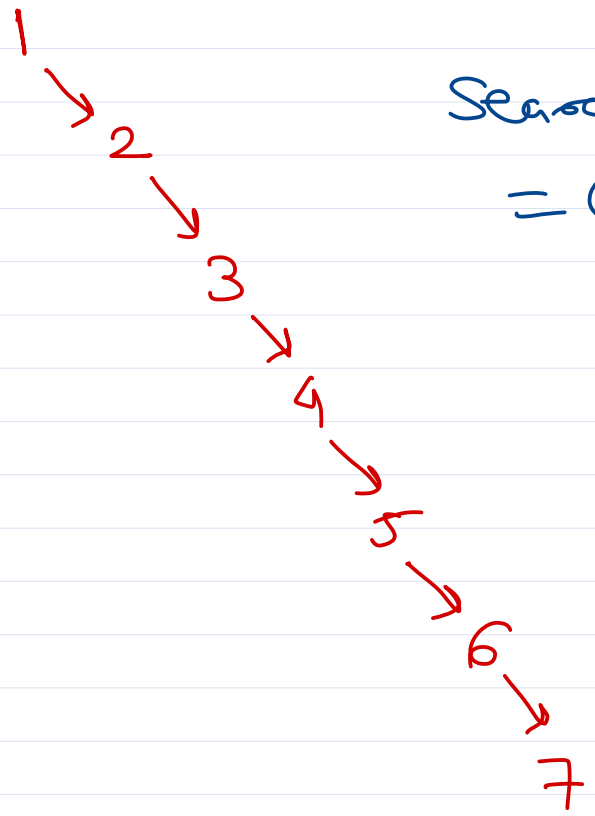
max itrs = 3

$2^{h+1} = n$

$h = \frac{\log n}{\log 2} - 1$

$T \propto \frac{\log n}{\log 2} - 1$

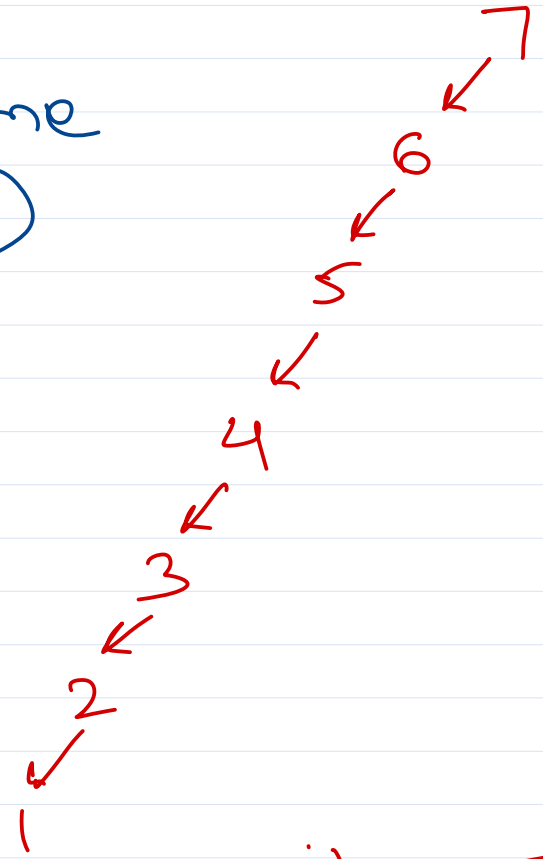
$T \propto \log n$



Search time
= O(n)

