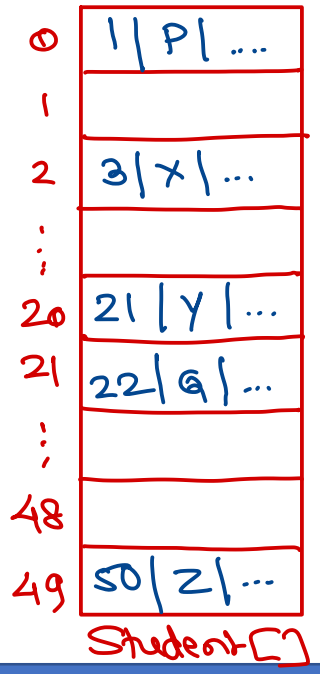
Data Structure & Algorithms

Nilesh Ghule

Hash Table

- Associative data structure → name, mobile / roll, Student
- Stores key-value so that for a given key, value can be searched in fastest possible time. Ideal time complexity is O(1).
- Example:

Students → 50
roll num → 1 to 50



put: O(1)
```
index = roll - 1;
arr[index] = obj;
```

get: O(1)
```
index = roll - 1;
return arr[index];
```

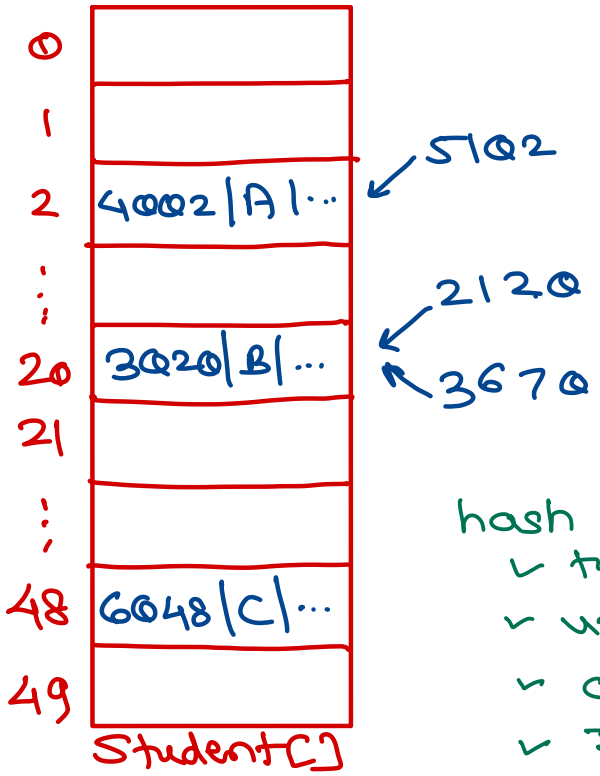
key = roll → value = student

key (roll) —hash fn (roll - 1)→ slot (index)

Hash Table

- Hash Function is math function of key, that yields slot in the table.
- If different keys resulting in same slot in the table, it is called as collision.

$h(k) = k \% 50$



- 4002
- 3020
- 6048
- 2120
- 3670
- 5102

Load Factor:

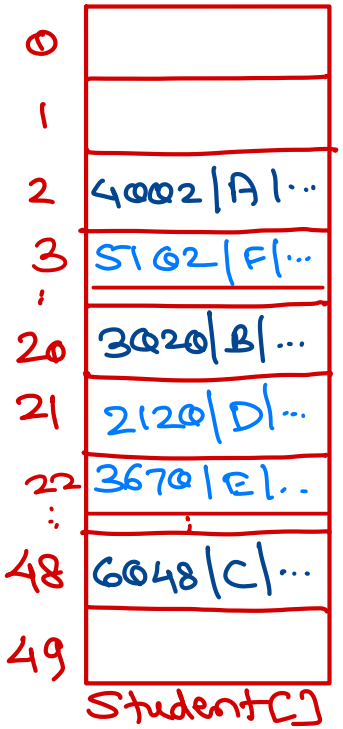
$$\frac{\text{num of entries}}{\text{num of slots}}$$

1. num of entries < num of slots, Load Factor < 1
2. num of entries = num of slots, Load Factor = 1
3. num of entries > num of slots, Load Factor > 1

hash fn must be
- two diff keys may have same index (not ideal)
- uniform distribution
- consistent (not random / not time based)
- It is observed that multiplication with prime num generates more uniform index/slots.

