Data Structure & Algorithms

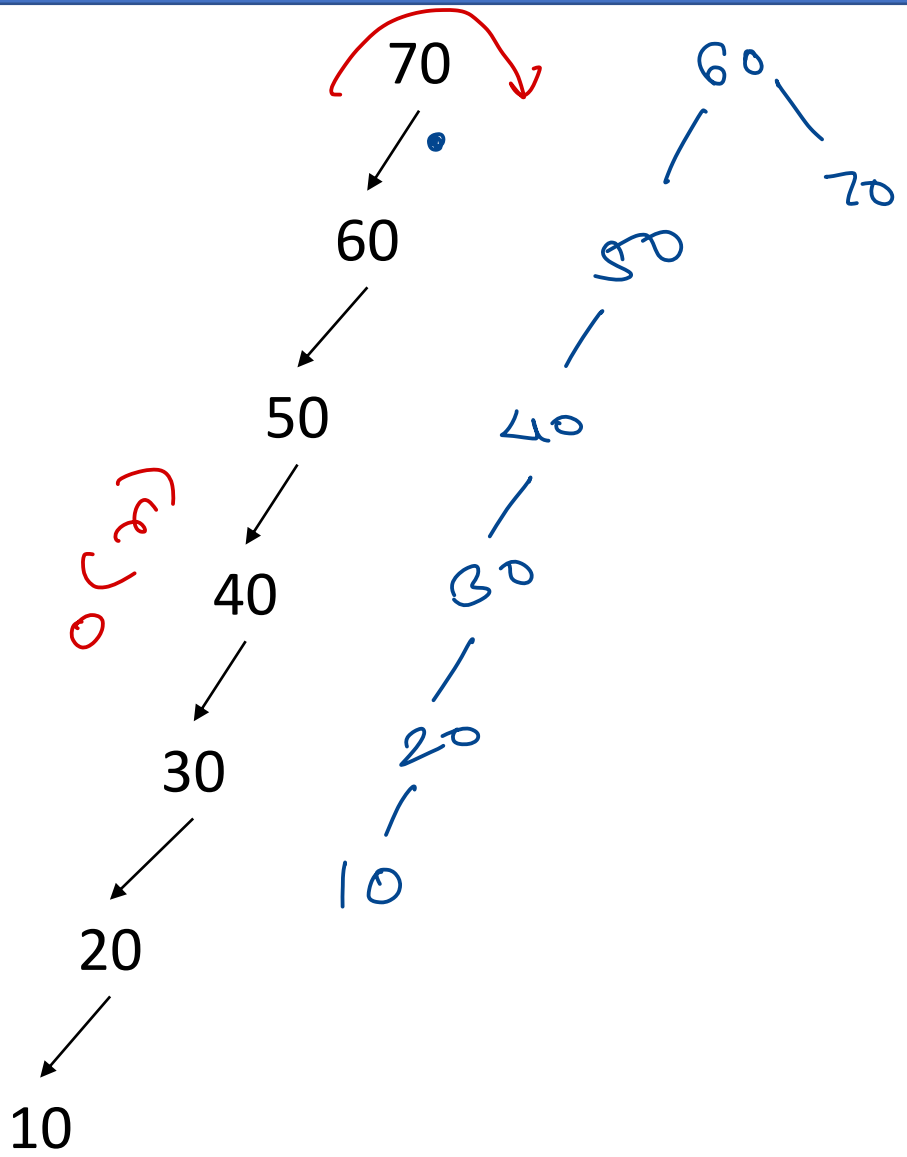
Sunbeam Infotech

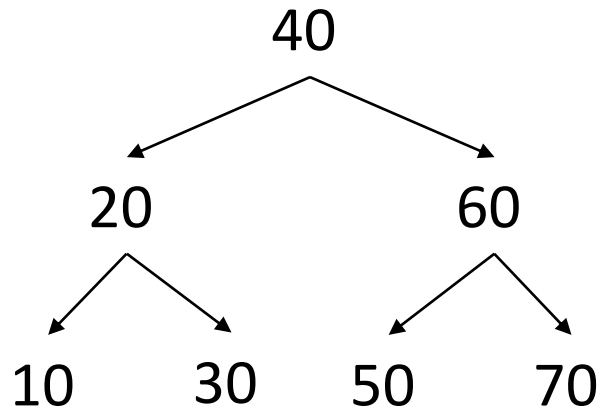
Agenda

- Left & Right rotations
- BST – Balance tree
- BST – Types
  - Skewed Binary Tree
  - AVL Tree
  - R & B Tree
  - Threaded BST
  - Strict/Full Binary Tree
  - Perfect Binary Tree
  - Complete Binary Tree
- Heap
  - Max Heap
  - Min Heap
- Heap operations
  - Make heap
  - Delete from heap
- Heap Sort

Balanced BST

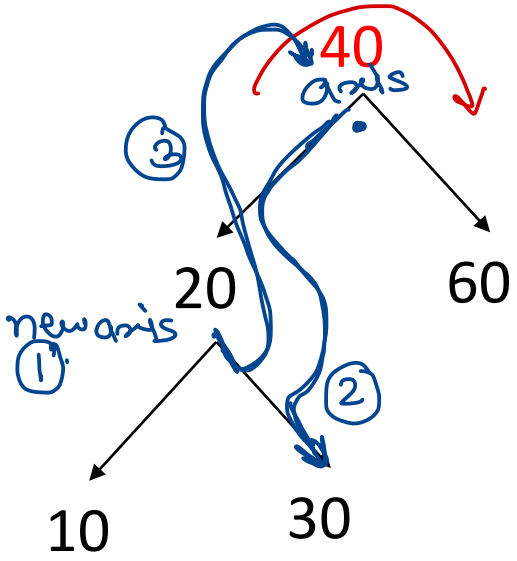
find → O(h)

