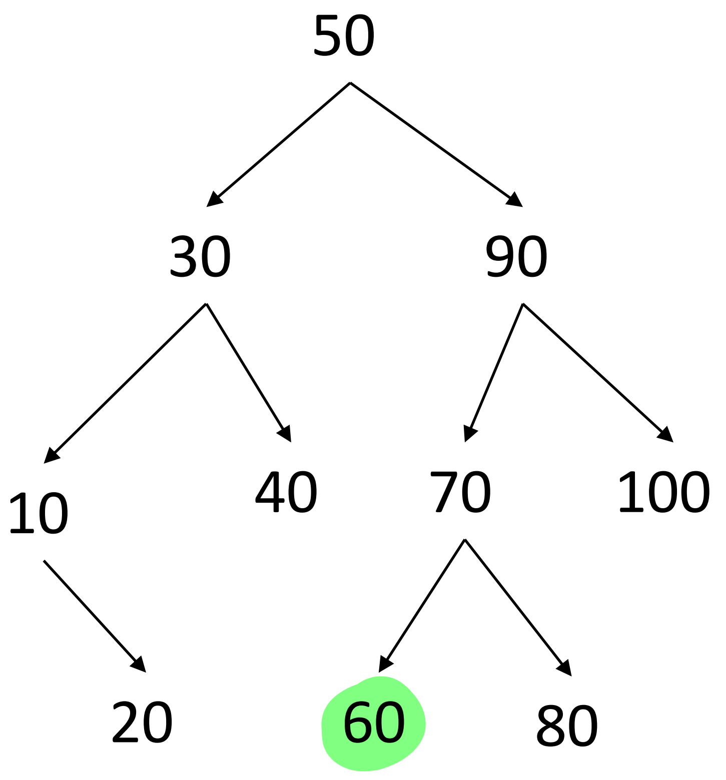
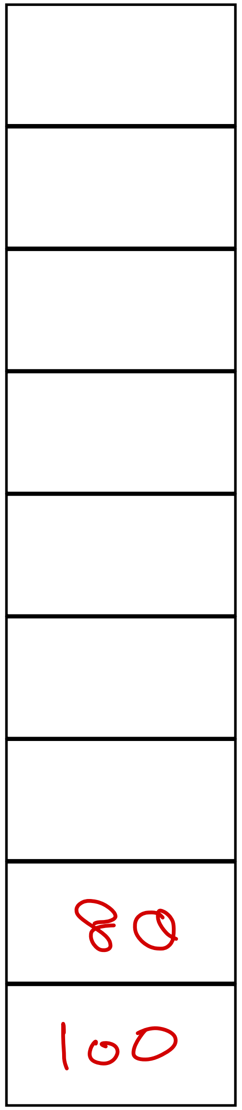
Data Structure & Algorithms

Sunbeam Infotech

Nilesh Ghule

BST – Non-Recursive Algorithm – DFS – depth wise.



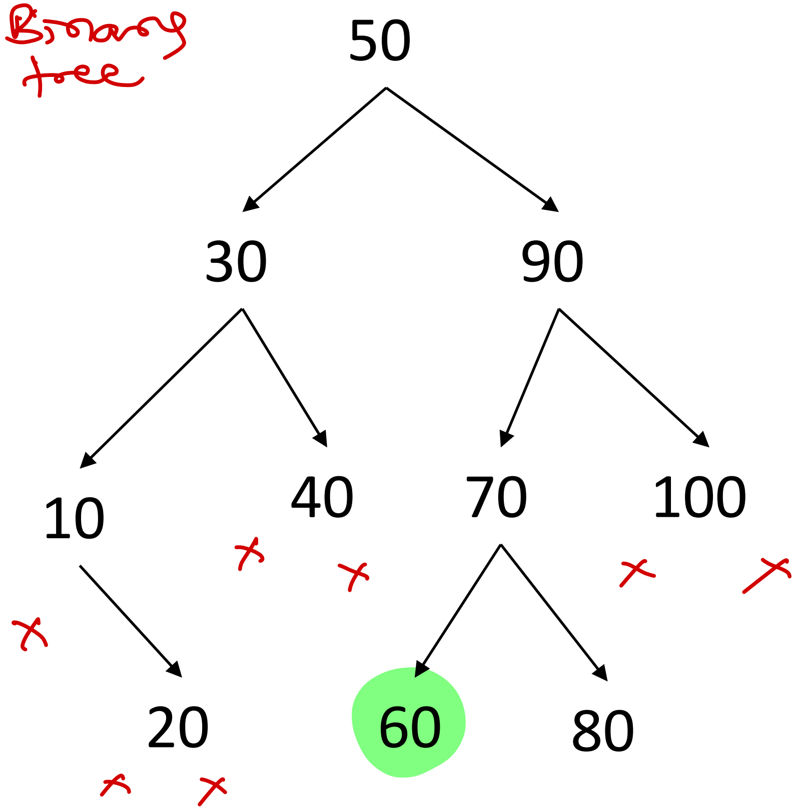


push root on stack;
while stack is not empty
    pop a node from stack;
    visit the node (compare).
    if node has right child,
        push child in stack;
    if node has left child,
        push child in stack;

50 30 10 20 40
90 70 60

BST – Non-Recursive Algorithm – BFS - levelwise search.

