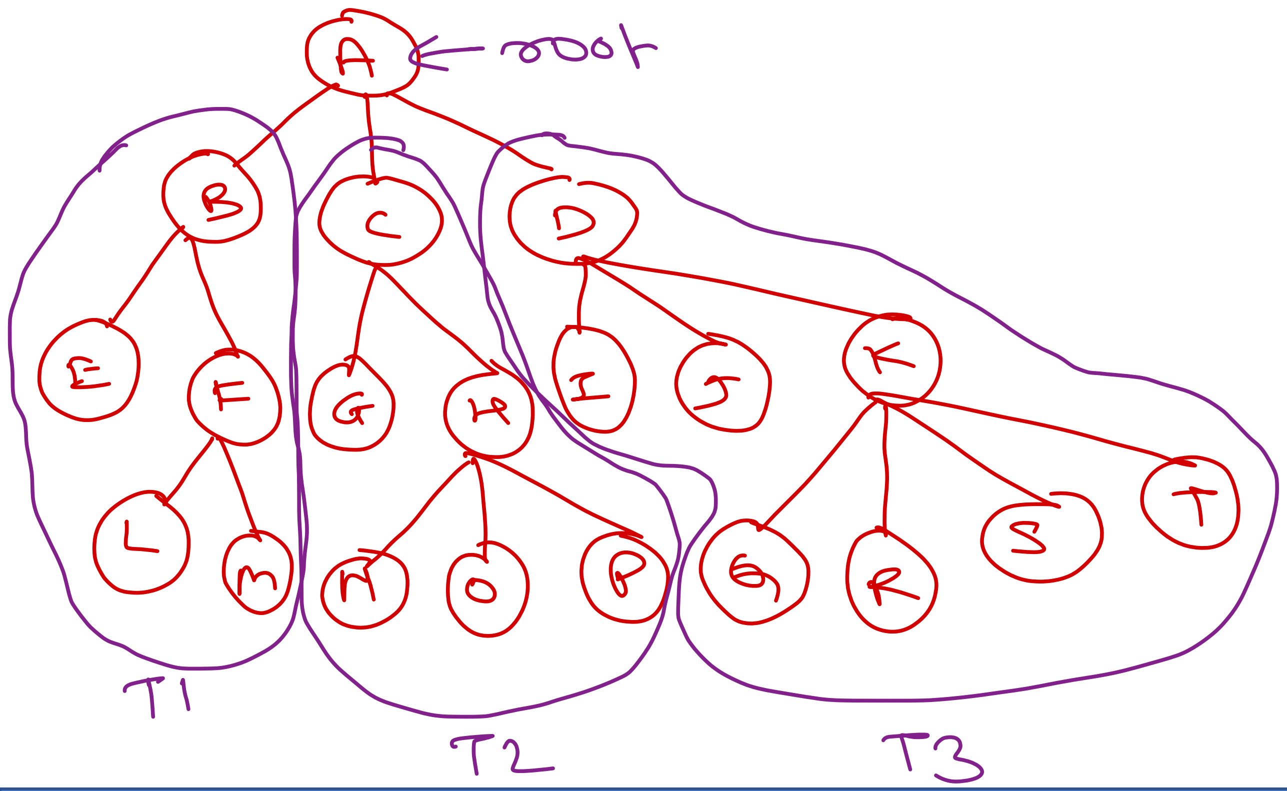
Data Structure & Algorithms

Sunbeam Infotech

<u>Nilesh Ghule</u>

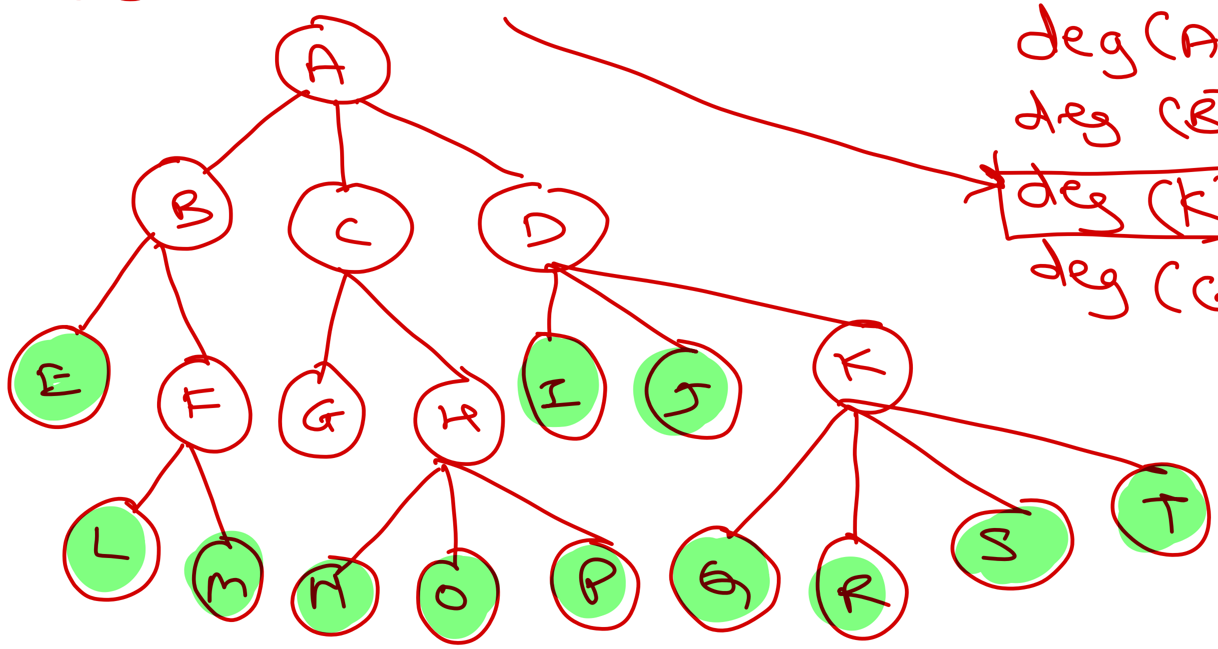
Tree Definition → non-linear

- **Tree** is a finite set of nodes with one specially designated node called the "**root**" and the remaining node are partitioned into disjoints sets T1 to Tn, where each of those sets is a TREE.
- T1 to Tn are called **sub-trees** of the root