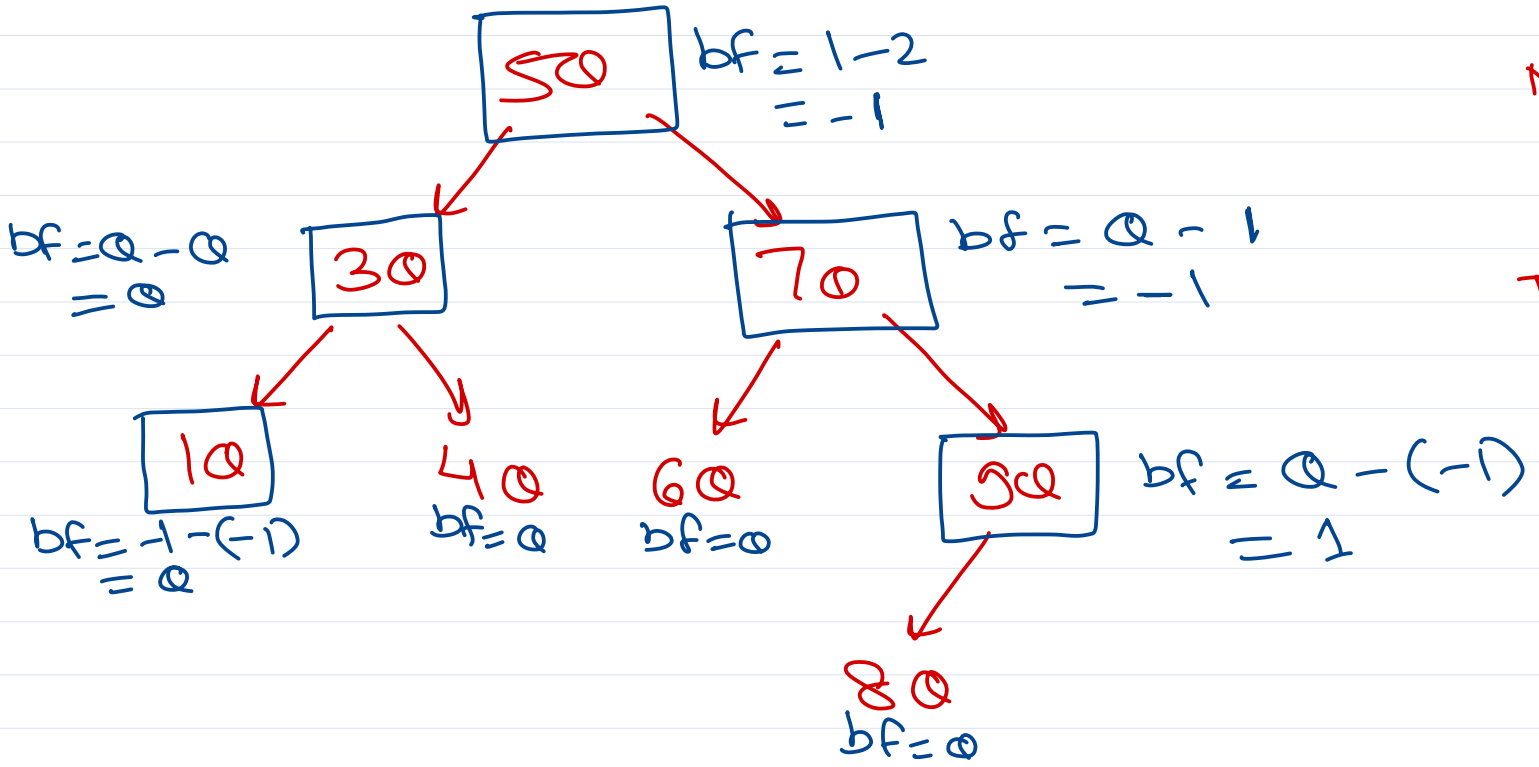
max itrs = 7

Right skewed BST



max itrs = 7

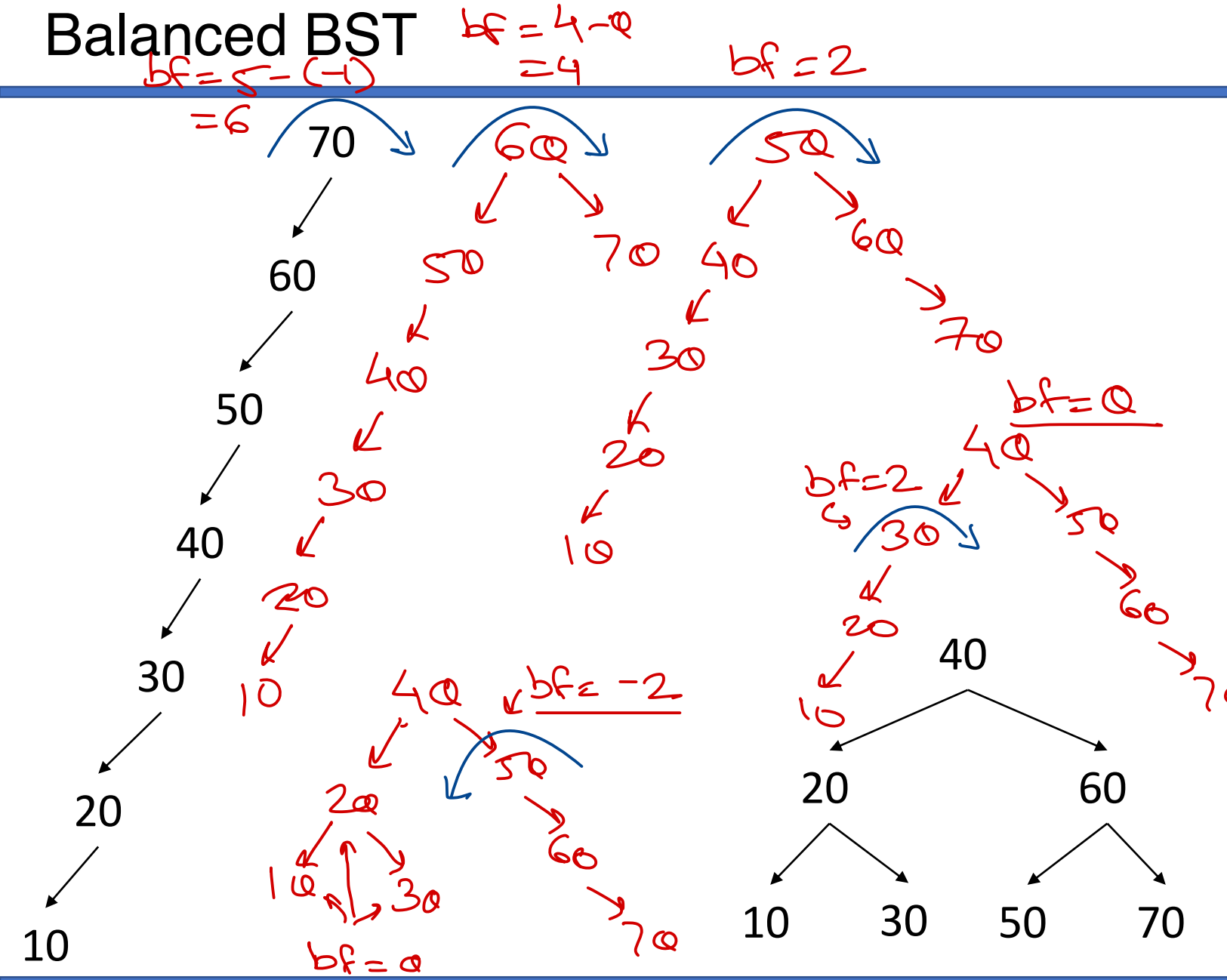
Left skewed BST