Binary tree

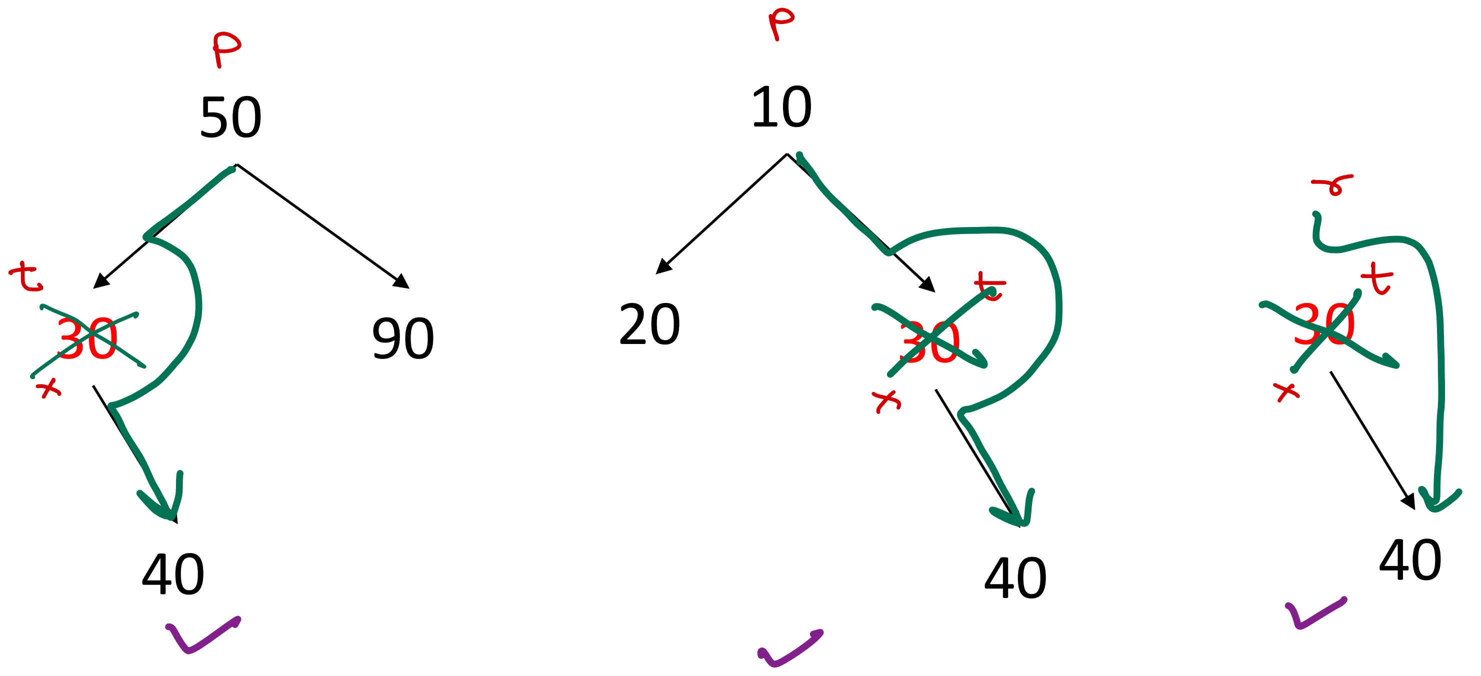


								80

push root on queue;
while queue is not empty
    pop a node from queue.
    visit the node (compare).
    if node has left child,
        push child in queue.
    if node has right child,
        push child in queue.

50 30 90 10 40
70 100 20 60

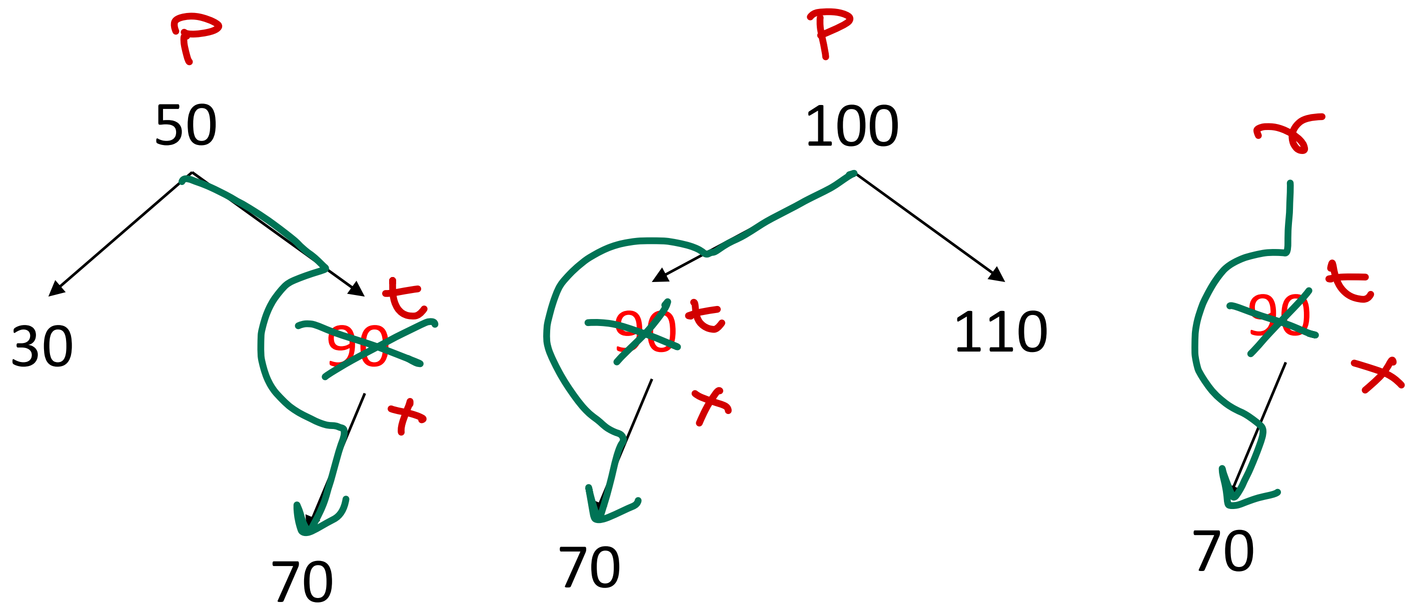
BST – Delete Node → trav.left == null.




```
if(trav.left == null){
    if(trav == root)
        root = trav.right;
    else if(trav == p.left)
        p.left = trav.right;
    else
        p.right = trav.right;
}
```

BST – Delete Node

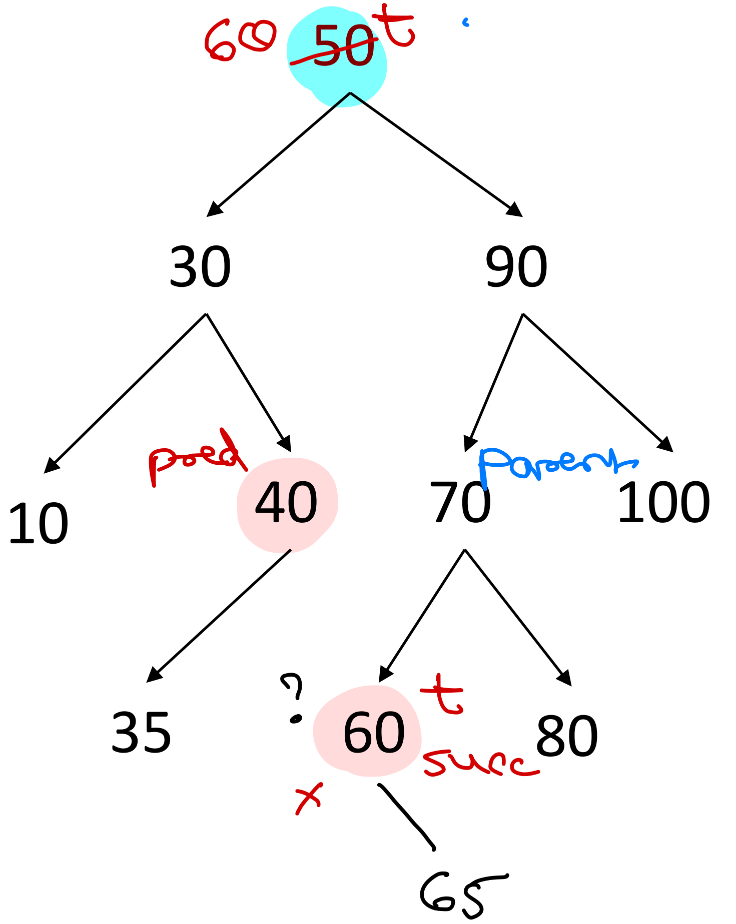
t.right == null




```
if (t.right == null) {
    if (t == root)
        root = t.left;
    else if (t == p.left)
        p.left = t.left;
    else
        p.right = t.left;
}
```

BST – Delete Node

t.left != null && t.right != null.