Hash Table

- Collision handling methods: Open addressing or Chaining
- Open addressing → can be used only if load factor <= 1.
  - Rehashing: Linear probing, Quadratic probing, …



h(k) = k % 50

rh(k) = (prevSlot + 1) % 50

rehash is math fn to be used to find next possible slot for given key (if current found slot is occupied).

- 4002 → 2
- 3020 → 20
- 6048 → 48
- 2120 → 20 21
- 3670 → 20 21 22
- 5102 → 2 3

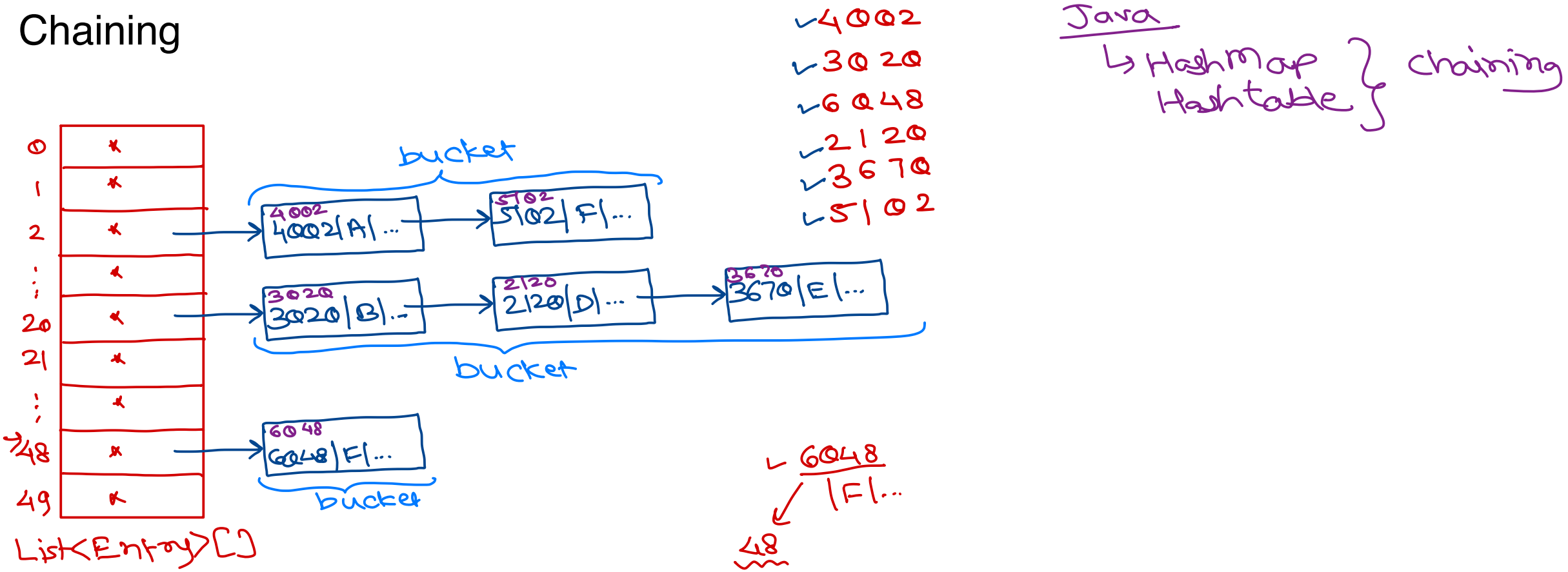
```
put(k, v):
    i = h(k)
    while (arr[i] is not empty)
        i = rh(k)
    arr[i] = v

v = get(k):
    i = h(k)
    while (arr[i] is not empty)
        if (arr[i] == k)
            return v
        i = rh(k)
    return null
```

Hash Table

- Load factor = Number of entries / Number of slots
- Cases
  - Load factor < 1
  - Load factor = 1
  - Load factor > 1