Node is said to be balanced, if its B.F. is -1, 0, or 1.

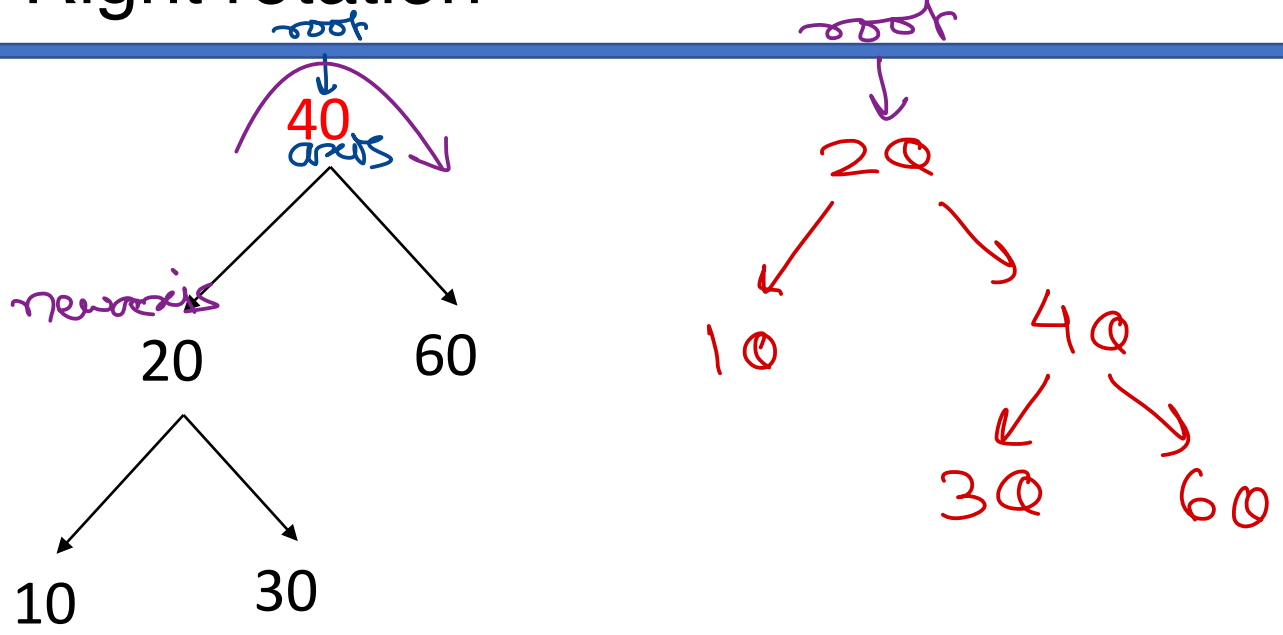
Tree is said to be balanced, if each node in tree is balanced.