10 30 35 40 50 60 65 70 80 90 100

① find inorder succ with its parent.

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

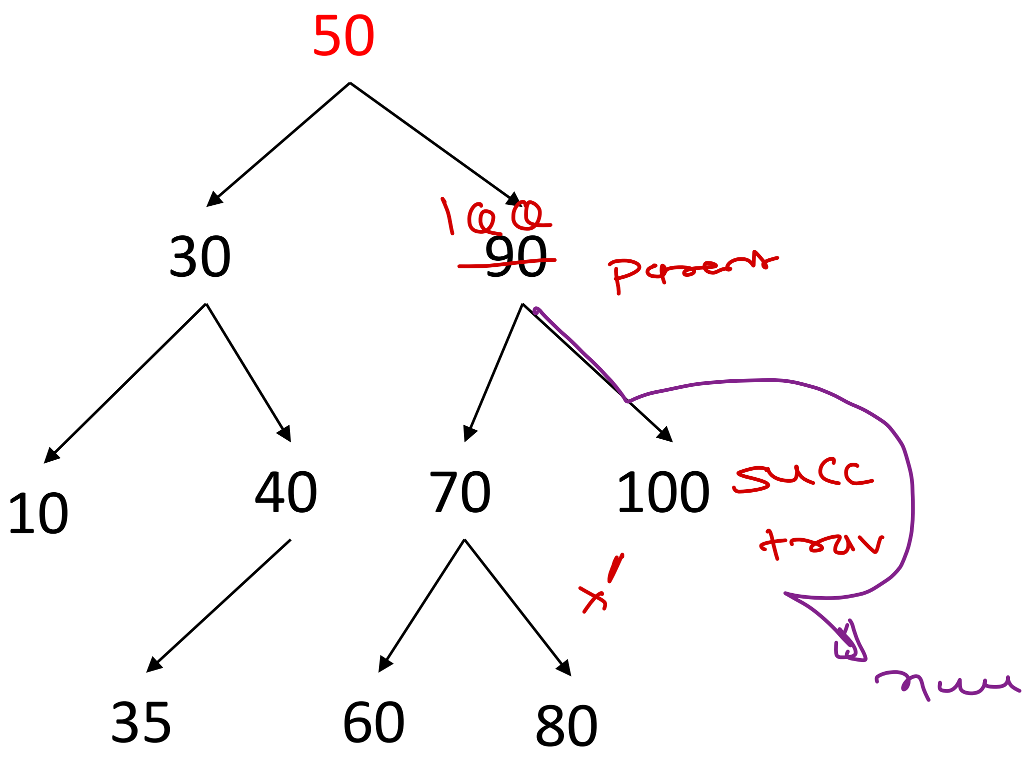
② replace node data with succ data;

```
t.data = succ.data;
```

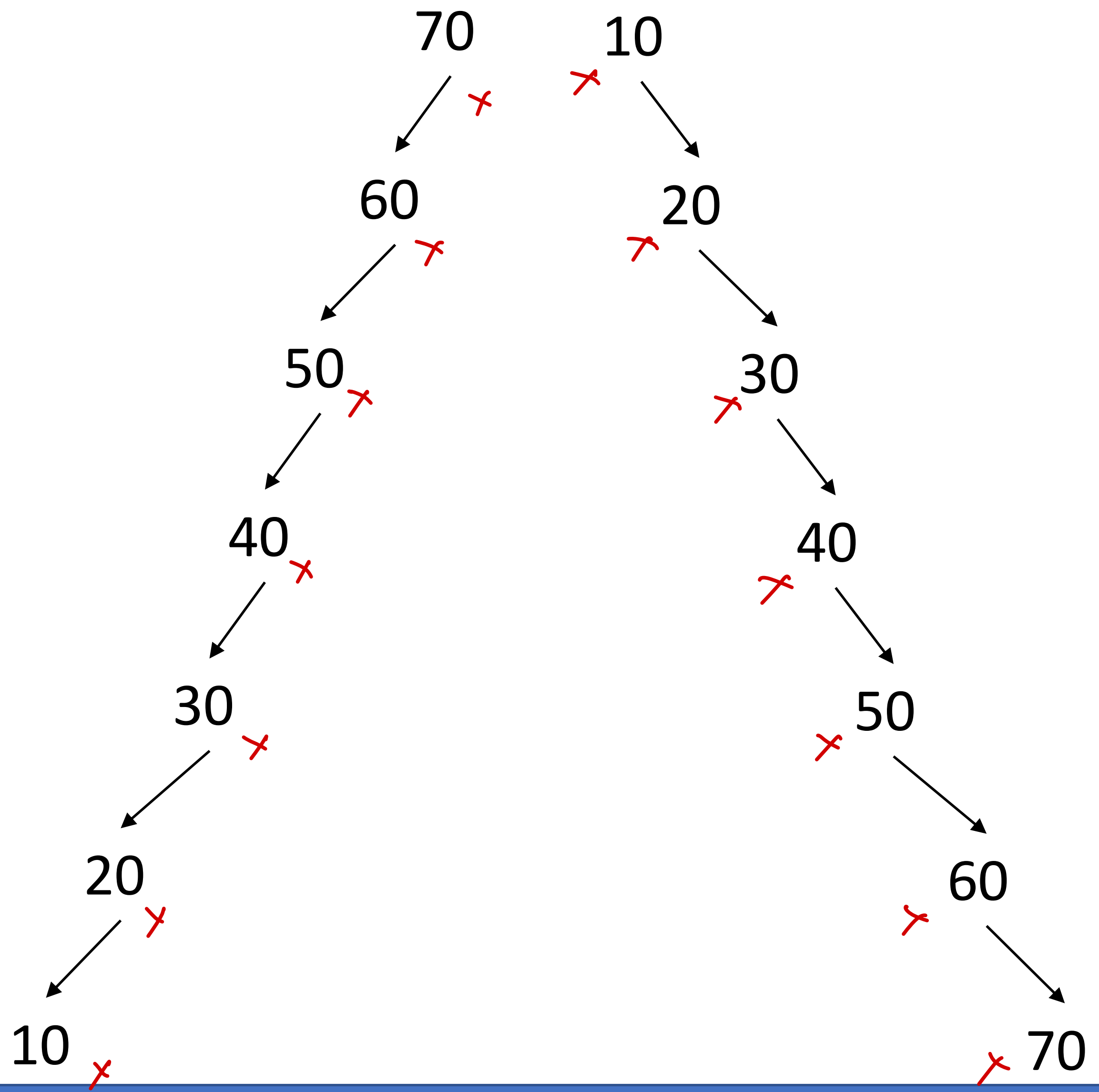
③ delete succ.

```
t = succ;
if(t.left == null){
    ...
}
```

