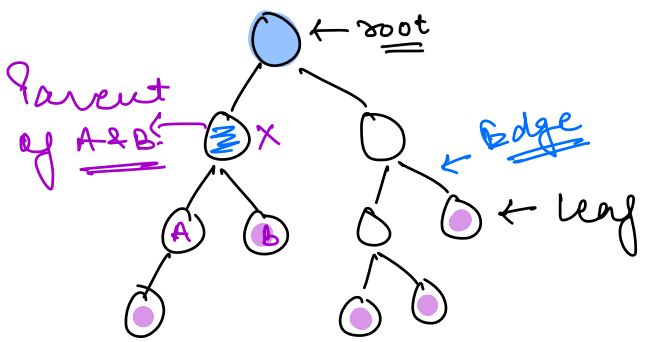
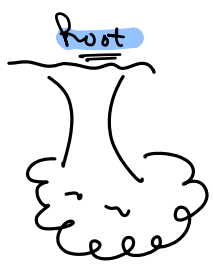
XML
HTML
YAML
JSON
Directory structure
HashMap / HashSet
Heap / Priority Queue
Tries
B-Trees (DB indexes)

Trees :-

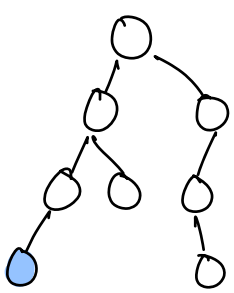
↳ Hierarchical data.



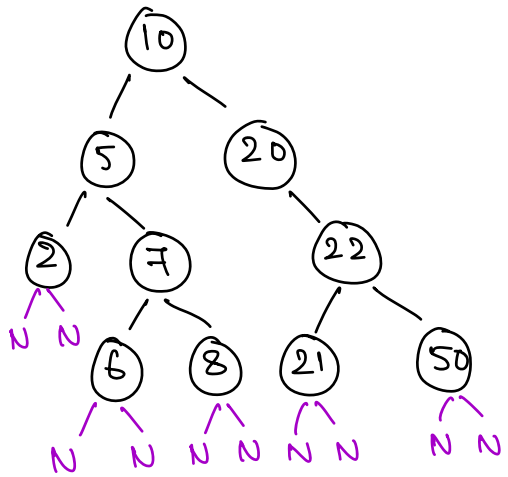

Root ⇒ No parent
Leaf ⇒ No children.

# No. of edges in a tree with N nodes.



⇒ N-1 edges.

# Height of a node ⇒ Distance of the node to farthest reachable leaf node




Ht(5) = 2

Ht(22) = 1

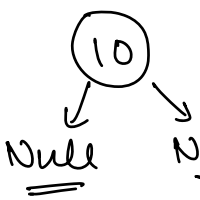
Ht(6) = 0

Ht(10) = 3

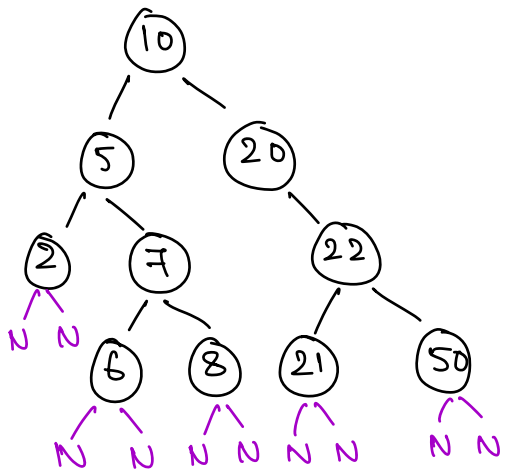
Ht(null) = -1

```
Class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;
    TreeNode (x) {
        this.data = x;
        this.left = Null;
        this.right = Null;
    }
}
```

```
TreeNode node = new TreeNode(10);
```



# Depth of a Node :- Distance of the node from the root node.




Depth (10) = 0
Depth (6) = 3
Depth (7) = 2

} Level

# Height of a Binary Tree :-

↳ Height of the root node.

→ Distance of the root from the farthest reachable leaf node.