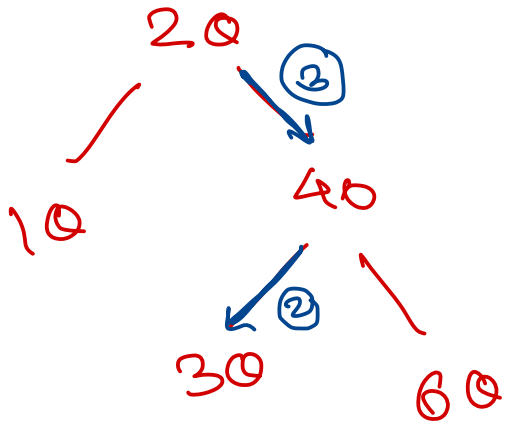


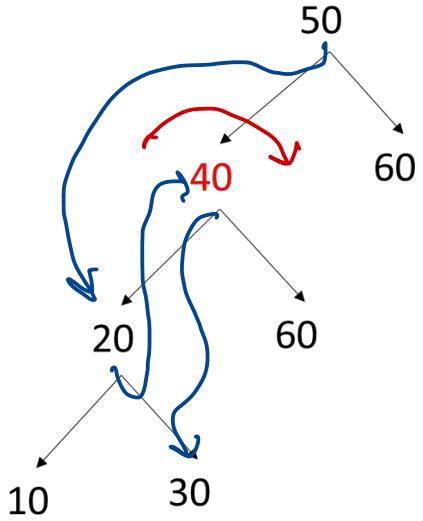


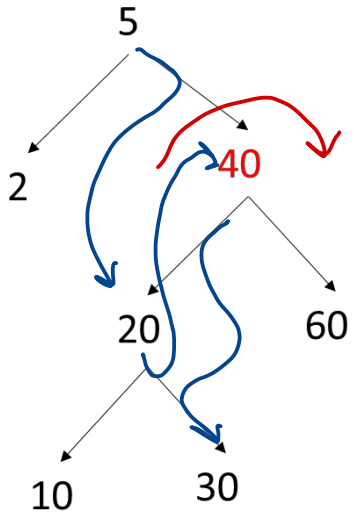
- To speed up searching, height of BST should minimum as possible.
- If nodes in BST are arranged so that its height is kept as less as possible, is called as Balanced BST.
- Balance factor of a node
  - = Height of left sub tree – Height of ~~left~~ right sub tree
- In balanced BST, BF of each node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

Right rotation



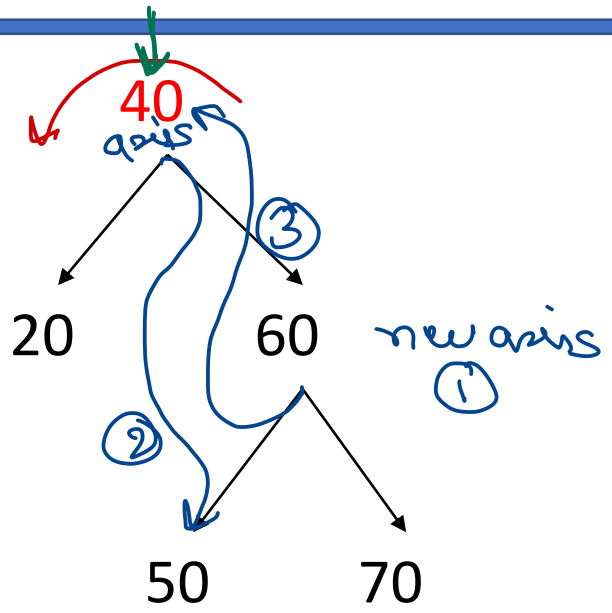


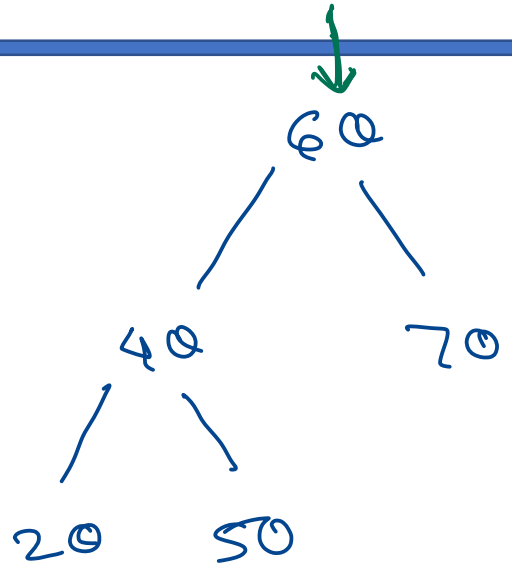


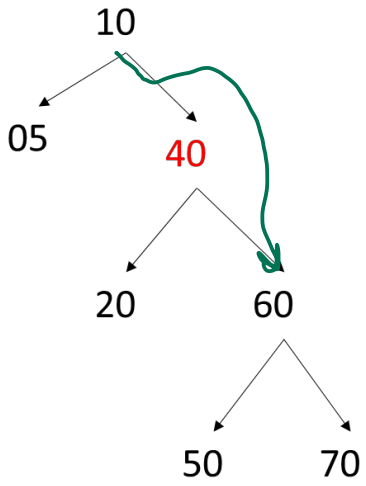


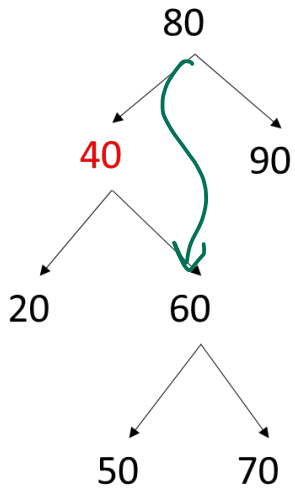

```
① newaxis = axis->left;
② axis->left = newaxis->right;
③ newaxis->right = axis;
④ if (axis == root)
      root = newaxis;
  else if (axis == parent->left)
      parent->left = newaxis;
  else
      parent->right = newaxis;
```

Left rotation









① newaxis = axis→right;

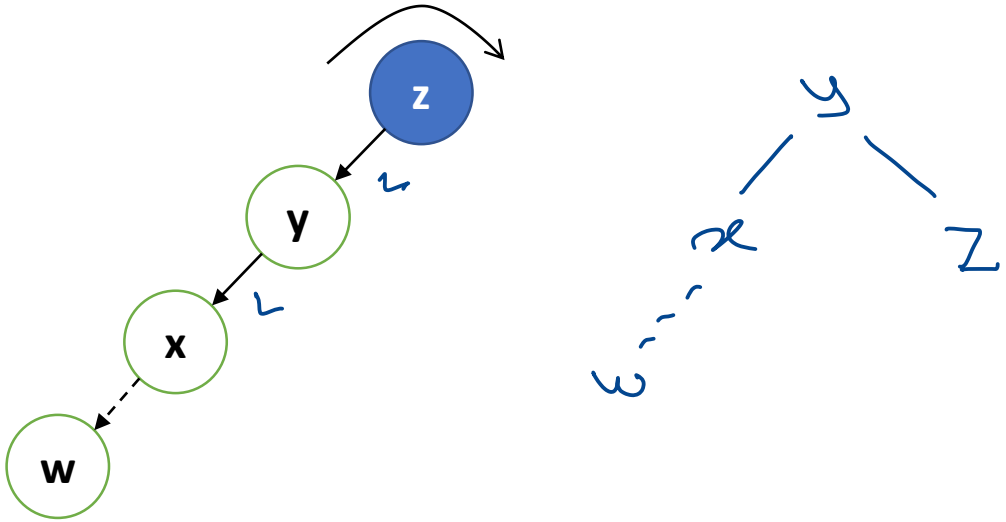
② axis→right = newaxis→left;

③ newaxis→left = axis;

④ if (axis == root)
root = newaxis;
else if (axis == parent→left)
parent→left = newaxis;
else
parent→right = newaxis;