BST – Delete Node



Skewed Binary Tree

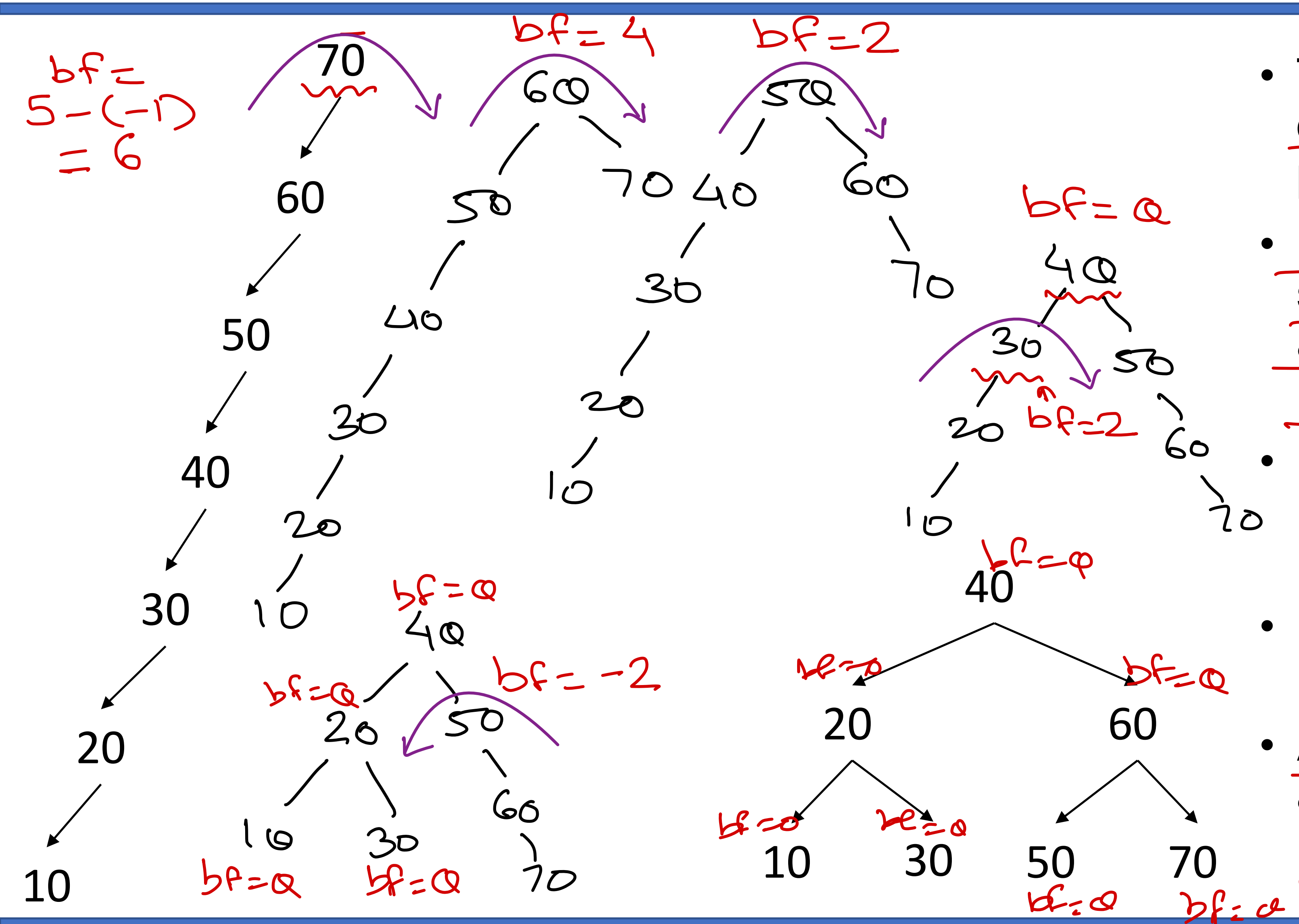


- In Binary tree if only left or only right links are used, tree grows only on one side. Such tree is called as skewed binary tree.
  - Left skewed binary tree
  - Right skewed binary tree

height

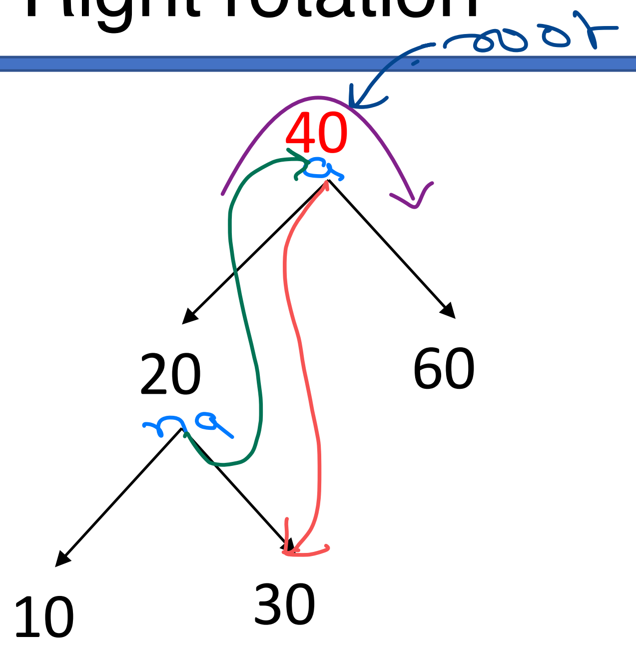
- Time complexity of any BST is O(h).
- Such tree have maximum height i.e. same as number of elements.
- Time complexity of searching in skewed BST is O(n).