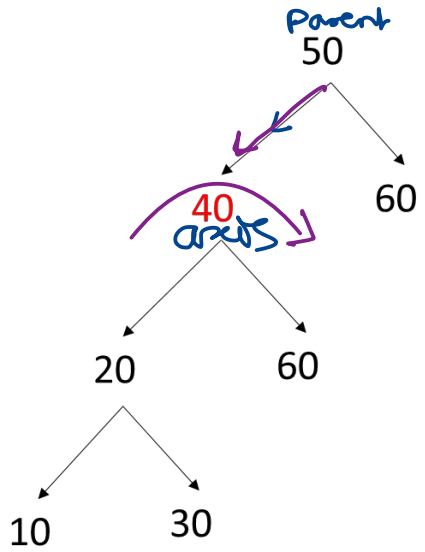
Balanced BST

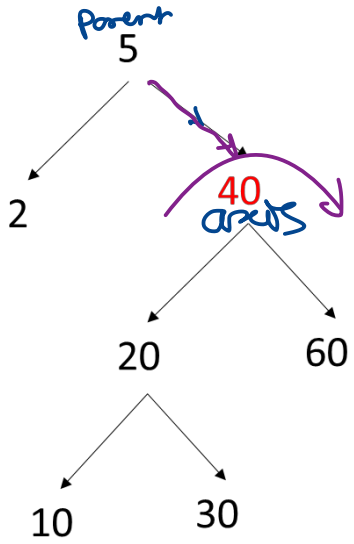


- To speed up searching, height of BST should minimum as possible.
- If nodes in BST are arranged so that its height is kept as less as possible, is called as Balanced BST.
- Balance factor of a node
  - = Height of left sub tree – Height of ~~left~~ right sub tree
- In balanced BST, BF of each node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

Right rotation

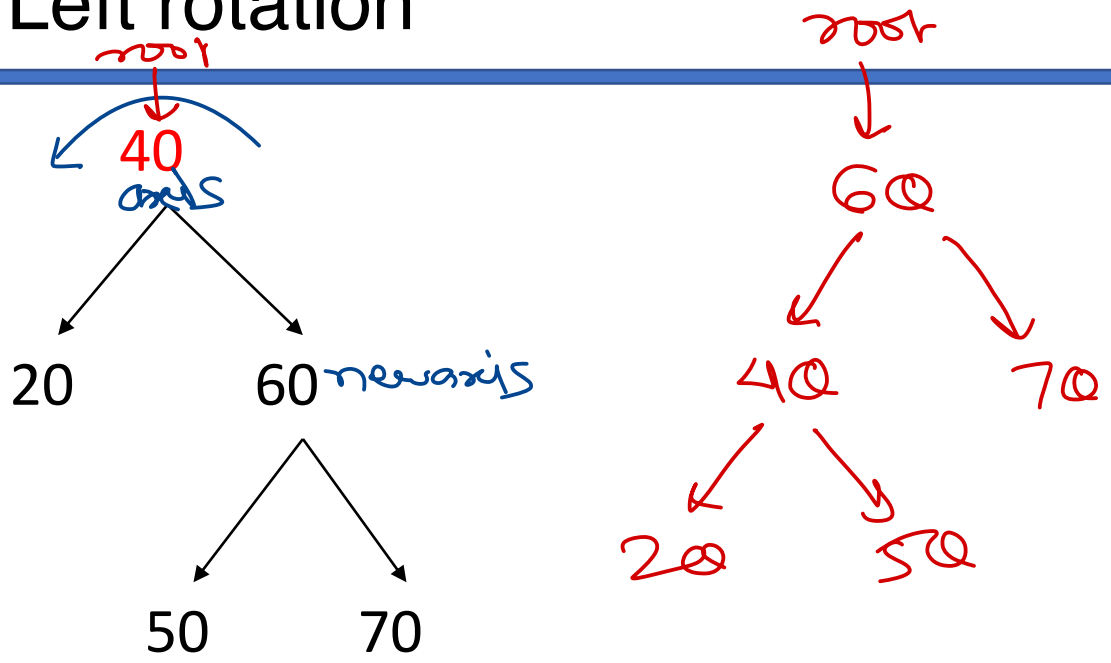




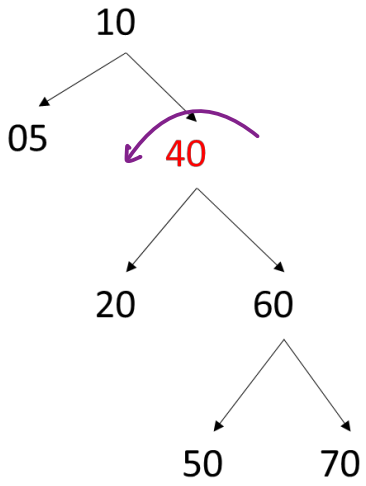


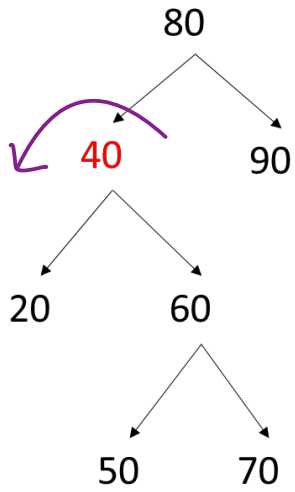

```
newaxis = axis.left;
axis.left = newaxis.right;
newaxis.right = axis;
if(axis == root)
    root = newaxis;
else if(axis == parent.left)
    parent.left = newaxis;
else
    parent.right = newaxis;
```