Rotation cases

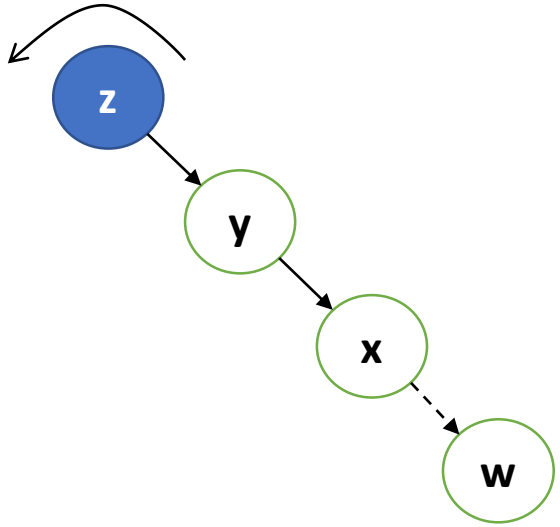
bf = 3

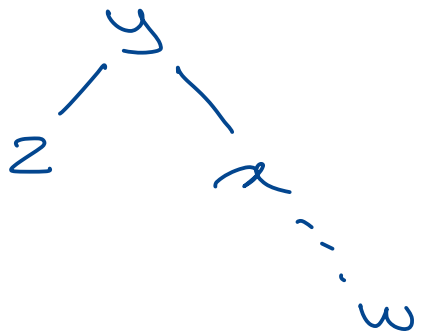


Left-Left case → right rotation on (z)

Rotation cases

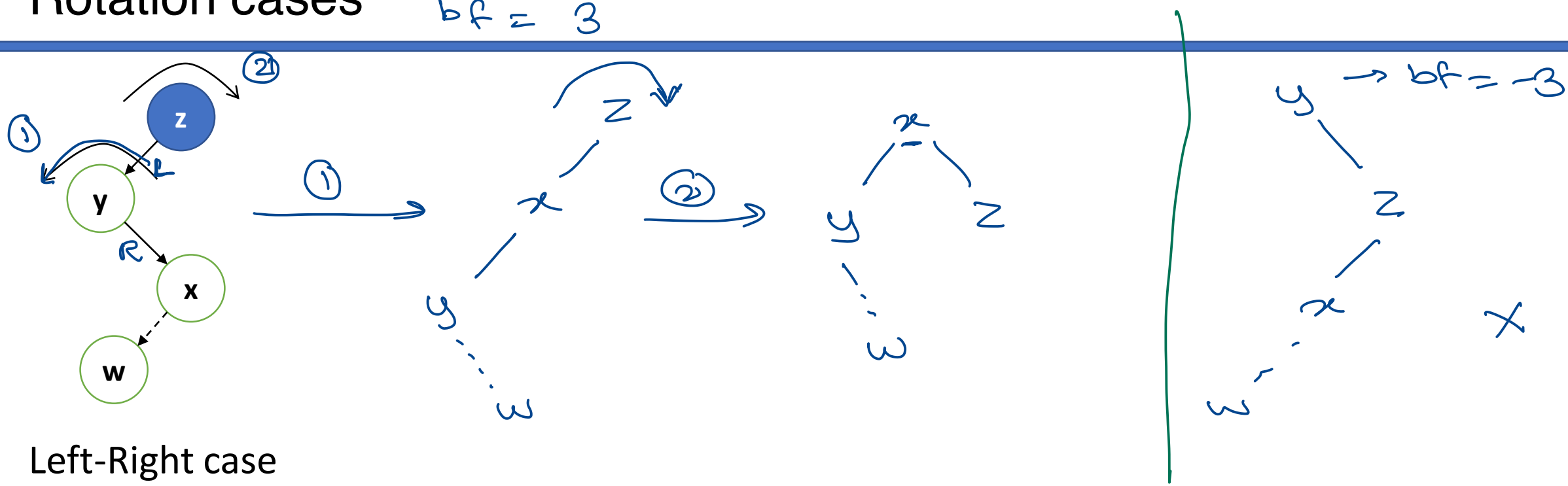
bf = -3





Right-Right case → left rotation on (z)