Tree terminologies

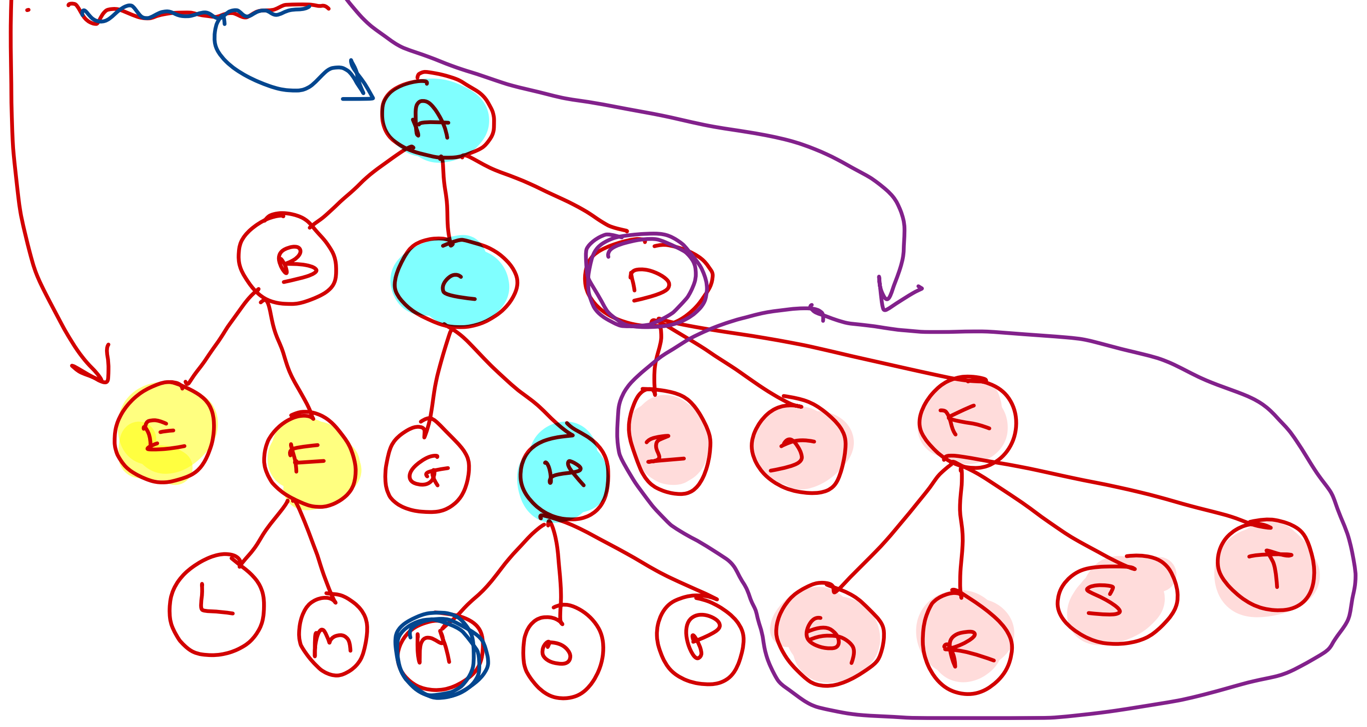
- Node: A item storing information and branches to other nodes
  pointers child
- Null Tree: Tree with no node (empty tree).
- Leaf Node: Terminal node of a tree & does not have any node connected to it
  child
- Degree of a Node: No of sub trees of a node
- Degree of a tree: Degree of a tree is maximum degree of a node in the tree



deg(A) = 3
deg (B) = 2
$\boxed{\text{deg (K) = 4}}$
deg (G) = 0

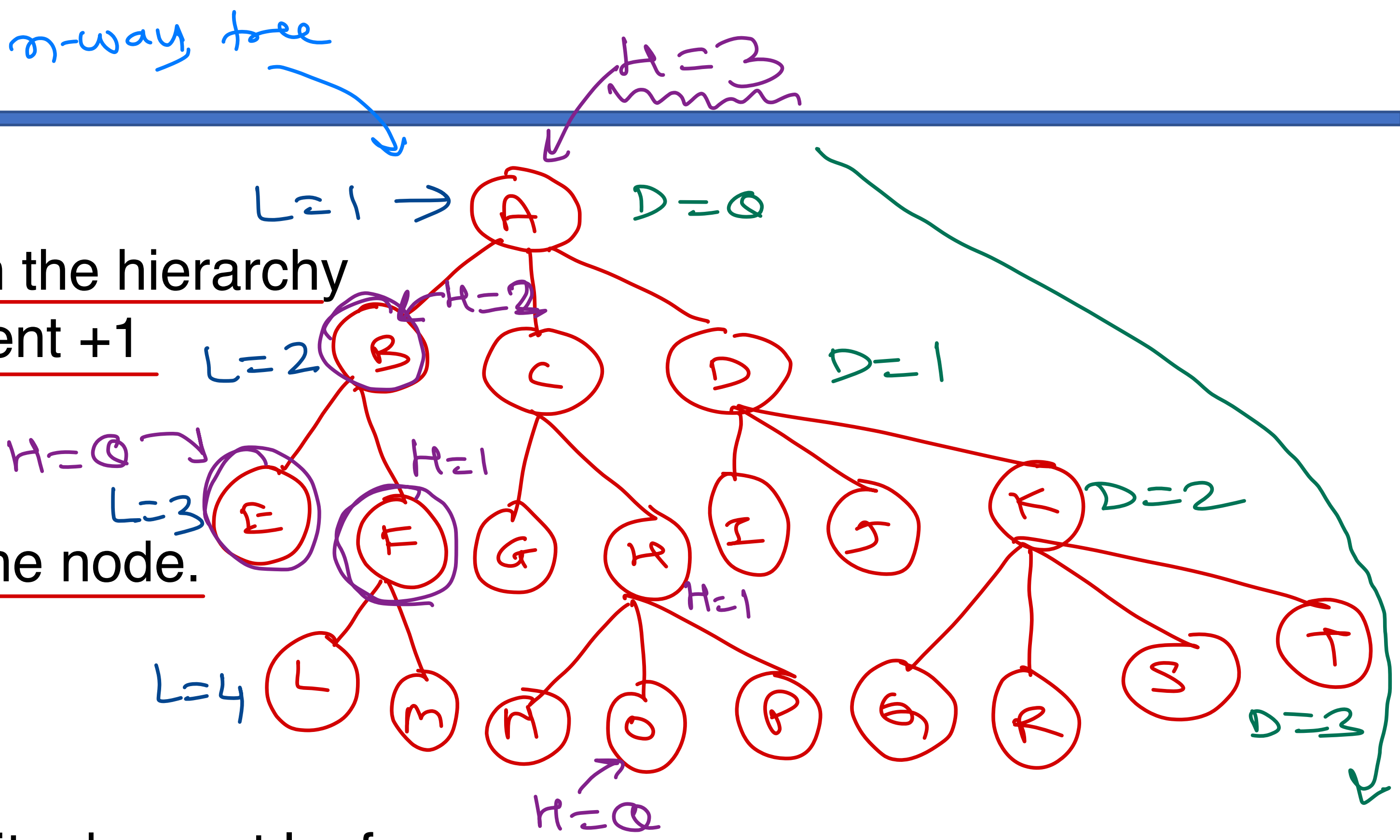
Tree terminologies

- Parent Node: node having other nodes connected to it (non-leaf nodes)
- Siblings: Children of the same parents
- Descendants: all those node which are reachable from that node
- Ancestor: all the node along the path from the root to that node