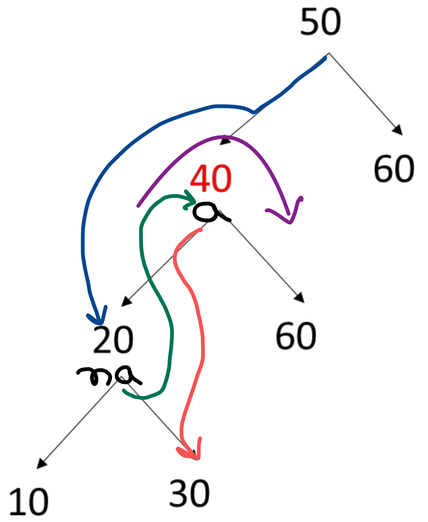
Balanced BST

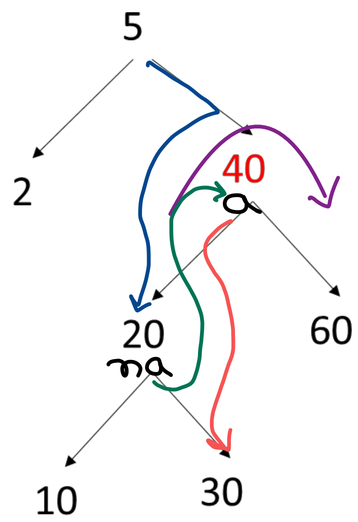


- To speed up searching, height of BST should minimum as possible.
- If nodes in BST are arranged so that its height is kept as less as possible, is called as Balanced BST.
- Balance factor of node.
  - = Height of left sub tree – Height of ~~left~~ right sub tree
- In balanced BST, BF of each node is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

Right rotation

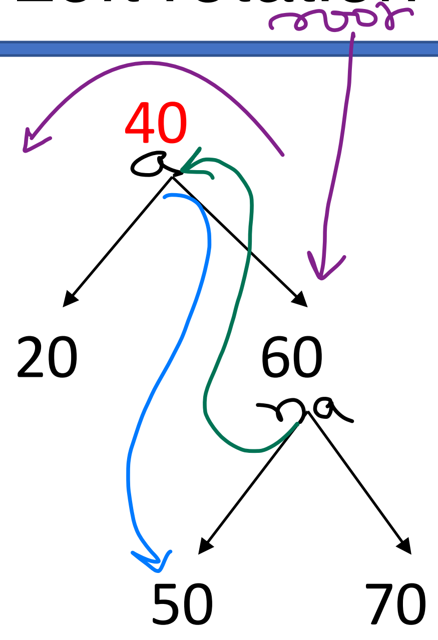




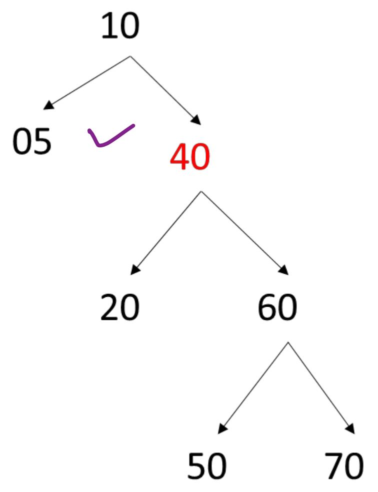


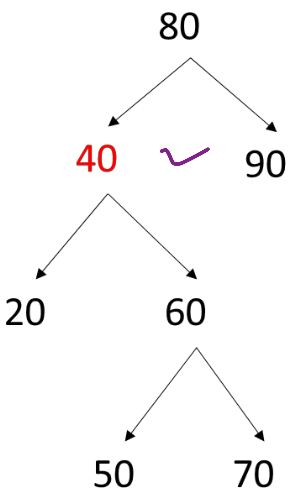

```
① na = a.left;
② a.left = na.right;
③ na.right = a;
④ if(a == root)
       root = na;
   else if(a == p.left)
       p.left = na;
   else
       p.right = na;
```

Left rotation

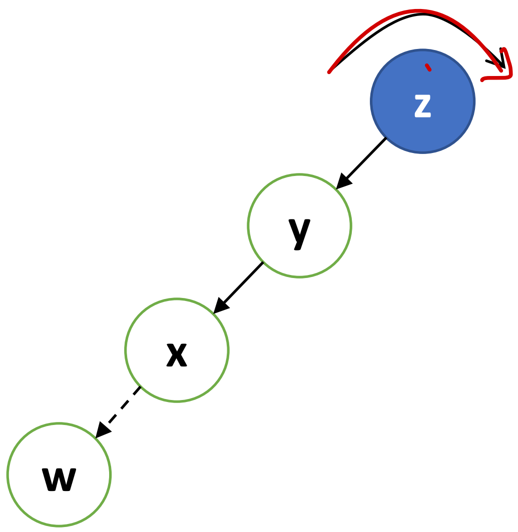


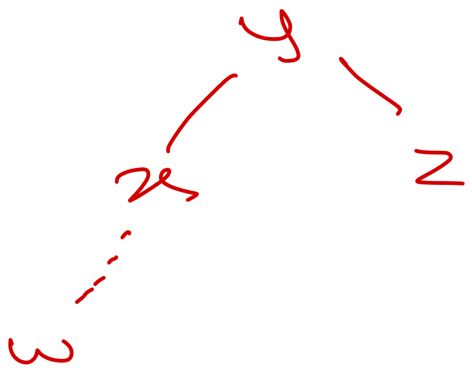

```
na = a.right;
a.right = na.left;
na.left = a;
if (a == root)
    root = na;
else if (a == p.left)
    p.left = na;
else
    p.right = na;
```