Rotation cases

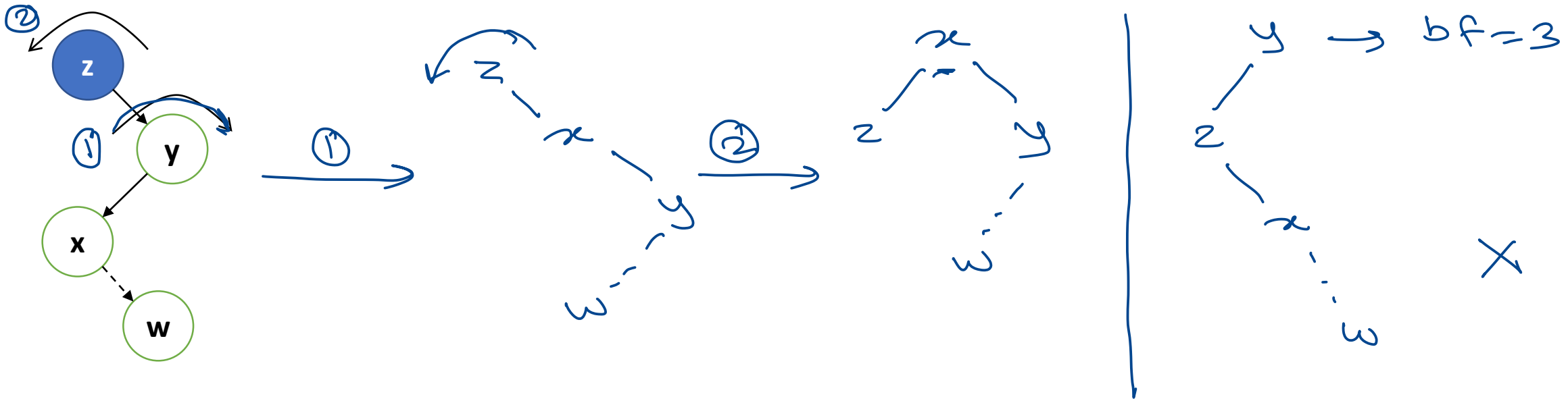


Left-Right case

1. left rotation on y
2. right rotation on z

Rotation cases

bf = -3



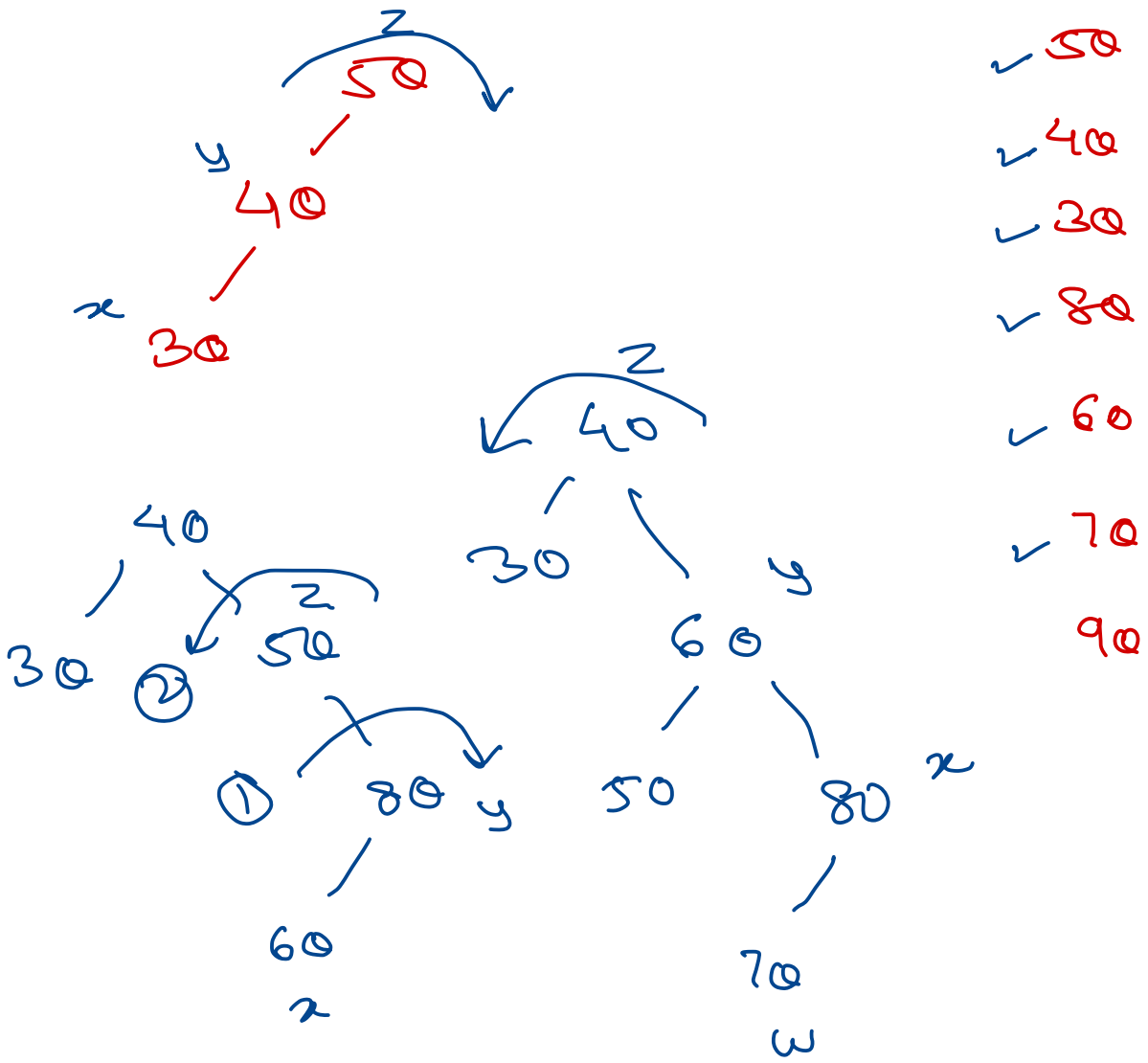
Right-Left case

① right rotation on y

② left rotation on z

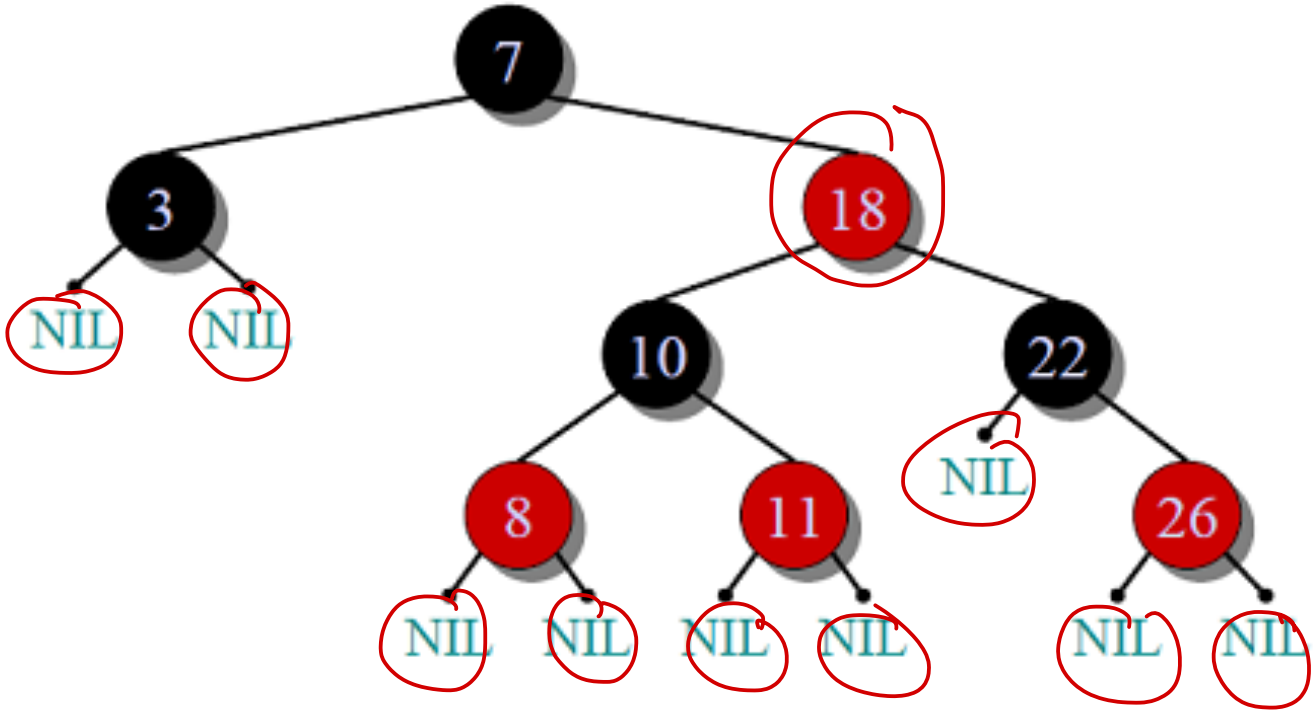
AVL Tree

- 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.