Left rotation

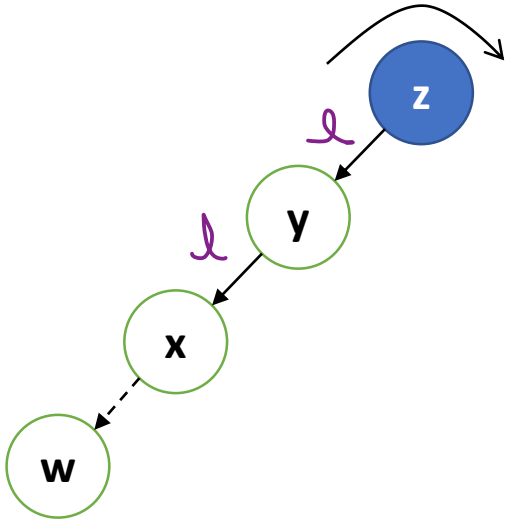


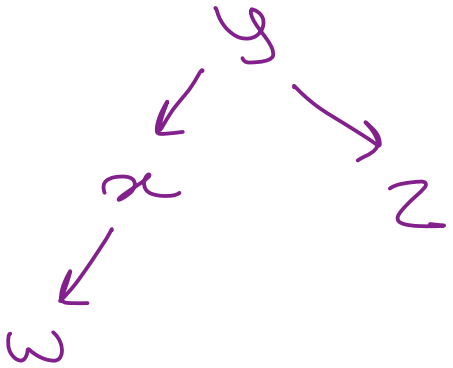

```
newaxis = axis.right;
axis.right = newaxis.left;
newaxis.left = axis;
if(axis == root)
    root = newaxis;
else if(axis == parent.left)
    parent.left = newaxis;
else
    parent.right = newaxis;
```