Tree terminologies

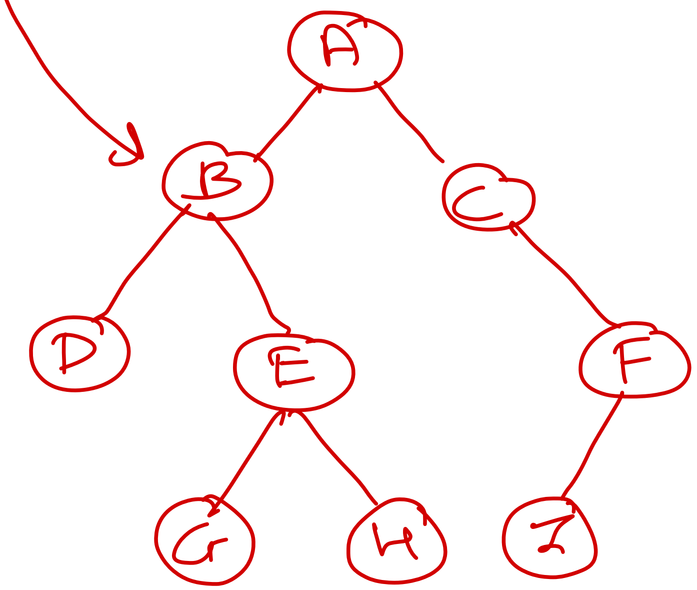
- Level of a Node:
  - Indicates the position of the node in the hierarchy
  - Level of any node is level of its parent +1
  - Level of root is 1
- Depth of a node:
  - Number of nodes from the root to the node.
  - Depth of root is 0
  - Level = Depth + 1
- Height of a node:
  - Number of nodes from the node to its deepest leaf.
  - Height of node = max height of its child + 1
  - Height of empty/null tree is -1
- Height of a tree: Height of root of the tree.
- Traversal: Visiting each node of tree exactly once

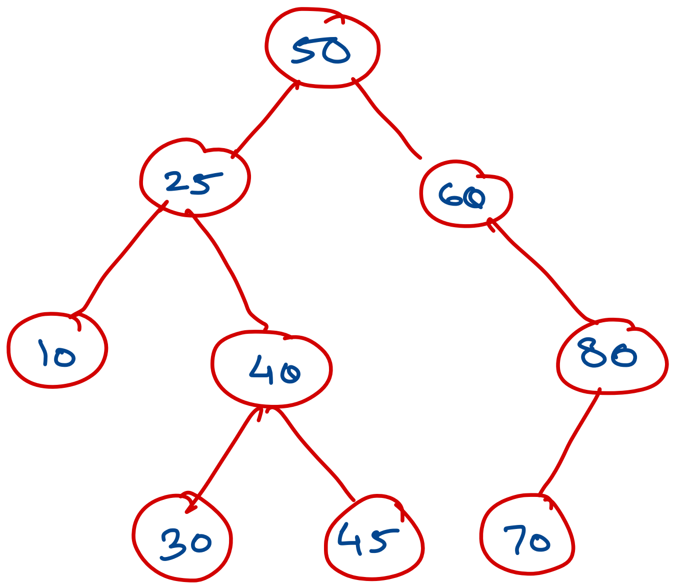


height(node) = MAX(height(left tree), height(right tree)) + 1

Types of trees

- Binary Trees → max 2 child nodes.
  - It is a finite set of nodes partitioned into three sub sets:- Root, Left sub tree, Right sub tree
- Binary Search tree
  - A binary search tree is a binary tree in which the nodes are arranged according to their values.





each node left child is smaller than the node & right child greater or equal to the node.

↑

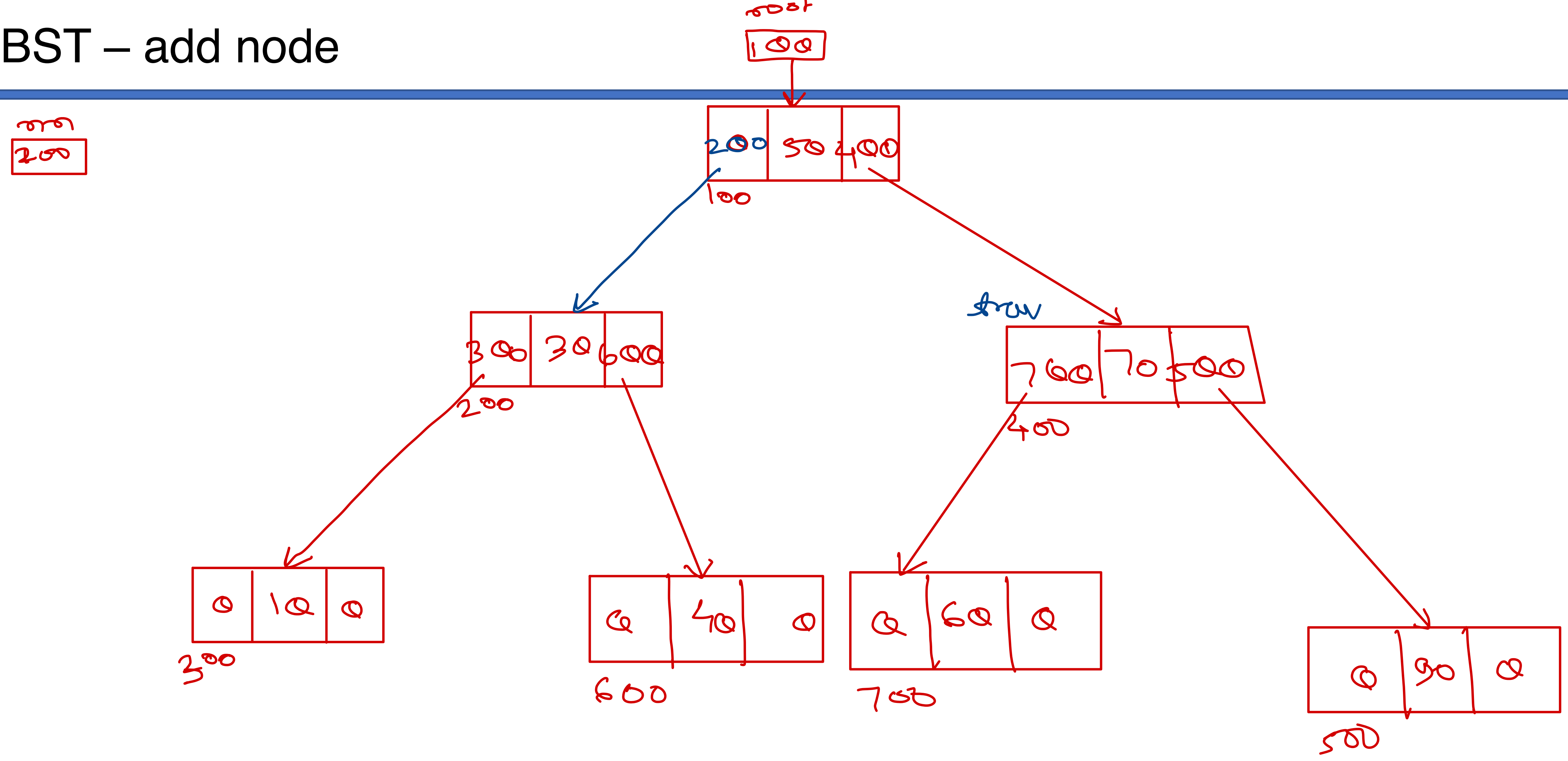
Common impl

Binary Tree Traversal

- In-order: L P R
- Pre-Order: P L R
- Post-Order: L R P
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal needs stack to store node pointers.