→

Height of a B.T = 1 + max (height of LST, height of RST);

Tree Traversals :-

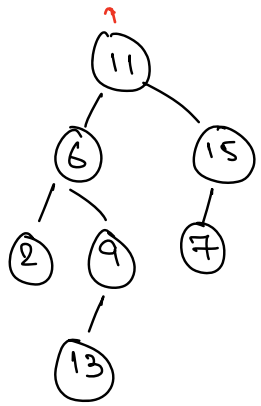
↳ Left Child will be visited before right child

Pre-Order : Root Left Right

In-order : Left Root Right

Post-order : Left Right Root

```
void preOrder(root) {
    if(root == Null)
        return;
    print(root.data);
    preOrder(root.left);
    preOrder(root.right);
}
```

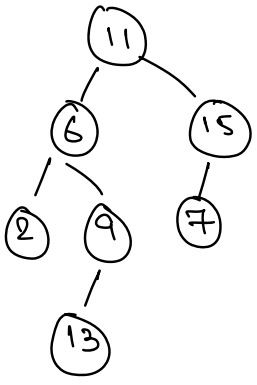



11, 6, 2, 9, 13, 15, 7

* We are traversing one depth at a time.
* DFS (Depth First Search)
* Fail Fast Approach (Pre Order) : If our fun^ has to fail then it should fail as fast as possible.

# Post Order Traversal :-

```
void postOrder(root) {
    if(root == Null)
        return;
    postOrder(root.left)
    postOrder(root.right);
    print(root.data);
}
```




2, 13, 9, 6, 7, 15, 11

* DFS ( Depth first Search)

# Height of a Binary Tree :-

Ht(tree) = Ht(root)

= max(Ht(LST), Ht(RST)) + 1

```
int height(root) {
    if(root == null)
        return -1;
    l = height(root.left);
    r = height(root.right);
    return max(l, r) + 1;
}
```

TC: O(N)

SC: O(H) → O(N)

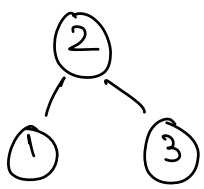
Q: Given a Binary Tree, search a value K.

```
bool search ( root , K ) {
    if ( root == null ) return false;
    if ( root.data == K )
        return true;
    return search(root.left, K) ||
           search(root.right, K)
}
```

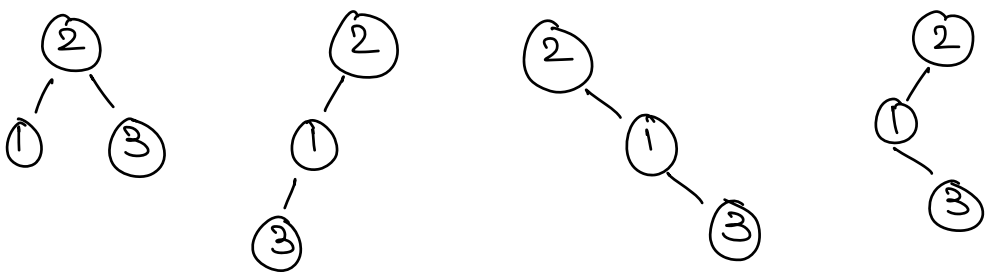
Q: Given the preOrder & inorder traversals of a tree, construct the tree.
[No duplicates]

Amazon
MS.
...