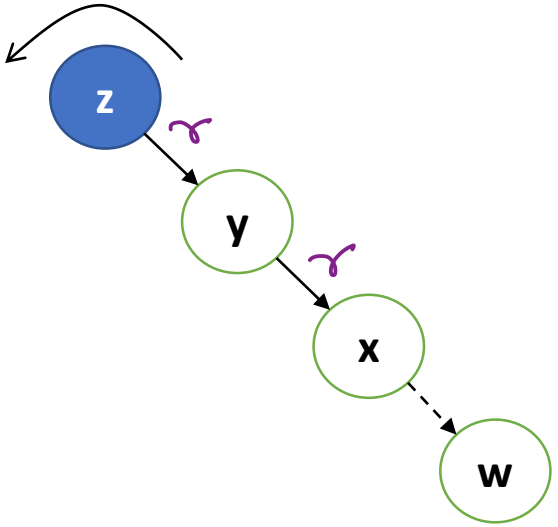
Rotation cases

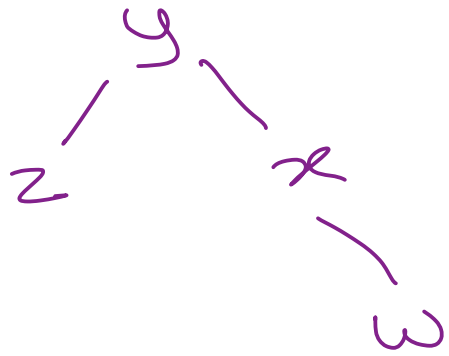




Left-Left case → right rotation

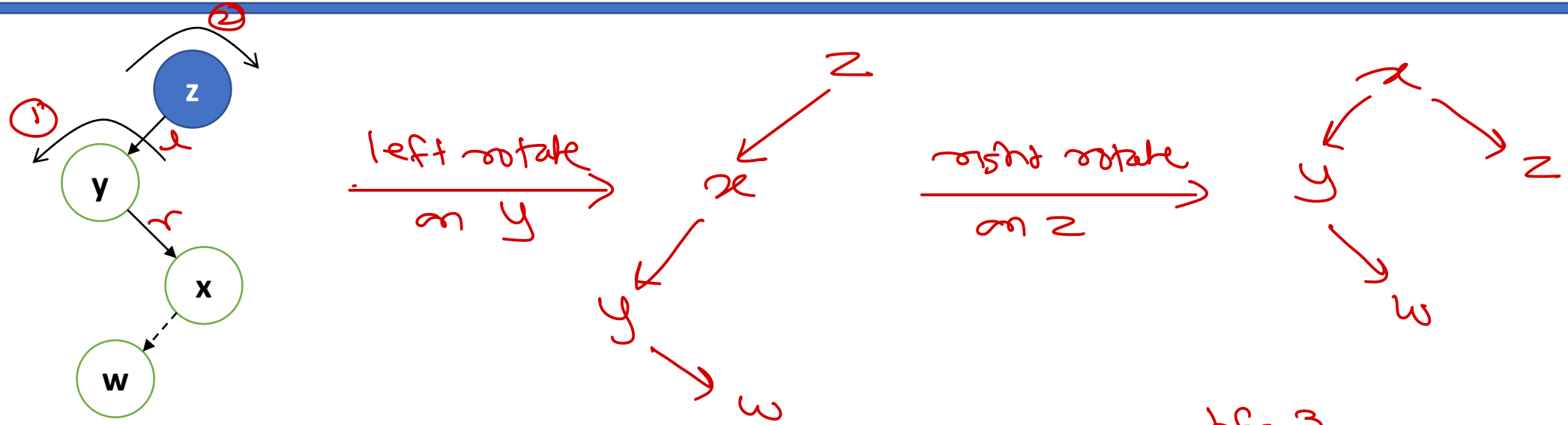
Rotation cases



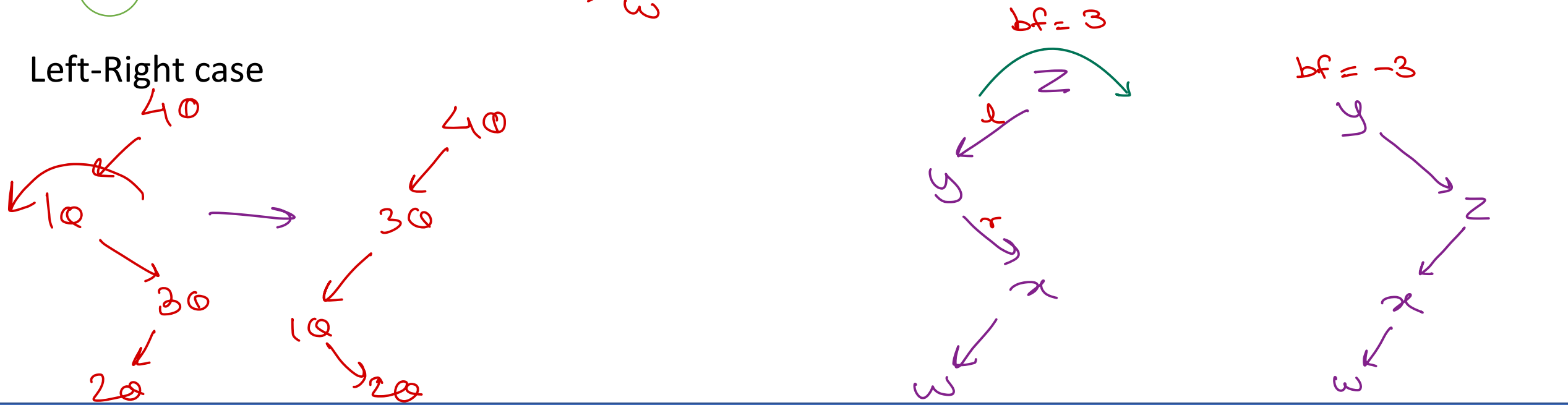


Right-Right case → left rotation

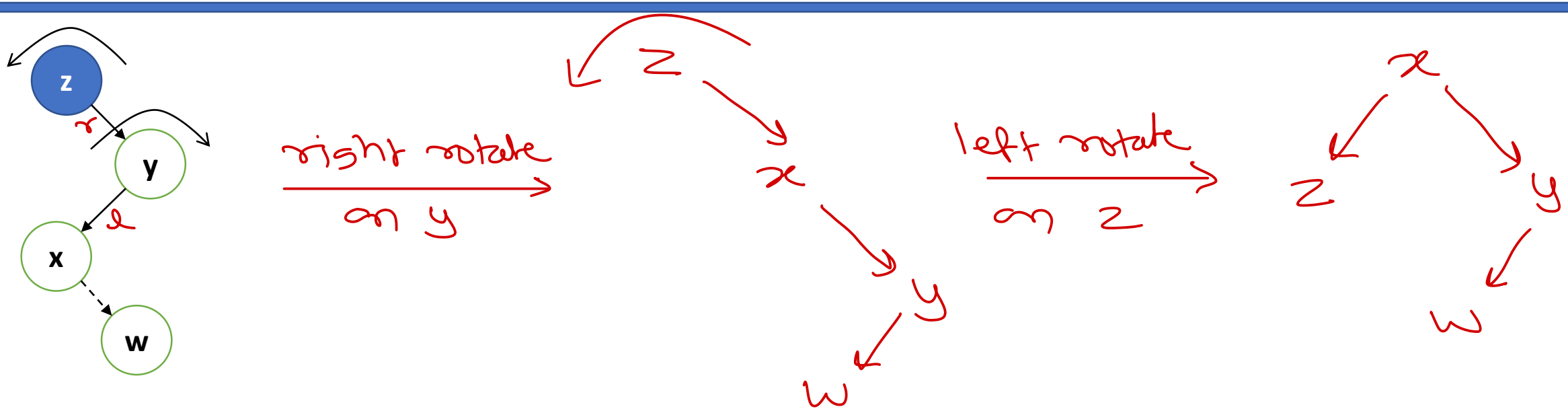