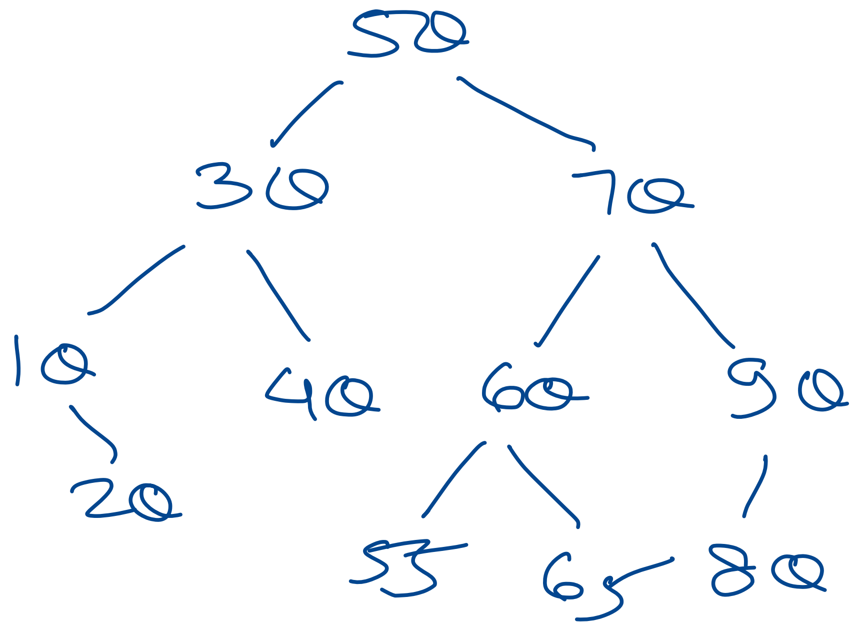
```
class BinarySearchTree {

    static class Node {
        int data;
        Node left, right;
        ctor() ...
    }

    Node root; // ptr to first(root) node
               of root.

    ctor().

    add(int val) {... }

    preorder() { ... }

}
```

BST – add node

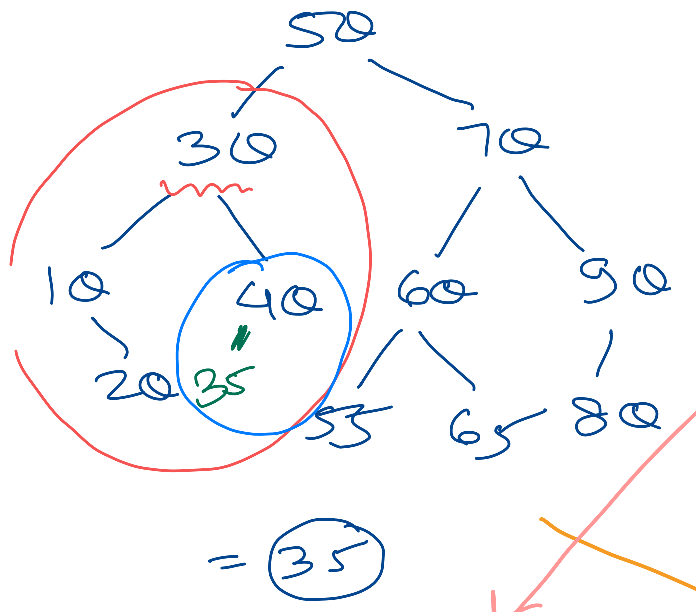

root
100
trav
200
200 | 50 | 400
100
300 | 30 | 600
200
700 | 70 | 500
400
0 | 10 | 0
300
0 | 40 | 0
600
0 | 60 | 0
700
0 | 90 | 0
500

BST – add node

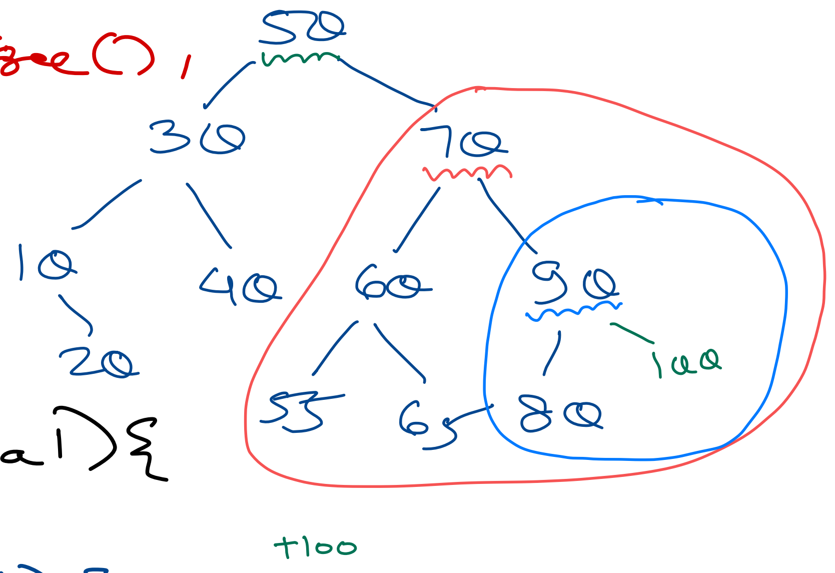
- ✓ 50
- ✓ 30
- ✓ 10
- ✓ 70
- ✓ 90
- ✓ 40
- ✓ 60
- ✓ 20
- ✓ 55
- ✓ 65
- ✓ 80