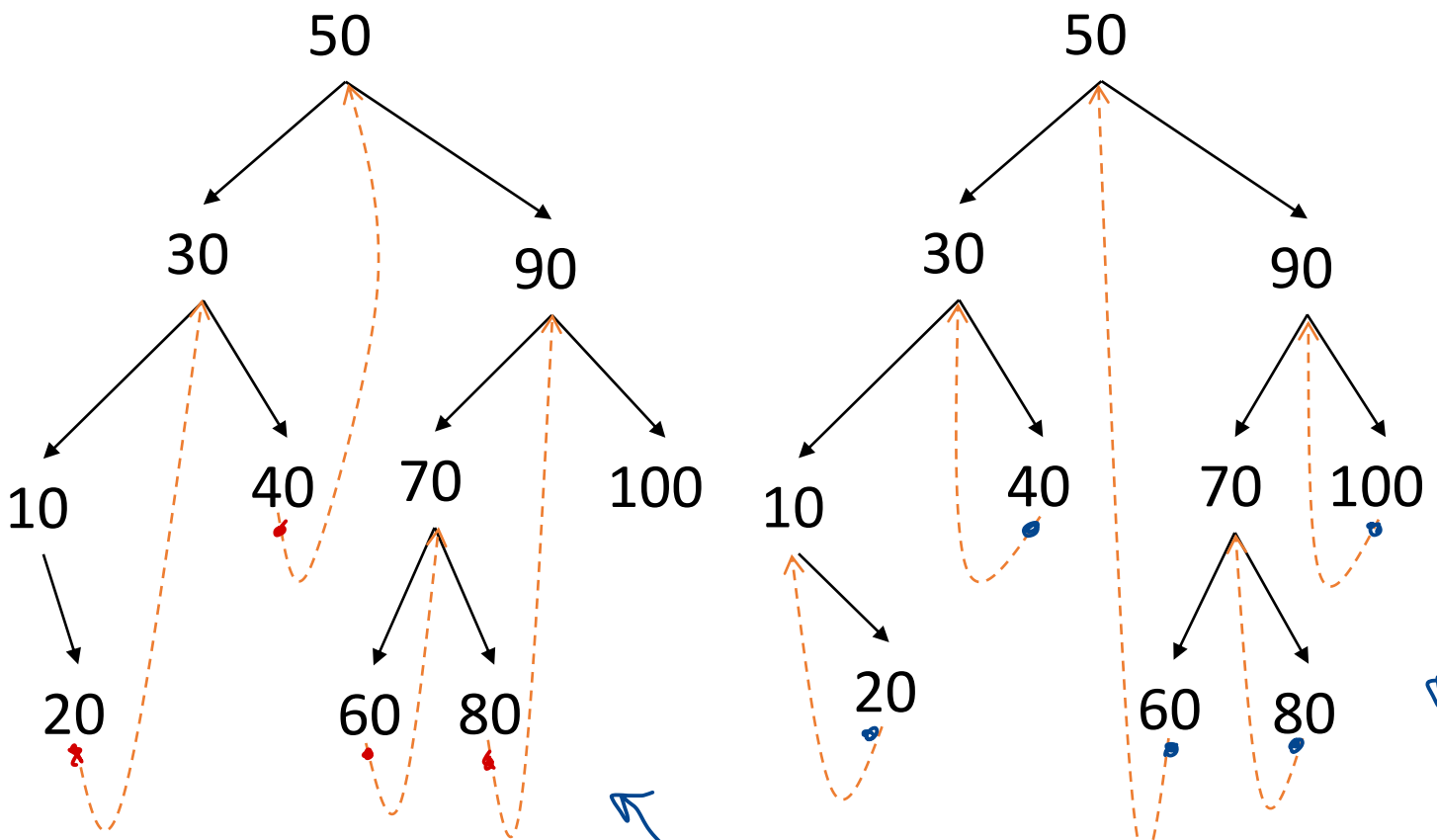


Red & Black tree

- 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.

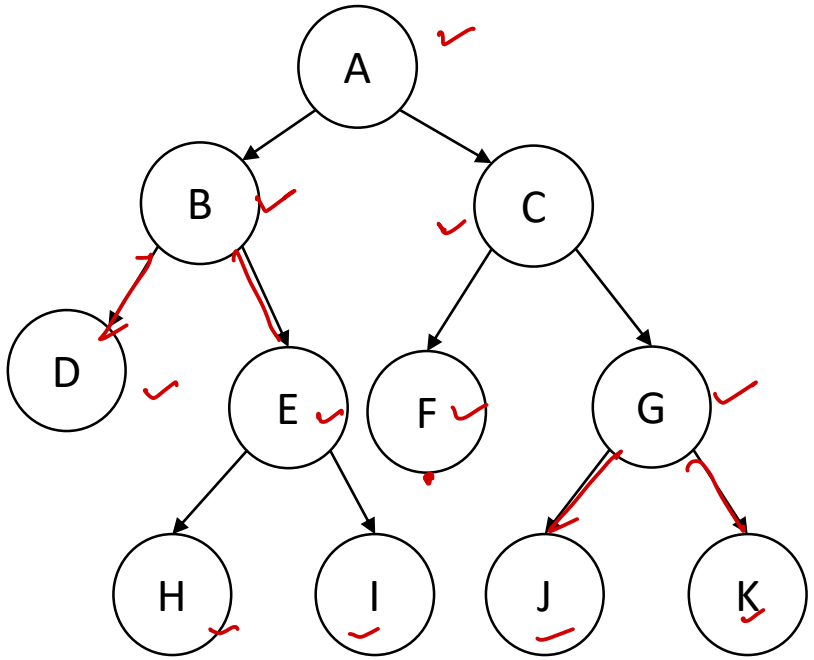


Threaded BST



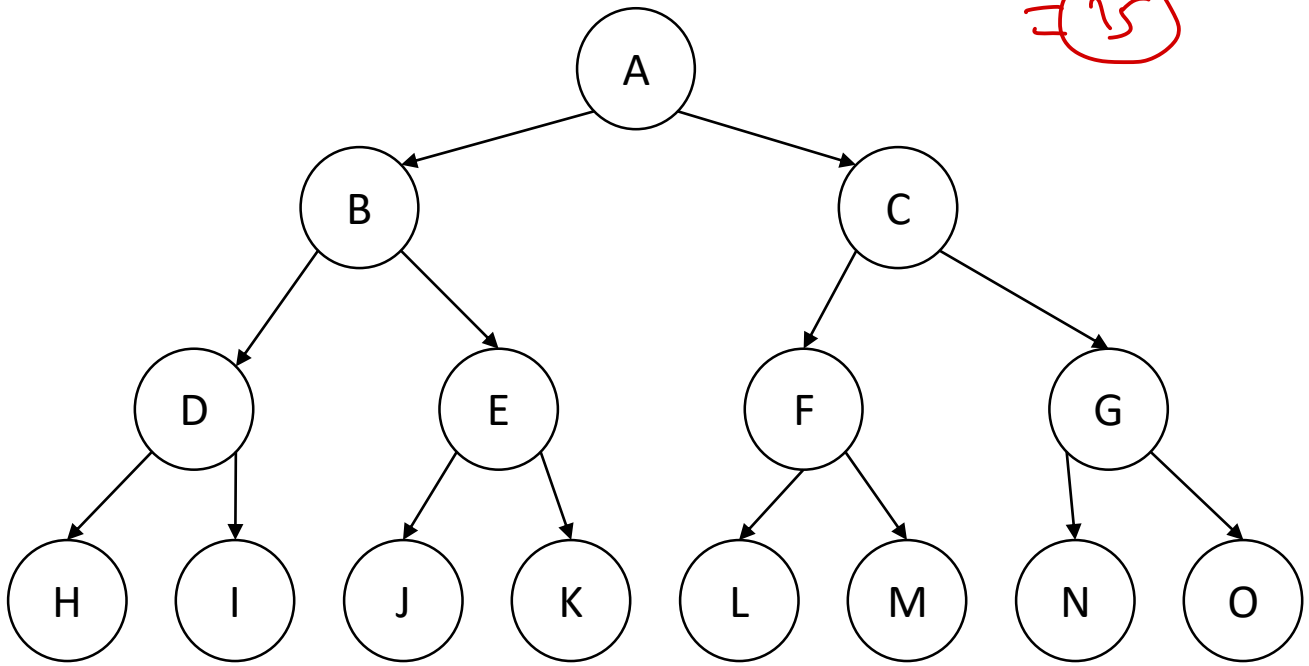
- Typical BST in-order traversal involves recursion or stack. It slows execution and also need more space.
- Threaded BST keep address of in-order successor or predecessor addresses instead of NULL to speed up in-order traversal (using a loop).
- Left threaded BST
- Right threaded BST
- In-threaded BST