Rotation cases



Left-Right case



Rotation cases



Right-Left case

Nodes

1

2

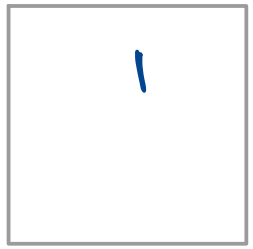
3

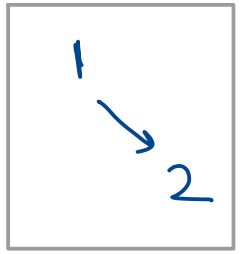
4

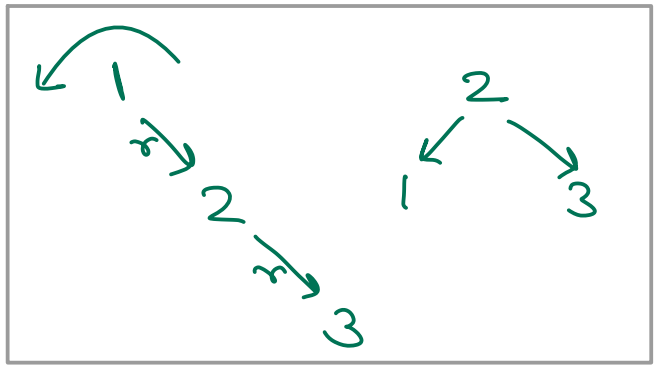
5

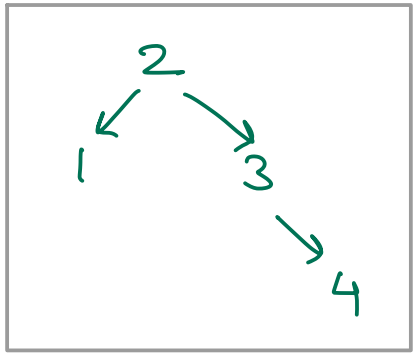
6

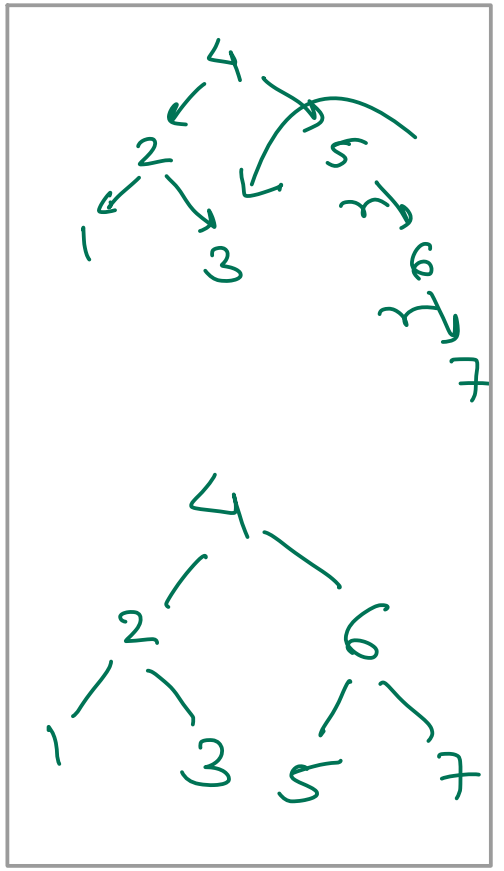
7