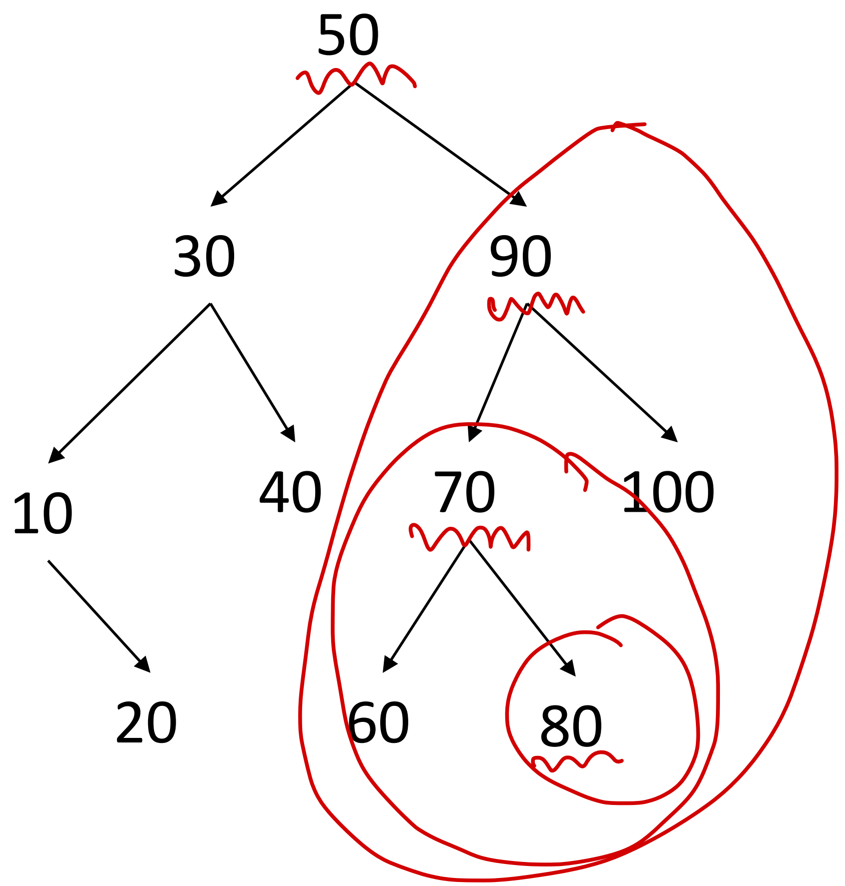
BST – add node (recursive)



```
main() {
    BSTree t = new BSTree();
    t.add(—);
}

void add(int val){
    if(isEmpty())
        root = new Node(val);
    else
        add(root, val);
}

void add(Node trav, int val){
    if(val < trav.data) {
        if(trav.left == null) {
            trav.left = new Node(val);
            return;
        }
        add(trav.left, val);
    }
    else {
        if(trav.right == null) {
            trav.right = new Node(val);
            return;
        }
        add(trav.right, val);
    }
}
```

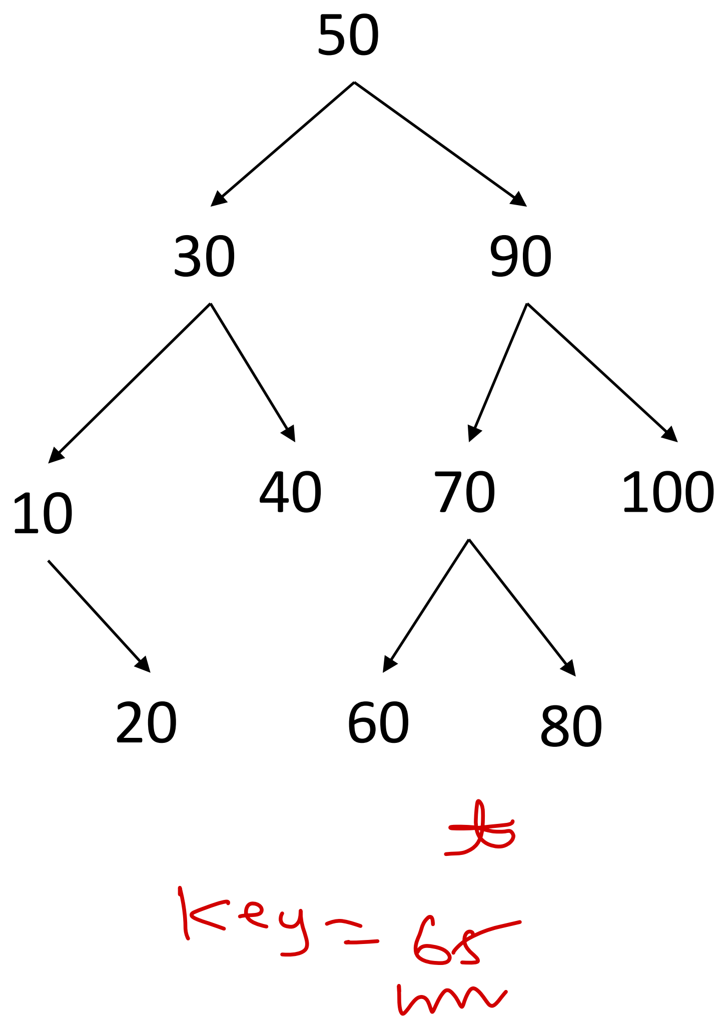


BST – search (recursive)

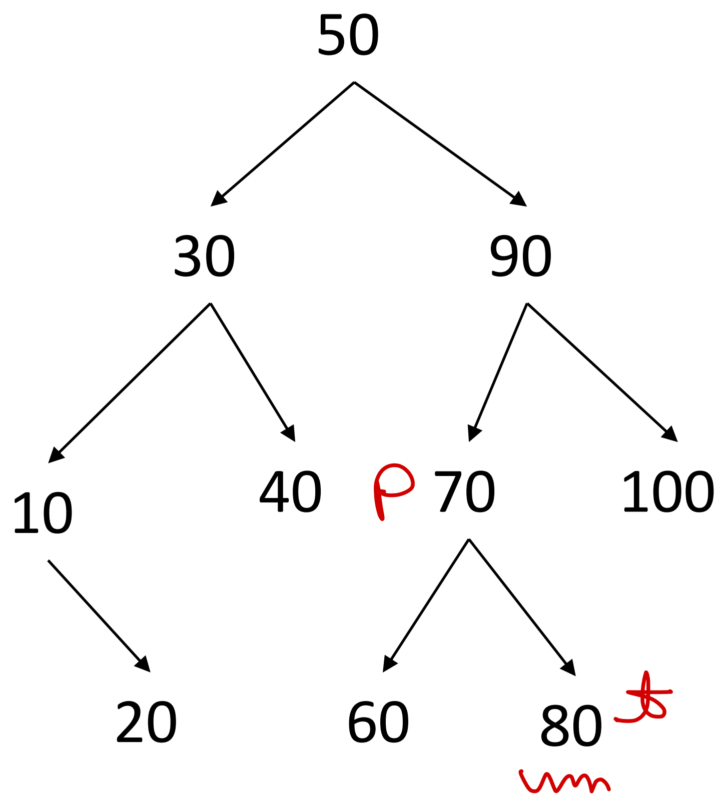


```
binSearch(Node trav, int key){
        if(trav == null)
            return null;
        if(key == trav.data)
            return trav;
        if( key < trav.data)
            binSearch(trav.left, key);
        else
            binSearch(trav.right, key);
}
```