Strict/Full Binary Tree



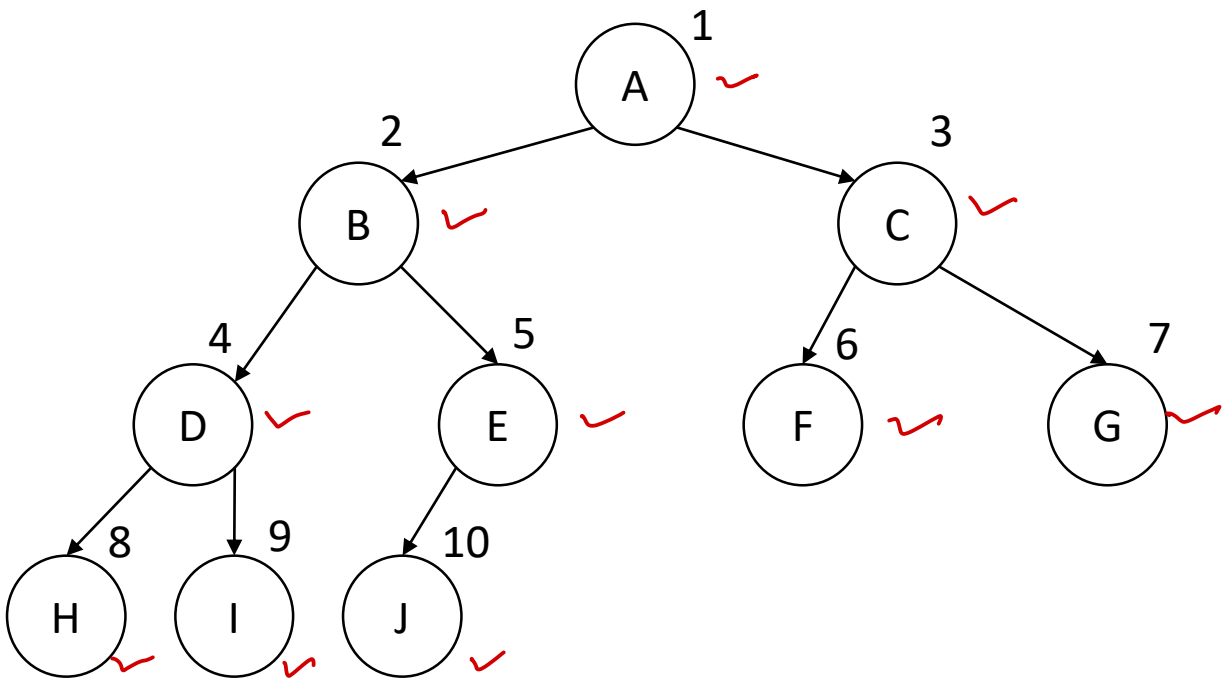
- Binary tree in which each non-leaf node has exactly two child nodes.

Perfect Binary Tree



- Binary tree which is full for the given height i.e. contains maximum possible nodes.
- Number of nodes = $2^h - 1$

Complete Binary Tree and Heap



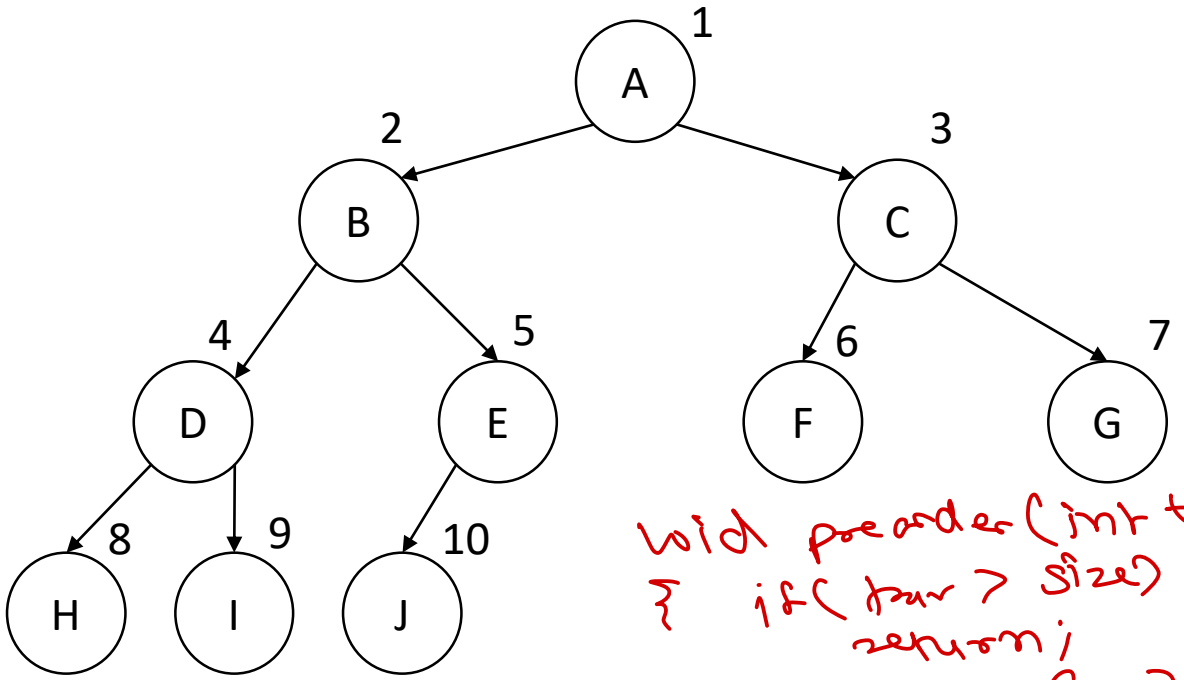
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- Heap is array implementation of complete binary tree.
- Parent child relation is maintained through index calculations
  - parent index = child index / 2
  - left child index = parent index * 2
  - right child index = parent index * 2 + 1