Hash Table

- Chaining

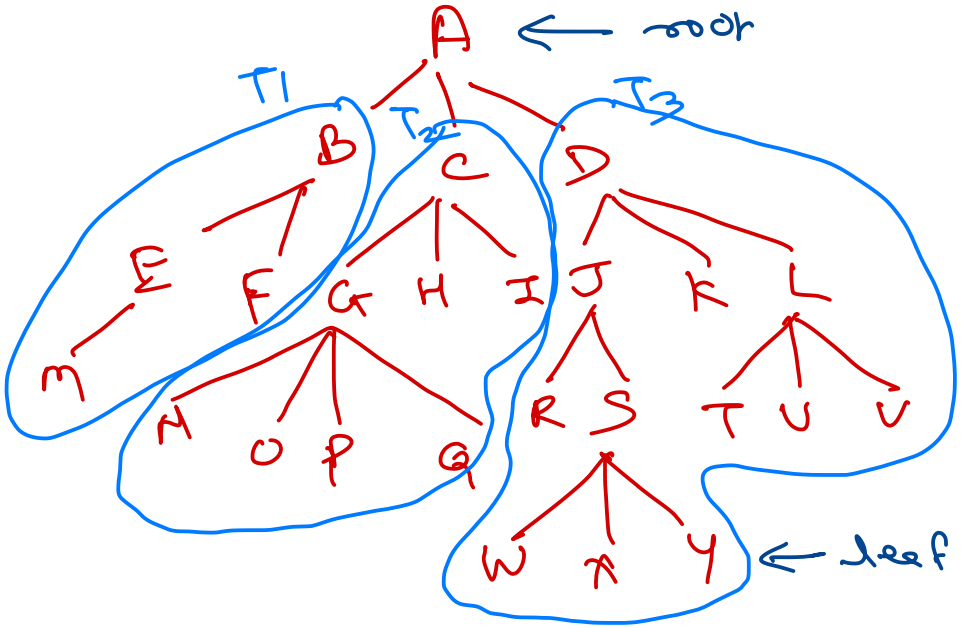


Tree Definition

- **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

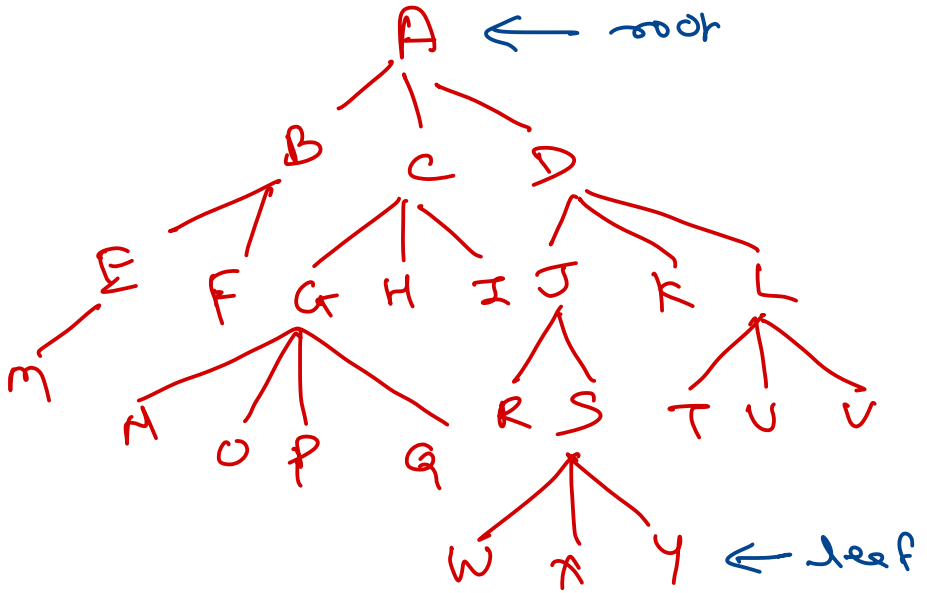
Non-Linear Data Structure

Hierarachial data



Tree terminologies

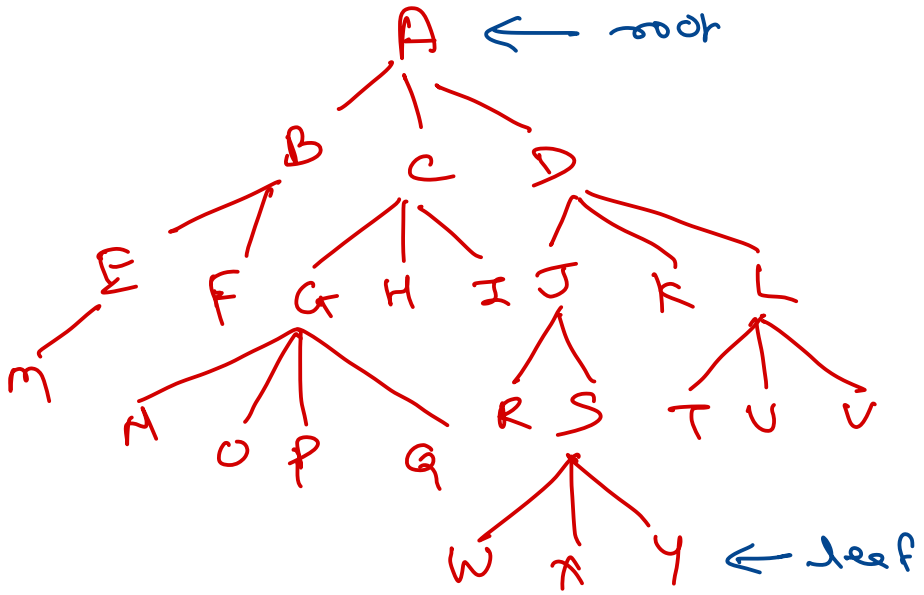
- Node: A item storing information and branches to other nodes
- Null Tree: Tree with no node