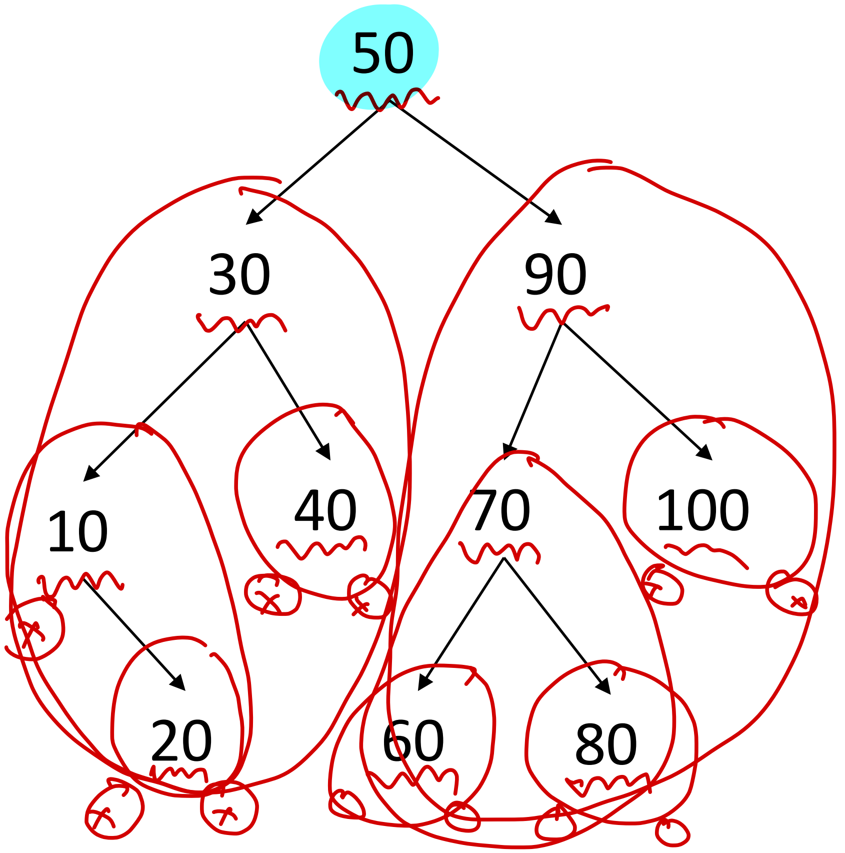
BST – search

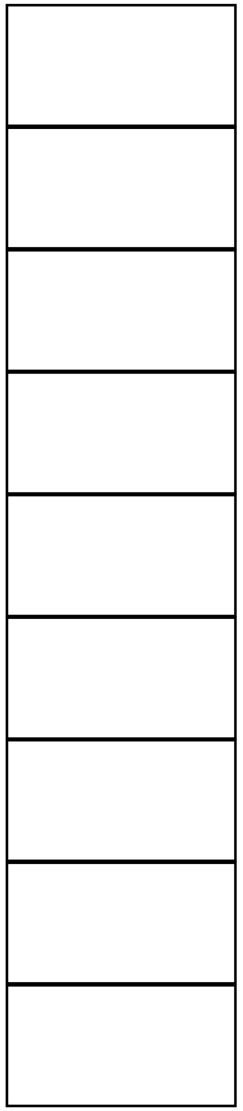


BST – search – with parent



BST – PreOrder → Recursive   P L R

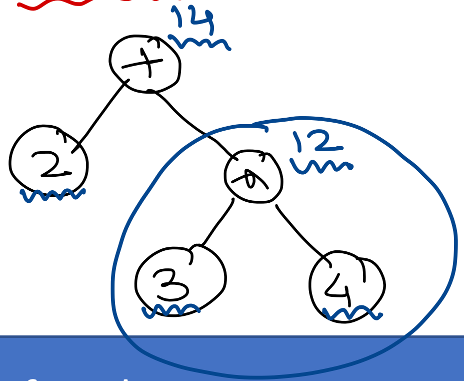




```
void preorder(Node trav) {
    if(trav==null)
        return;

    pf(trav.data);
    preorder(trav.left);
    preorder(trav.right);
}
```

50 30 10 20 40
90 70 60 80 100

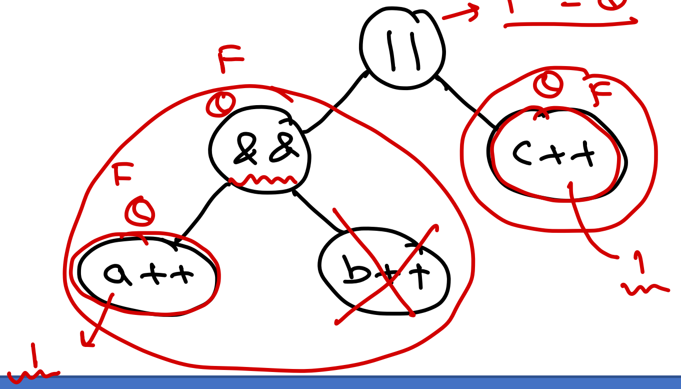
2 + 3 * 4
↓
expression tree
↓
post-order



a=0, b=0, c=0;

a++ && b++ || c++ ;