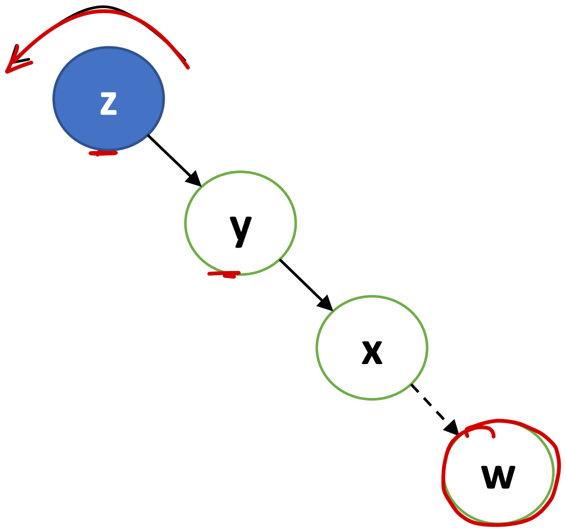
Rotation cases



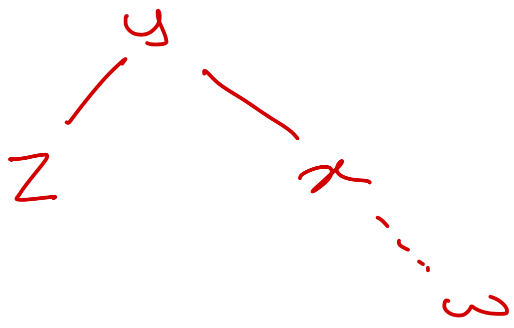


Left-Left case

Rotation cases

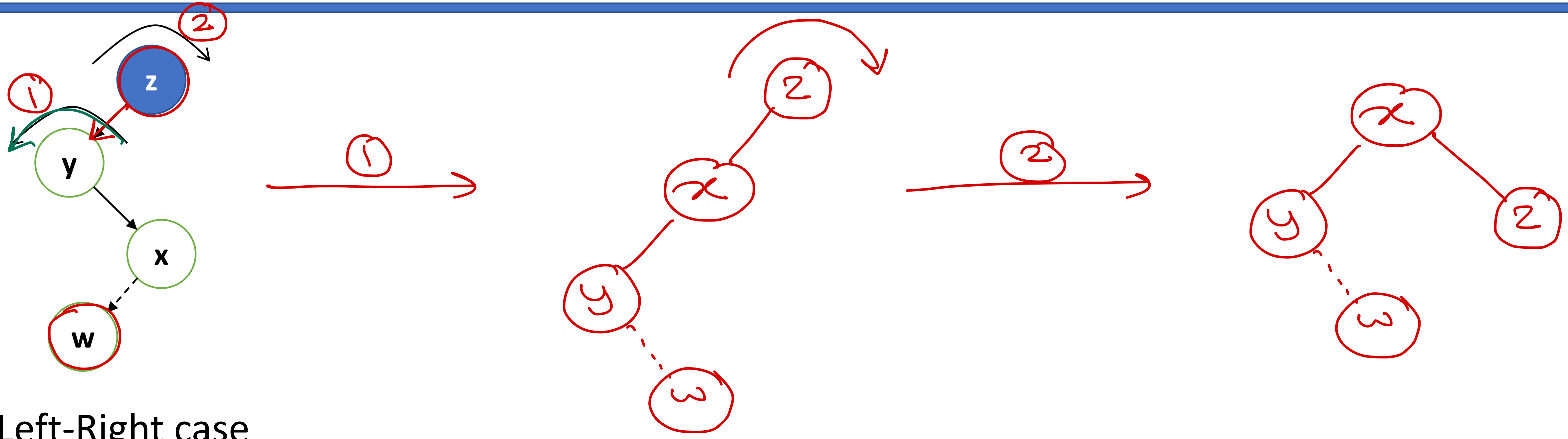


Right-Right case



Rotation cases

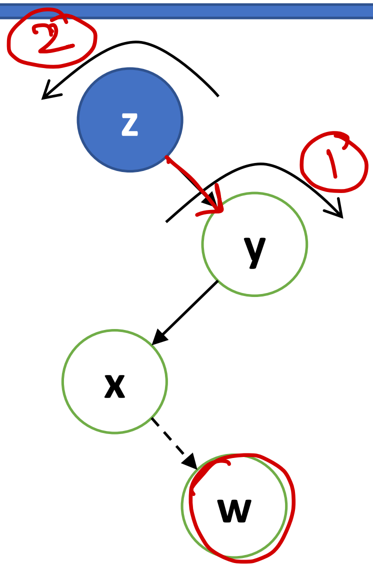
bf = 1 − 0 = 1



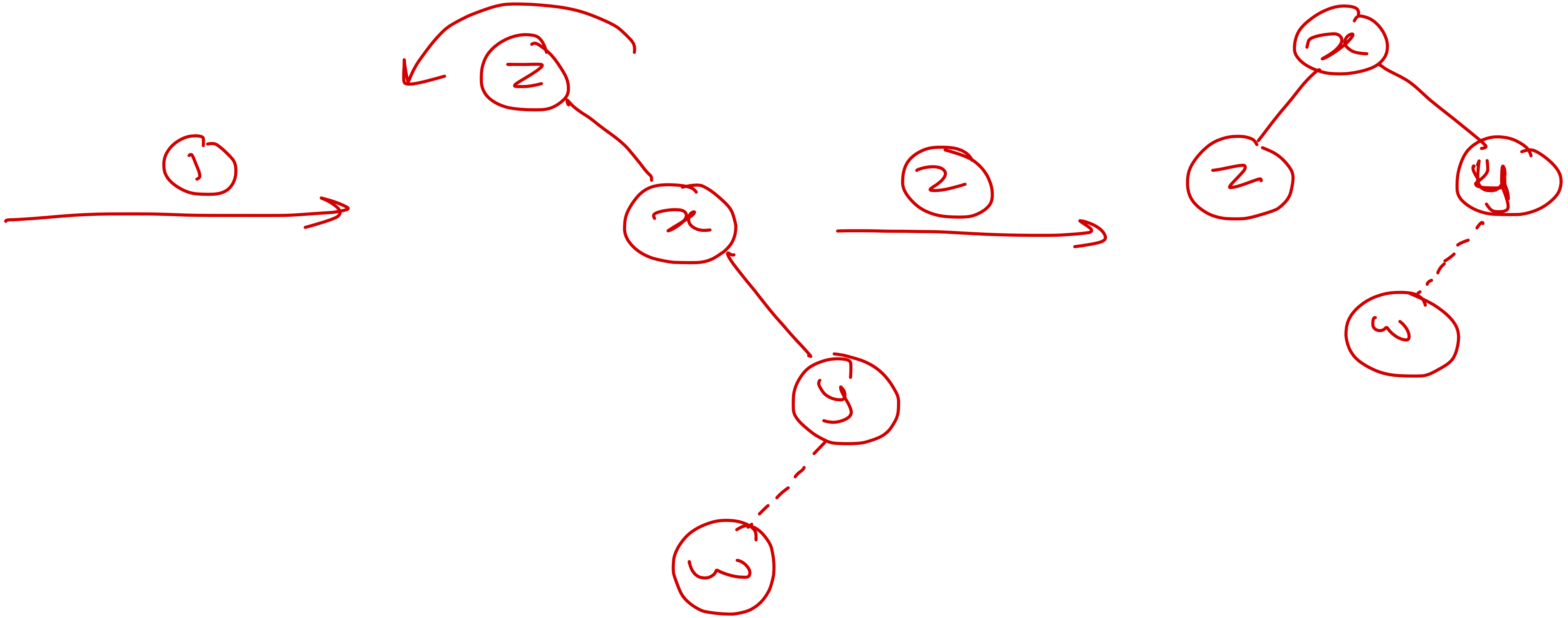
Left-Right case

Rotation cases

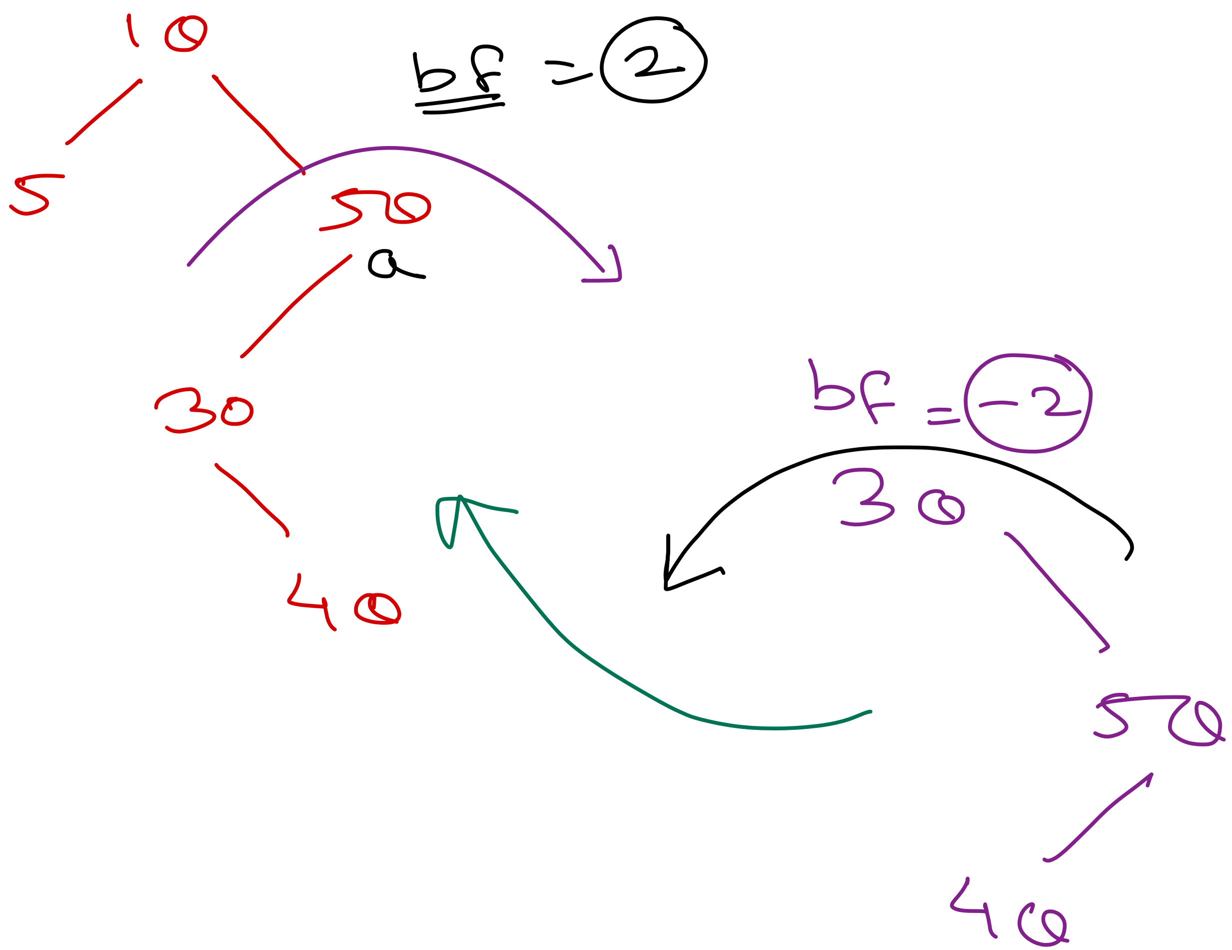
$bf = 0 - 1 = -1$



Right-Left case

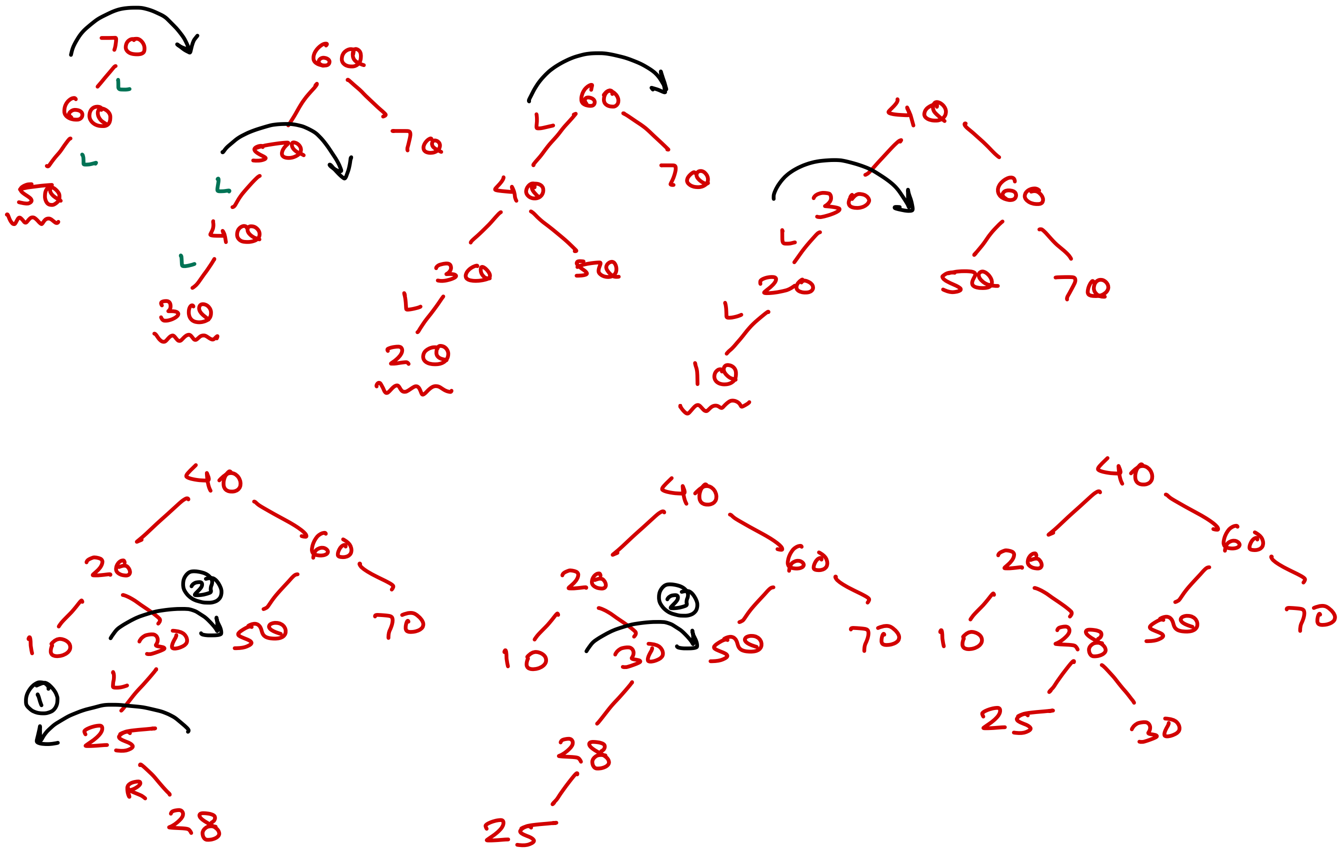


$bf = 0 - 2 =$



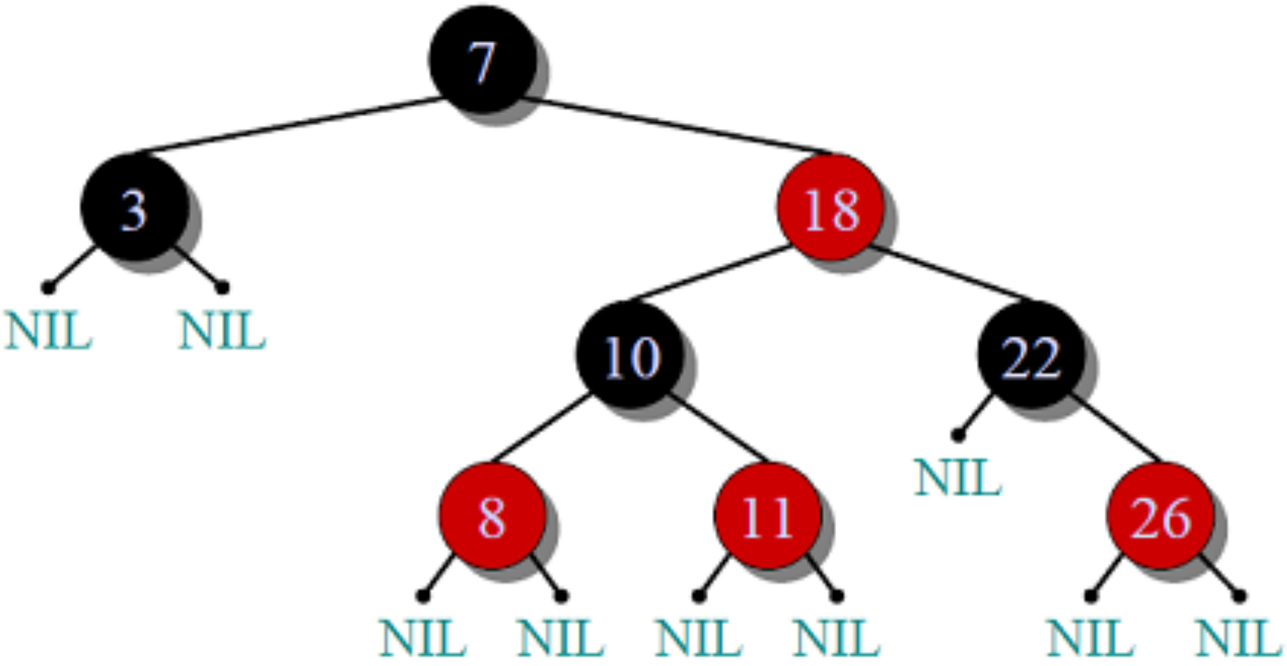
AVL Tree