- Leaf Node: Terminal node of a tree & does not have any node connected to it
- Degree of a Node: No of sub trees of a node
- Degree of a tree: Degree of a tree is maximum degree of a node in the tree



Tree terminologies

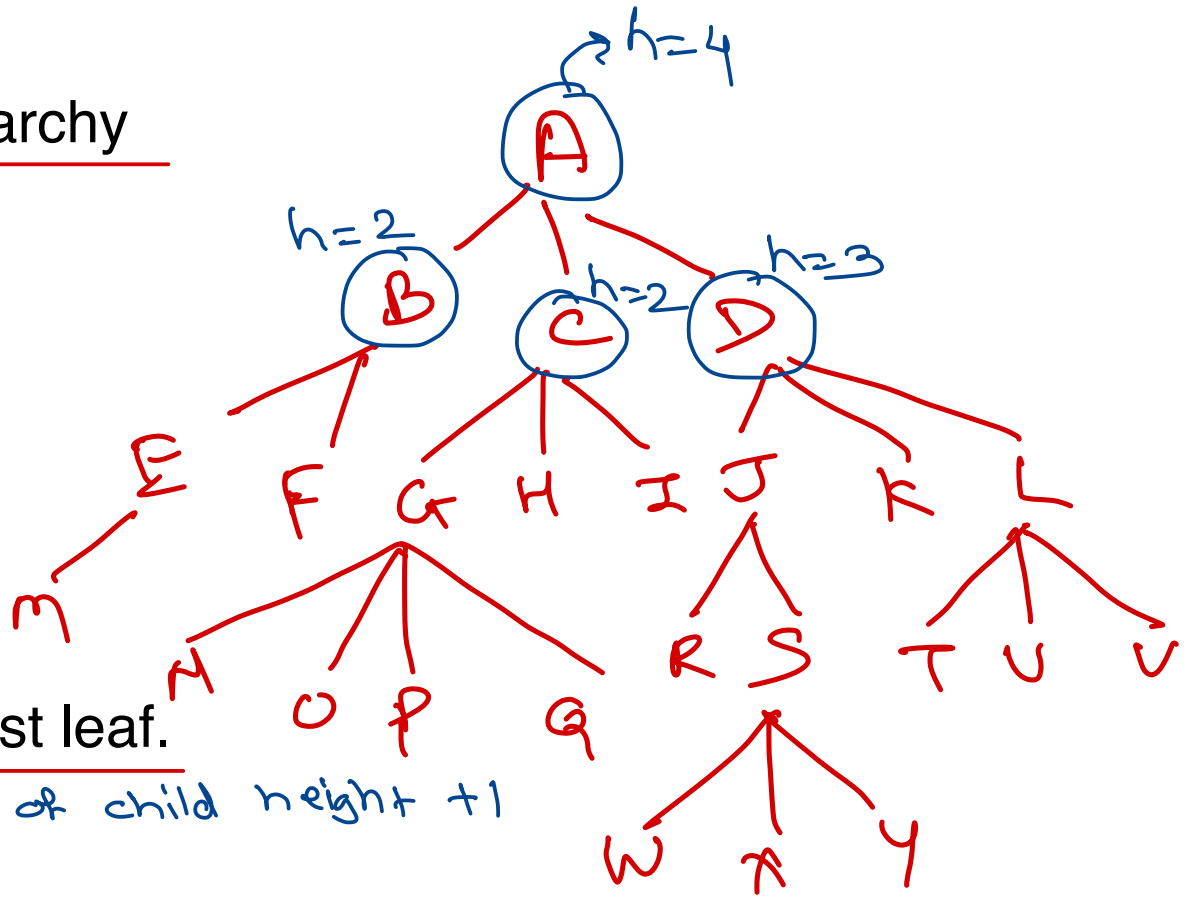
Non-Leaf

- Parent Node: node having other nodes connected to it
- 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