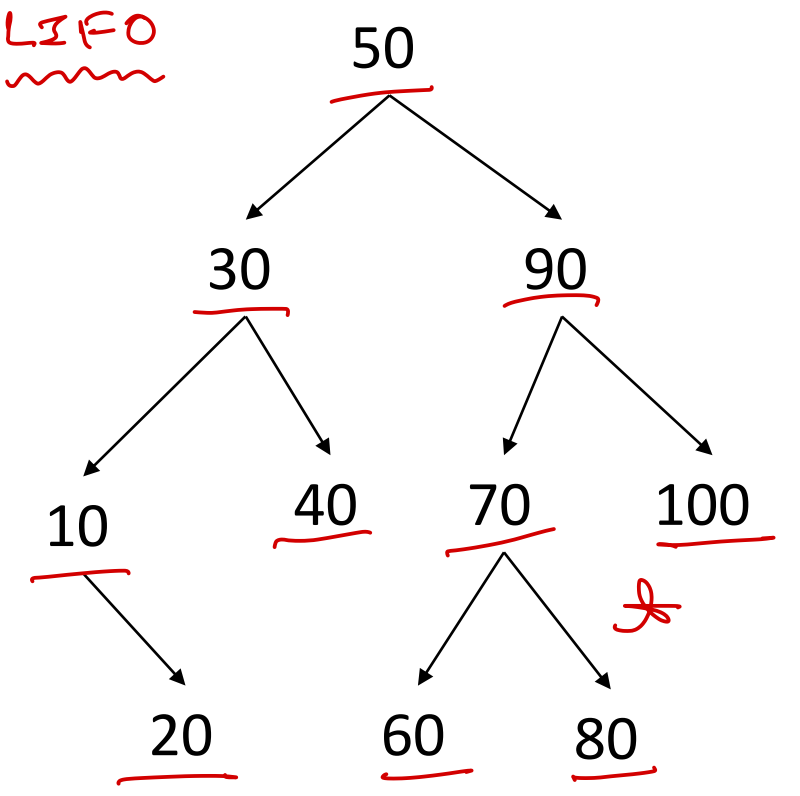
pf("%d %d %d", a,b,c);
1 0 1



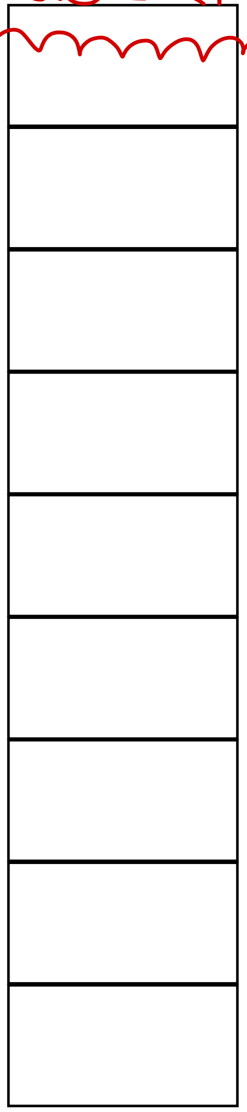
BST – PreOrder

- non-recursive

LIFO



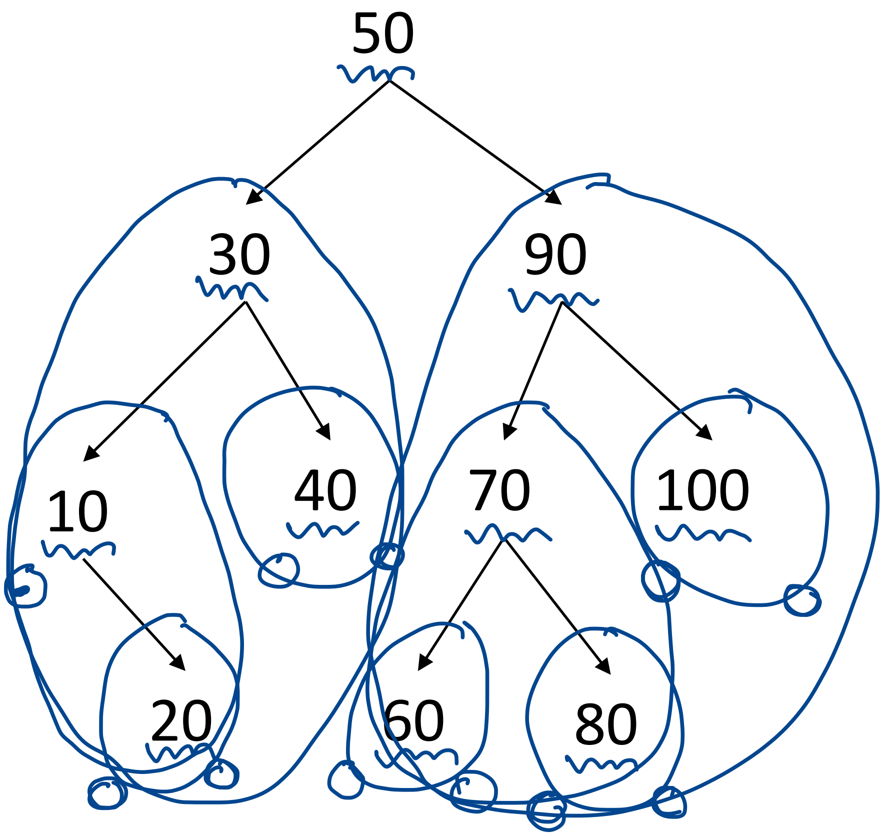
Stack<Node>

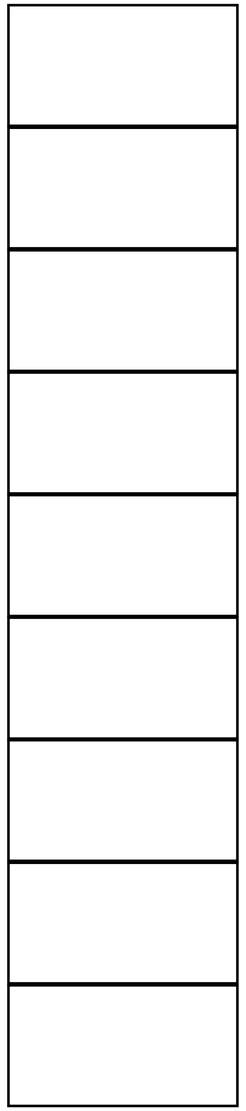


```
trav = root;
while(trav != null || !s.isEmpty()) {
    while(trav != null) {
        pf(trav.data);
        if(trav.right != null)
            s.push(trav.right);
        trav = trav.left;
    }
    if(!s.isEmpty())
        trav = s.pop();
}
```