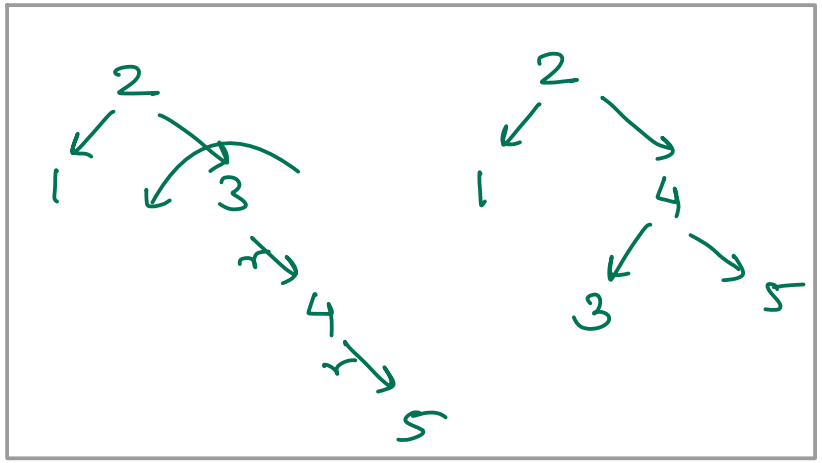


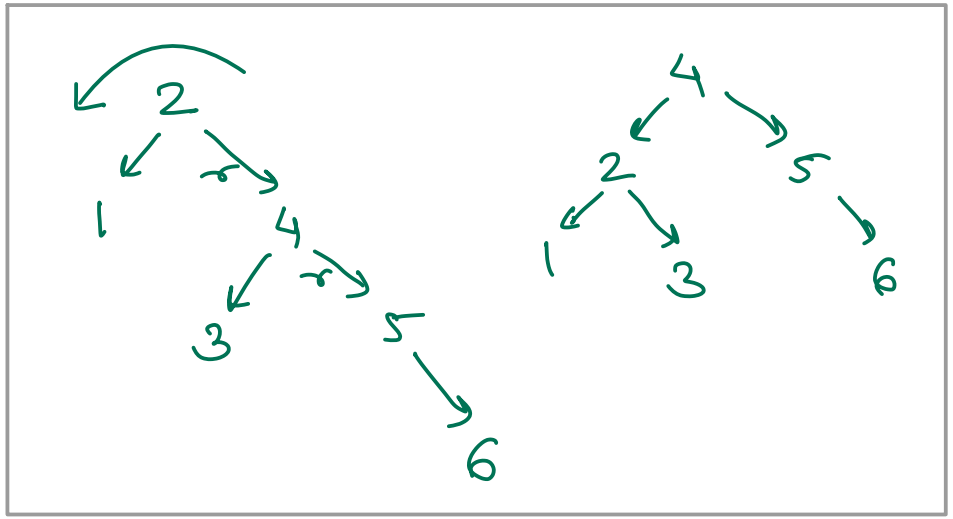










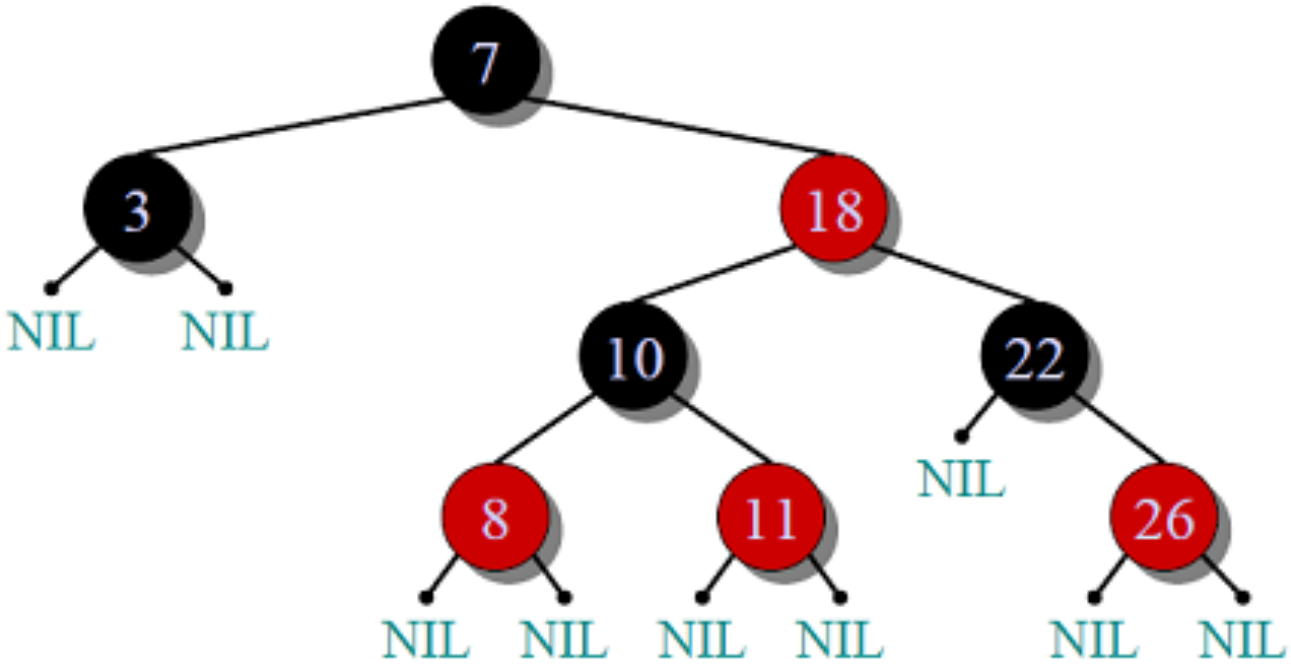


AVL Tree

- 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.

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.



Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>