- 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. $-1 \leq bf \leq 1$
- 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.

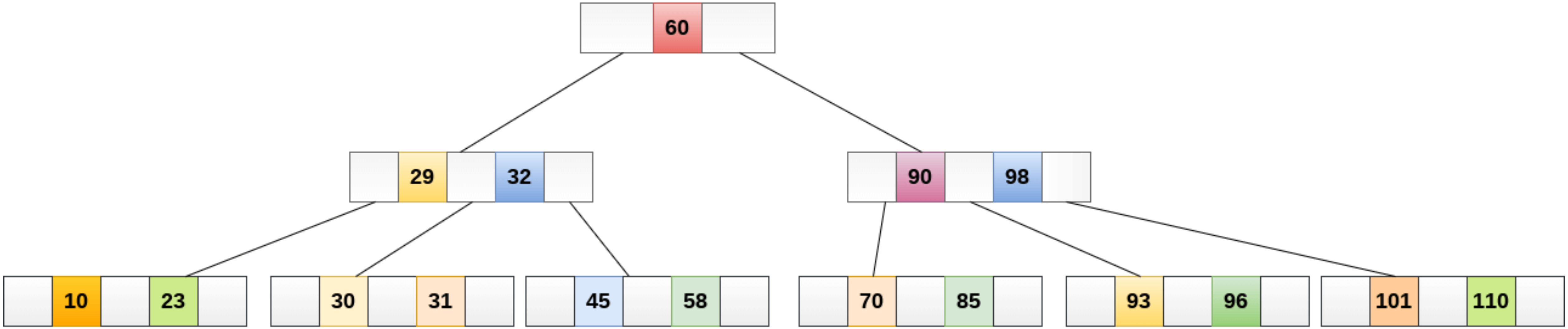


Red & Black tree

- 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.

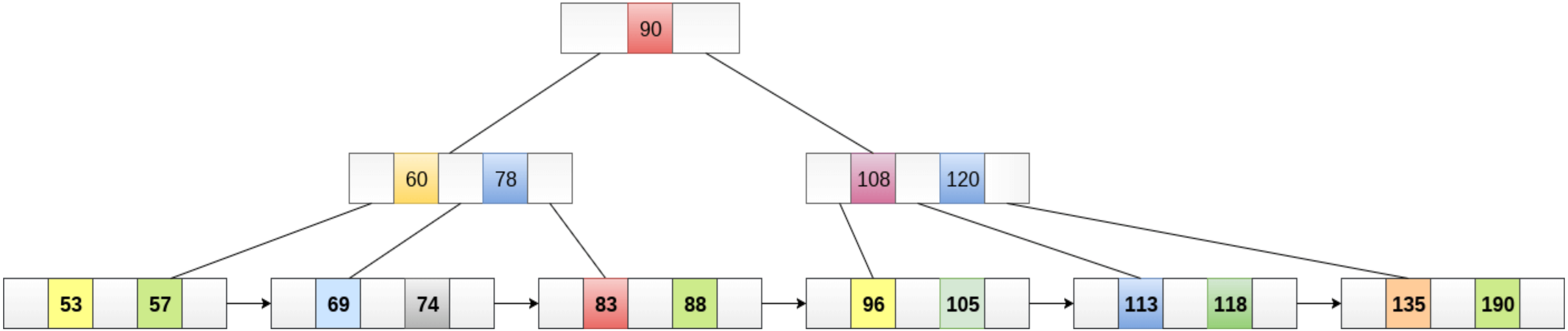


B Tree



- A B-Tree of order m can have at most m-1 keys and m children.
- B tree store large number of keys in a single node. This allows storing number of values keeping height minimal.
- Note that in B-Tree all leaf nodes are at same level.
- B-Tree is commonly used for indexing into file systems and databases. It ensures quick data searching and speed up disk access.

B+ Tree



- Extension of B-Tree for efficient insert, delete and search operation.
- Data is stored in leaf nodes only and all leaf nodes are linked together for sequential access.
- Search keys may be redundant.
- Faster searching, simplified deletion (as only from leaf nodes).
- B+Tree is commonly used for indexing into file systems and databases. It ensures quick data searching and speed up disk access.

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>