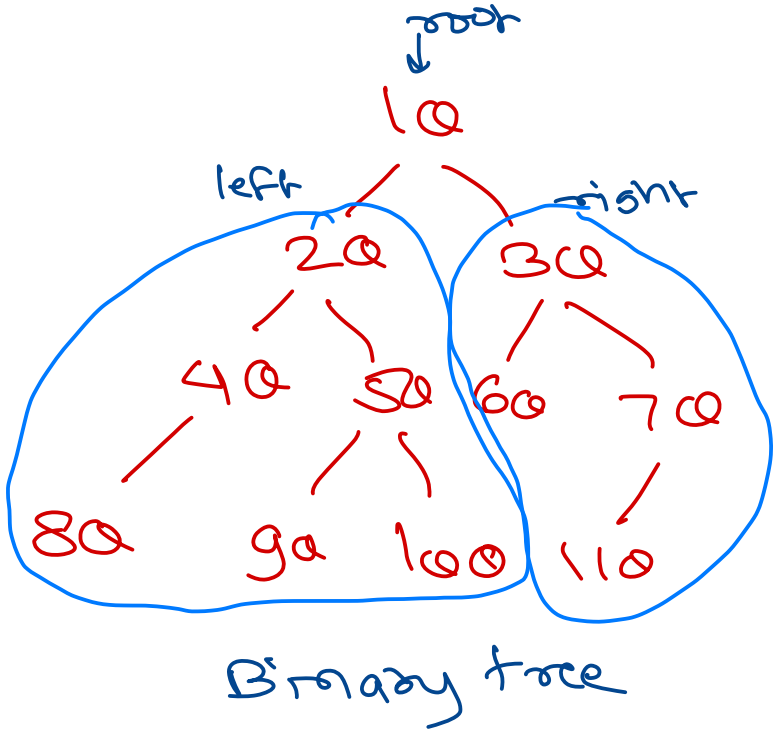
Tree terminologies

- 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 = ~~height of its child + 1~~ Max of child height +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



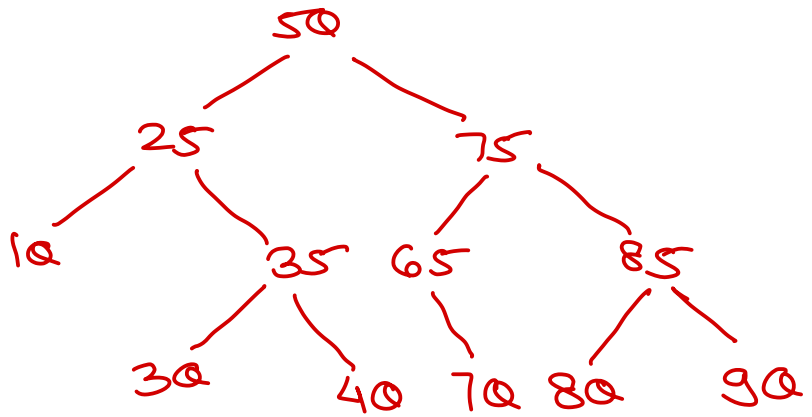
Types of trees

- Binary Trees
  - each node has max 2 child.
  - tree with degree = 2
  - 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.



Binary tree

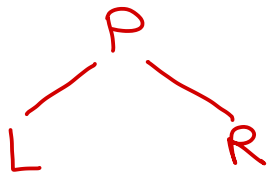
left child is smaller than its parent.
right child is greater or equal to its parent.