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10

Complete Binary Tree and Heap

max Heap
min Heap

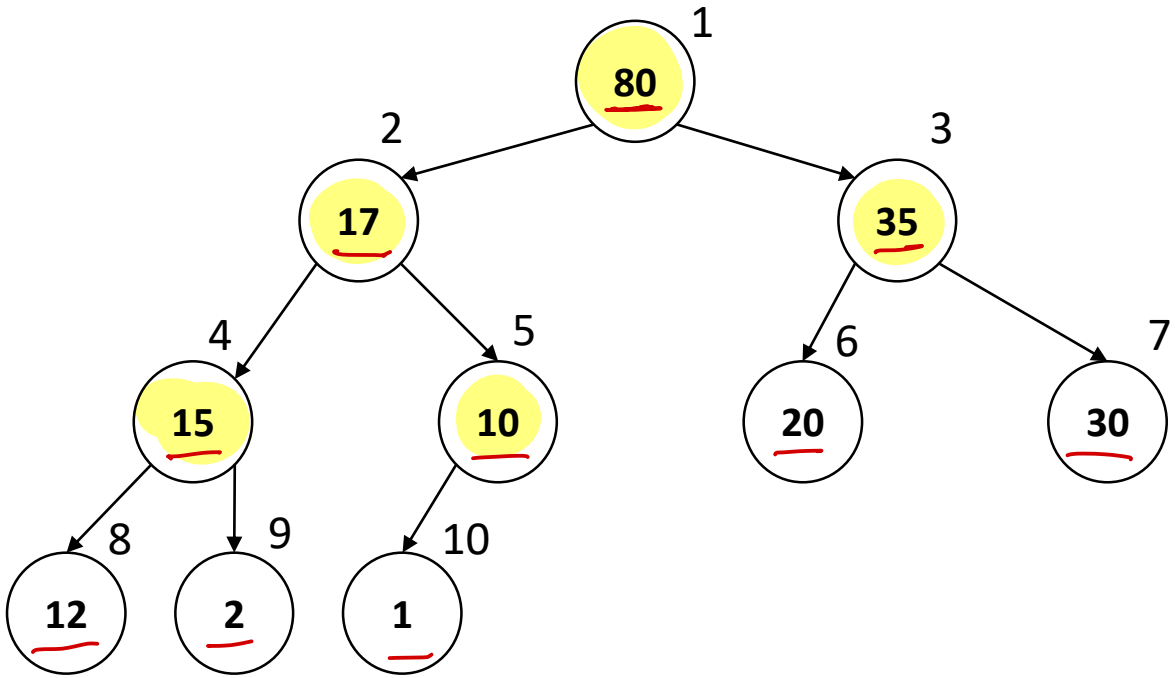



```
void preorder(int trav)
{   if( trav > size)
        return;
    cout<< arr[trav];
    preorder( trav*2);
    preorder( trav*2+1);
}
```

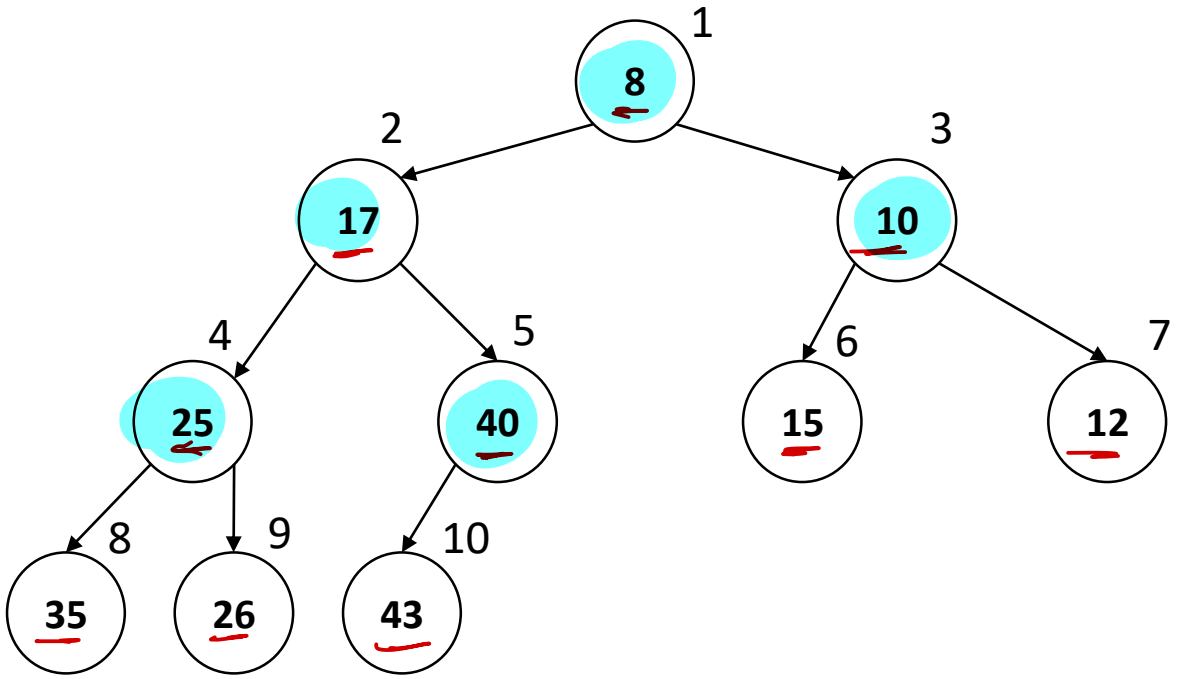
1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- Heap is array implementation of complete binary tree.
- Parent child relation is maintained through index calculations
  - parent index = child index / 2
  - left child index = parent index * 2
  - right child index = parent index * 2 + 1

Max Heap & Min Heap



- Max heap is a heap data structure in which each node is greater than both of its child nodes.

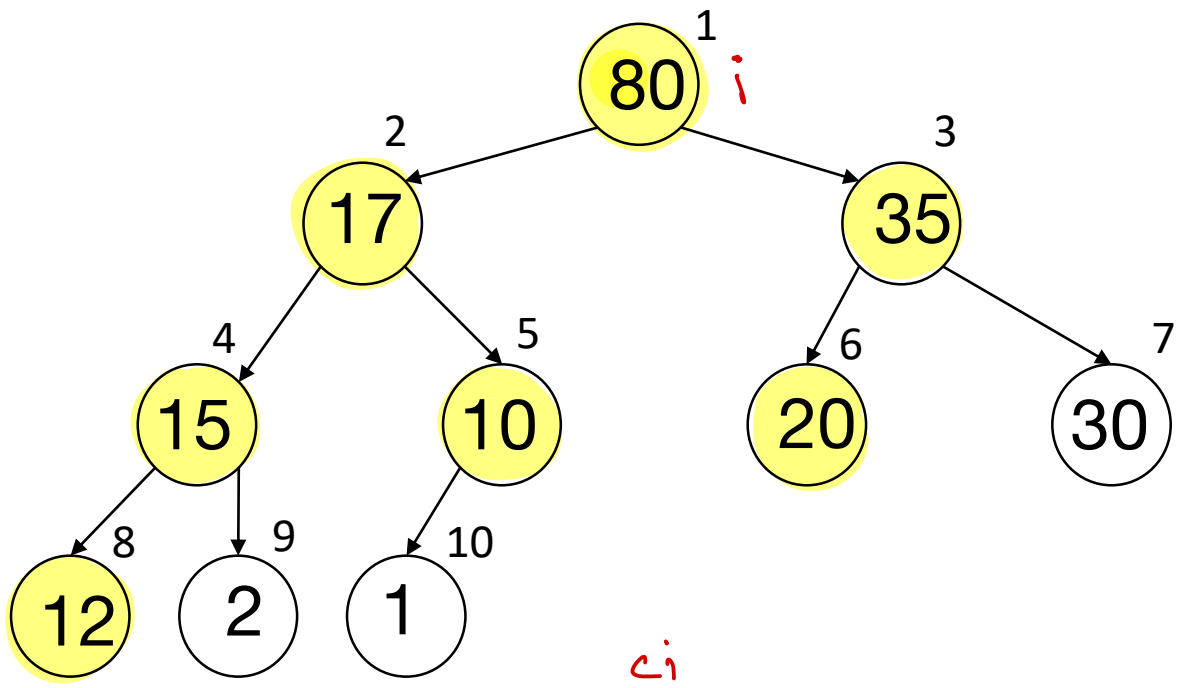


- ~~Max~~ min heap is a heap data structure in which each node is smaller than both of its child nodes.

max or min heaps are used to implement priority queue.