BST – InOrder

L P R

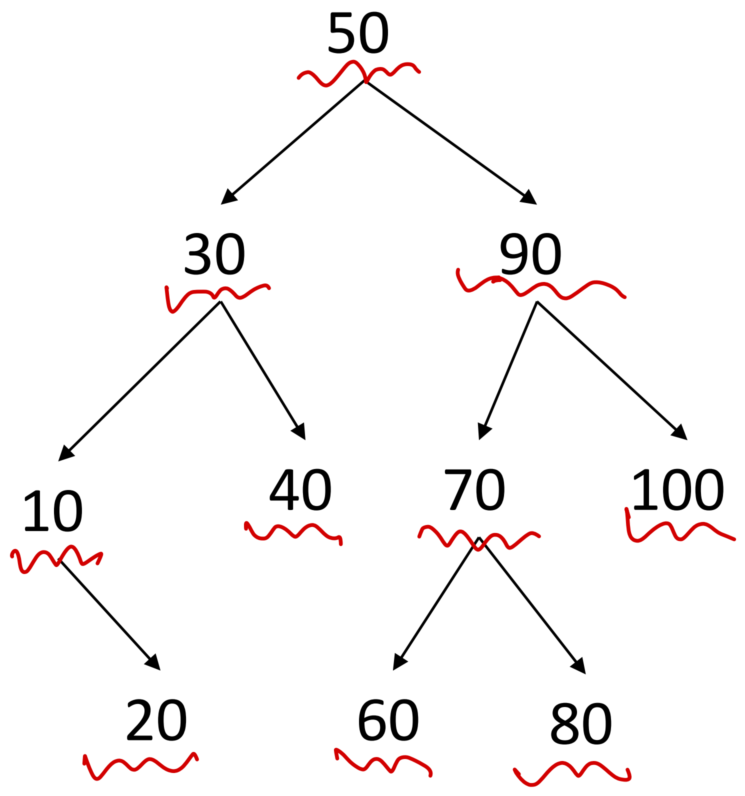


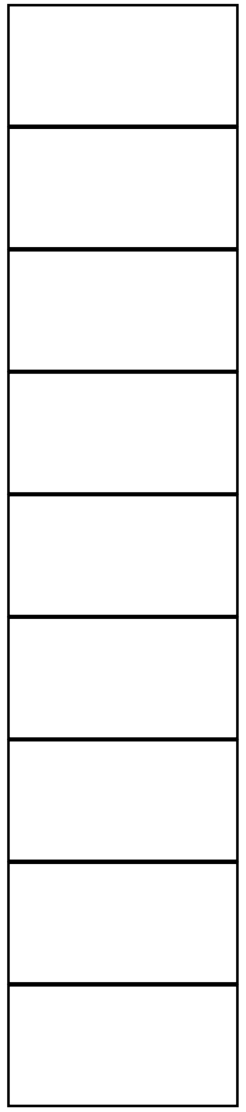


10 20 30 40 50

60 70 80 90 100

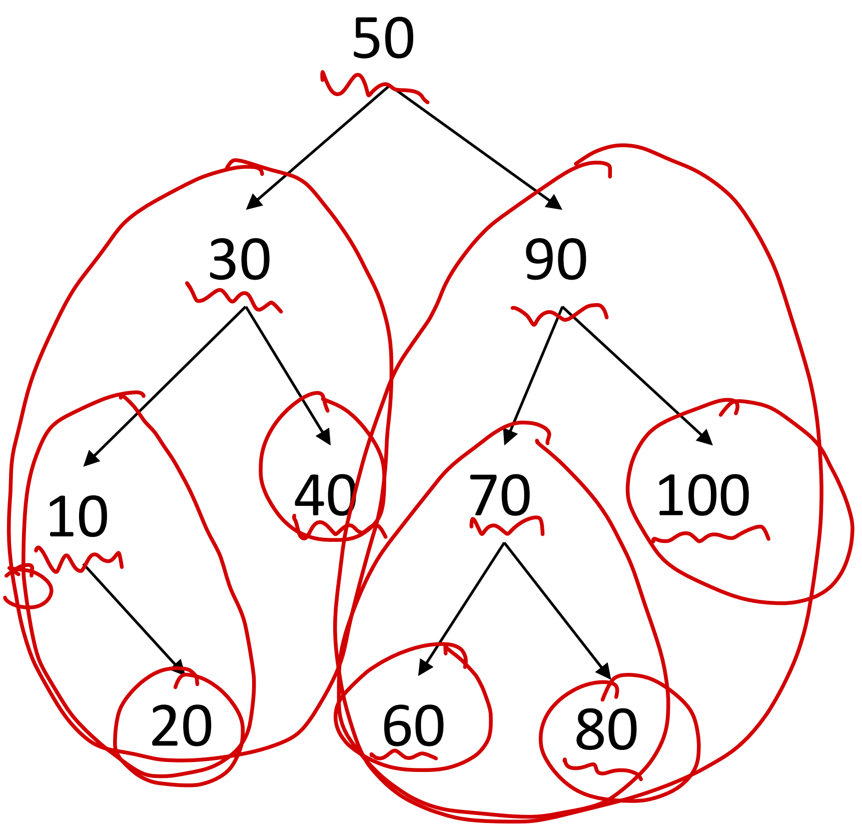
BST – InOrder - non recursive → L P R

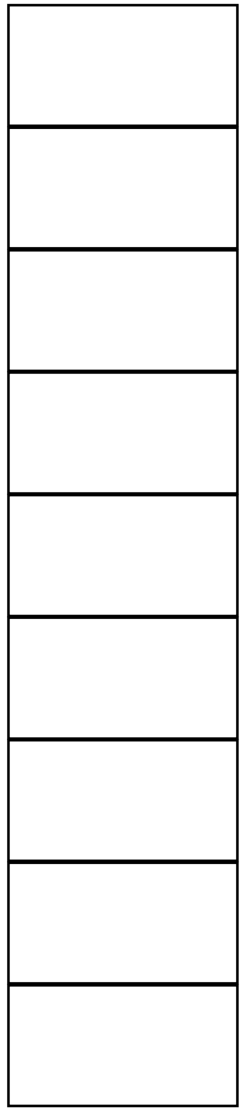




```
trav = root;
while(trav != null || !s.isEmpty())
{   while(trav != null) {
        s.push(trav);
        trav = trav.left;
    }
    if(!s.isEmpty()) {
        trav = s.pop();
        print(trav.data);
        trav = trav.right;
    }
}
```

BST –PostOrder

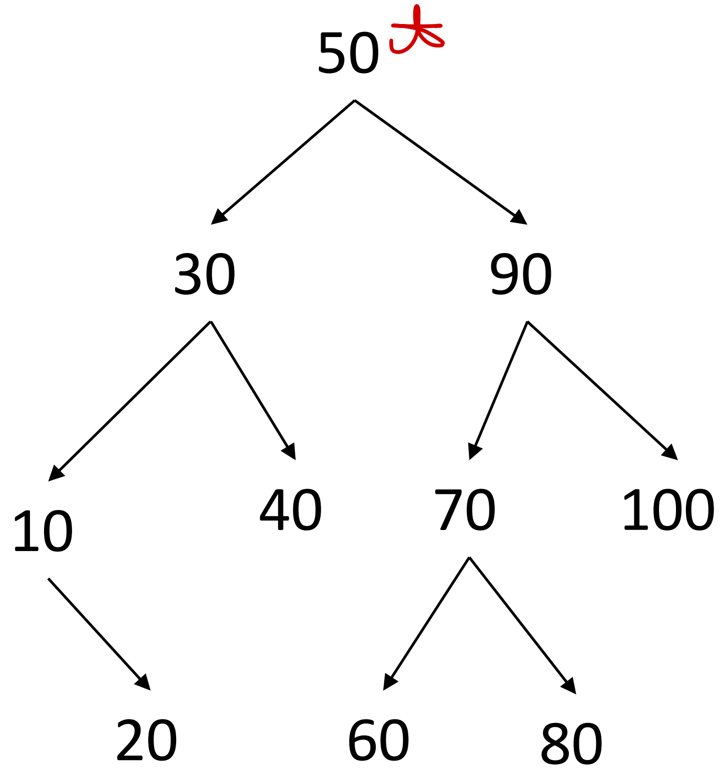


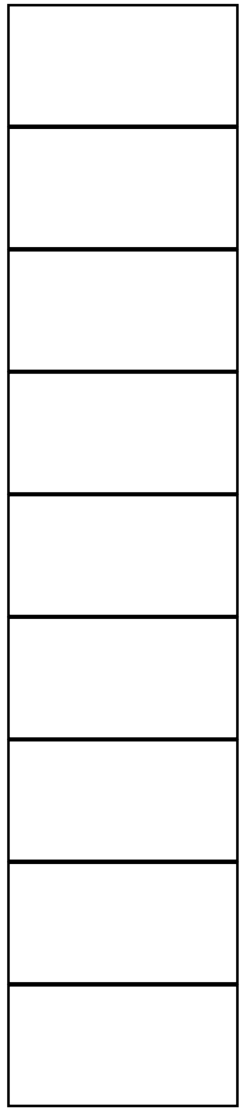


20 10 40 30 60

80 70 100 90 50

BST –PostOrder - non recursive - L R P





```
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.visited) {
                print(trav.data);
                trav.visited = true;
                trav = null;
            }
            else {
                s.push(trav);
                trav = trav.right;
            }
        }
}
```

SUNBEAM

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>