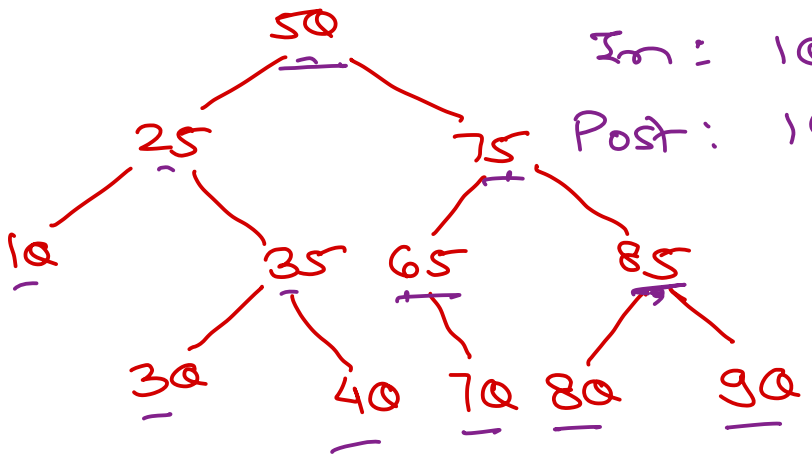


Binary Search Tree

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.





Pre : 50 25 10 35 30 40 75 65 70 85 80 90

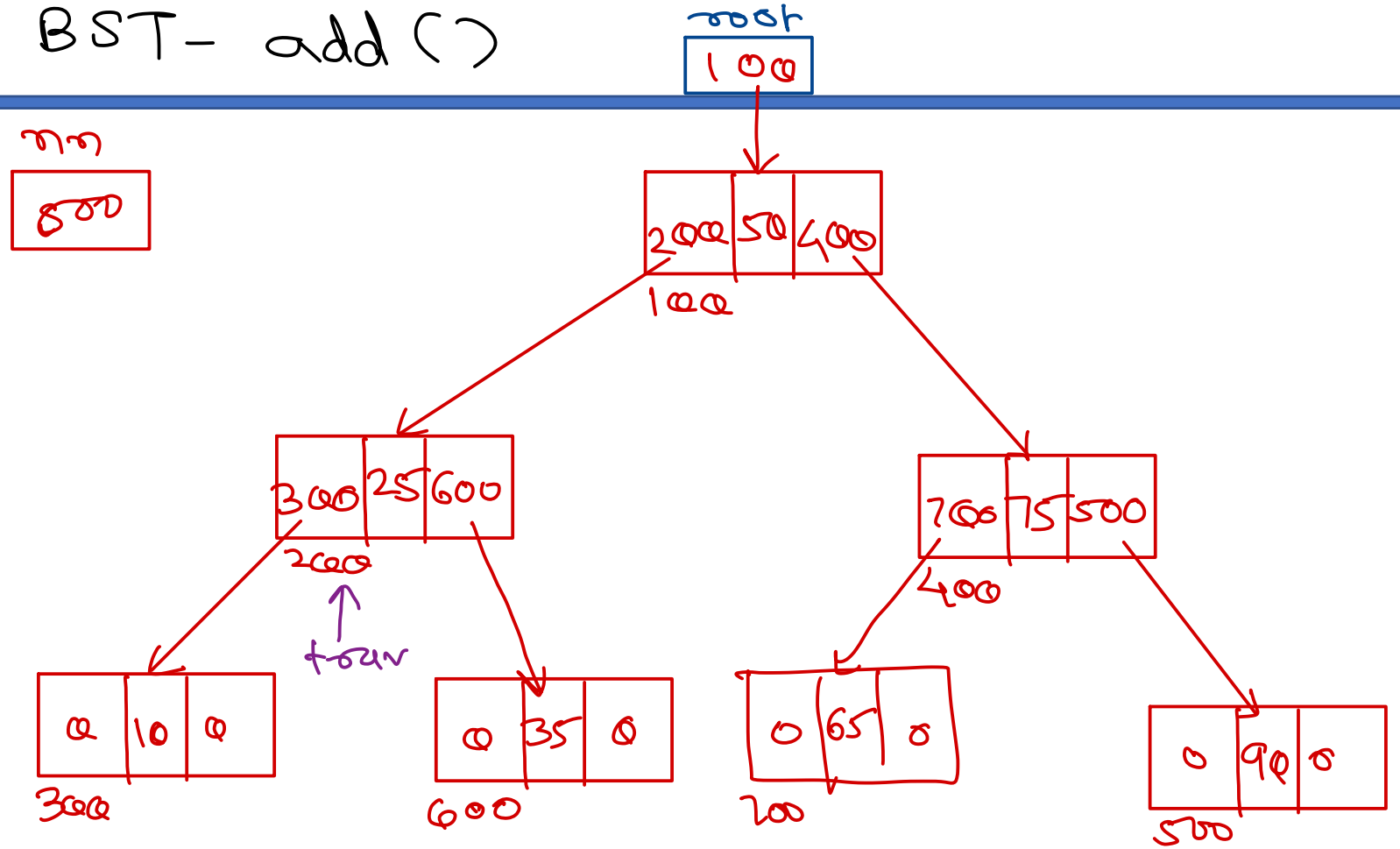
In : 10 25 30 35 40 50 65 70 75 80 85 90