Time complexity → O(log n) → push, pop

Max Heap – Initialize

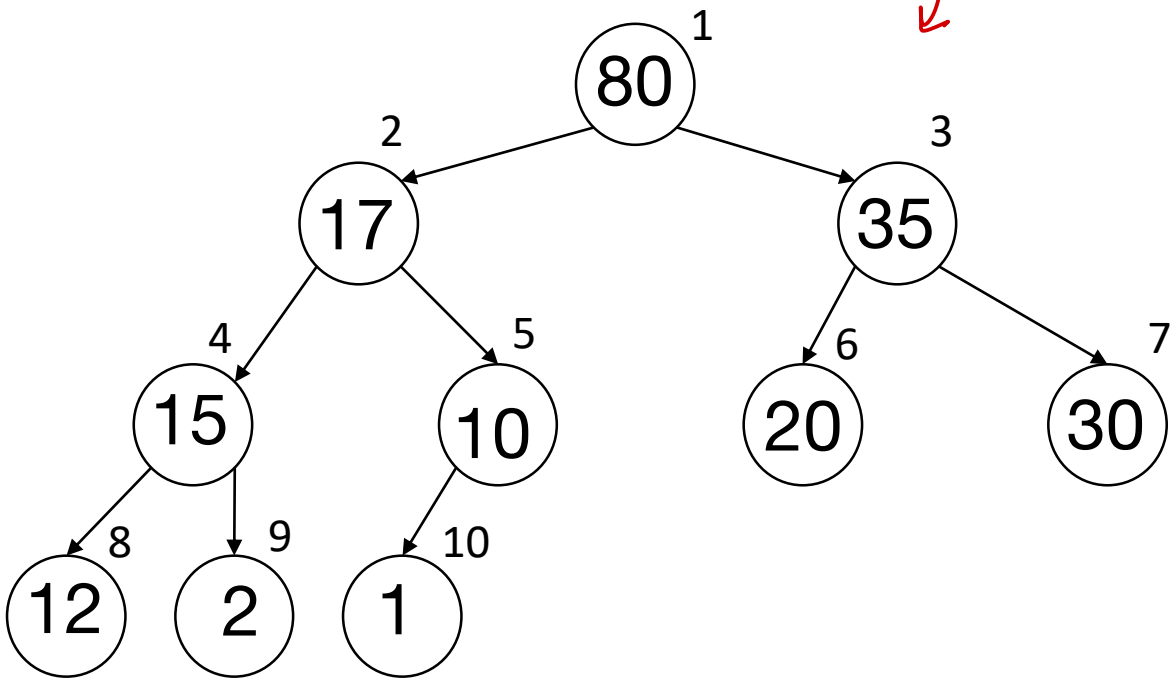


20	12	35	15	10	80	30	17	2	1
1	2	3	4	5	6	7	8	9	10

size=10

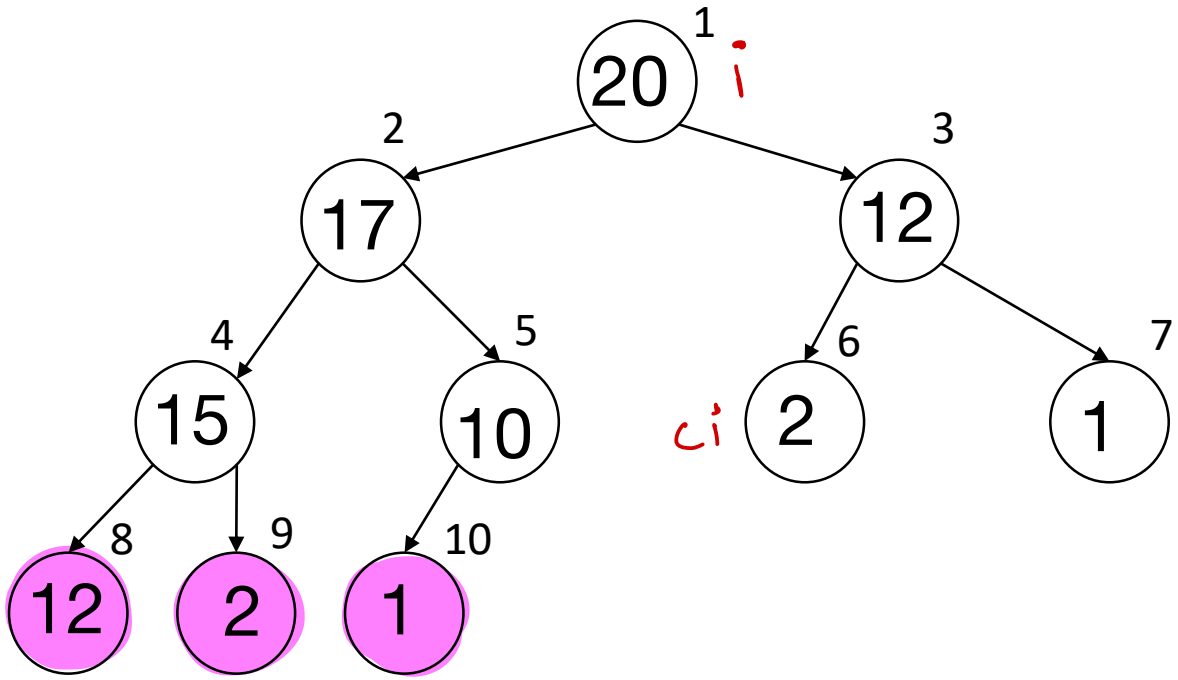
```
size = length - 1;
for(i=size/2; i>=1; i--) {
    temp = arr[i];
    ci = i * 2;
    while( ci<=size) {
        if ((ci+1)<=size && arr[ci+1] > arr[ci])
            ci++;
        if (temp > arr[ci])
            break;
        arr[ci/2] = arr[ci];
        ci = ci*2;
    }
    arr[ci/2] = temp;
}
```

Max Heap – Initialize – Ready



1	2	3	4	5	6	7	8	9	10
~~20~~	~~12~~	~~35~~	~~15~~	~~10~~	~~80~~	~~30~~	~~17~~	~~2~~	~~1~~

Max Heap – Delete Element



30 35 80

80	17	35	15	10	20	30	12	2	1
1	2	3	4	5	6	7	8	9	10

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>