Post : 10 30 40 35 25 70 65 80 90 85 75 50

```
preorder(cur):
    if(cur == null)
        return
    print(cur.data)
    preorder(cur.left)
    preorder(cur.right)
```

```
inorder(cur):
    if(cur == null)
        return
    inorder(cur.left)
    print(cur.data)
    inorder(cur.right)
```

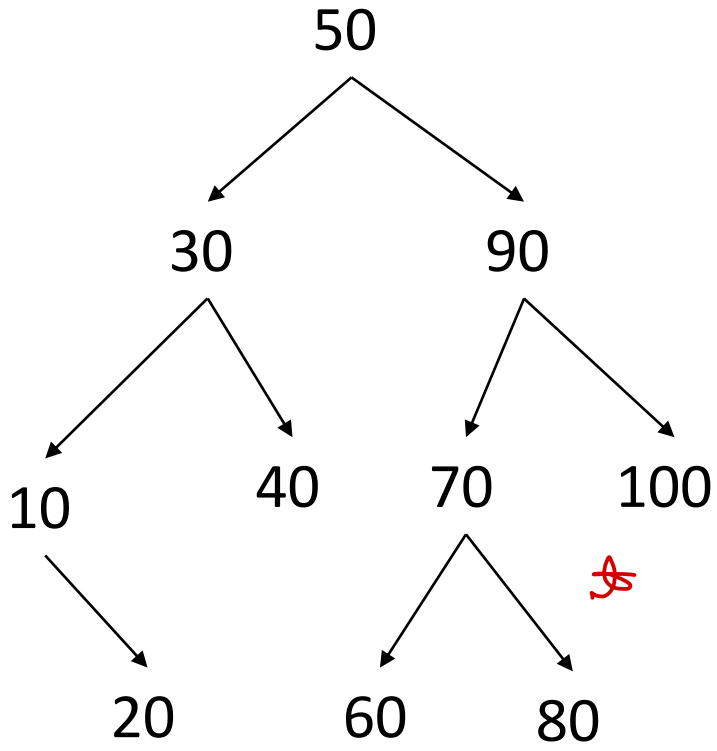
BST- add()




```
if(root == null)
    root = nn;
else {
    trav = root;
    while(true) {
        if(val < trav.data){
            if(trav.left == null){
                trav.left = nn;
                break;
            }
            else
                trav = trav.left;
        }
        else {
            if(trav.right == null){
                trav.right = nn;
                break;
            }
            else
                trav = trav.right;
        }
    }
}
```

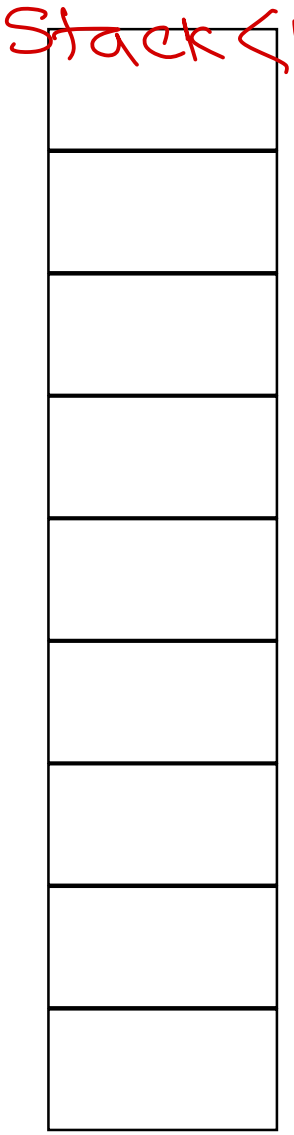
50
25
10
75
90
35
65

BST – PreOrder – Non-Recursive



50 30 10 20 40

90 70 60 80 100




```
trav = root;
while(trav!=null || !s.isEmpty()){

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

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>