pre-order : 2 , 1 , 3
in-order : 1 , 2 , 3




# pre-order : 2 , 1 , 3

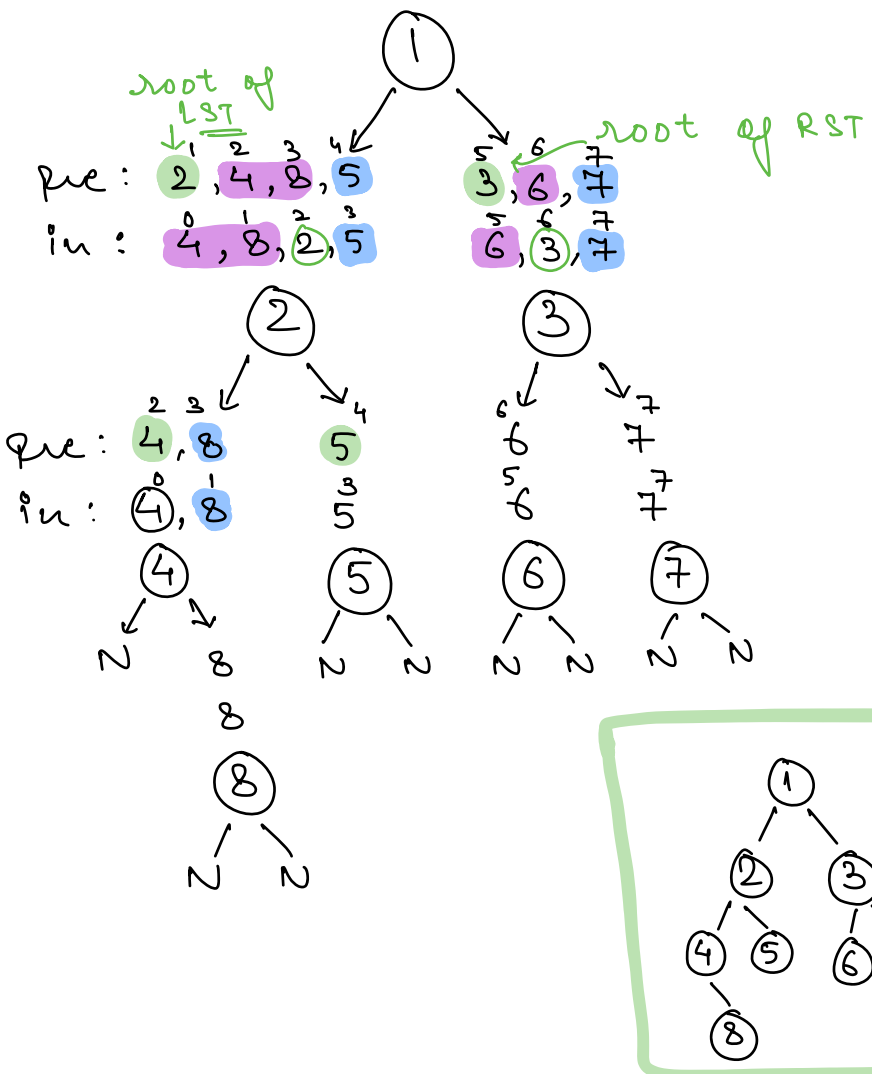


* first element in Pre Order is root node.

Pre-order: Root LST RST

In-order: LST Root RST

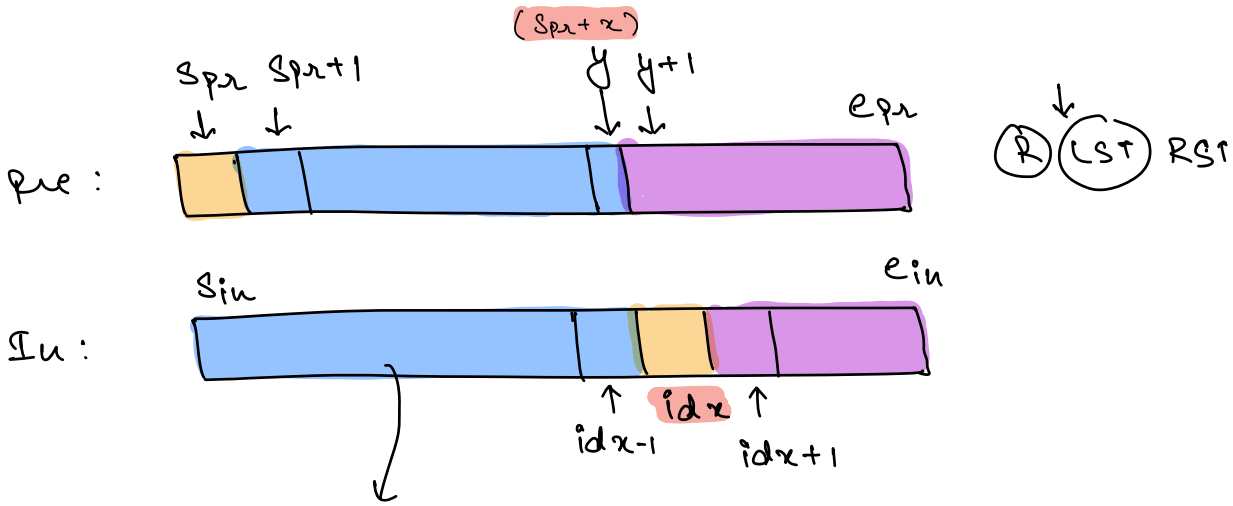
pre[0] => root

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| in: | 4, | 8, | 2, | 5, | 1, | 6, | 3, | 7 |

map < K, V >
↓ ↘
in[i] i

map
<4, 0>
<8, 1>
<2, 2>
<5, 3>
<1, 4>
<6, 5>
<3, 6>
<7, 7>




# of elements = [Sin, idx-1]

⇒ idx - Sin = x

[Spr+1, y] = idx - Sin

y - Spr = idx - Sin

y = Spr + idx - Sin = Spr + x

```
TreeNode buildTree(pre[], in[], sin, ein, spr, epr){
    // Assumption :- buildTree(pre[], in[], sin, ein, spr, epr)
    // returns the root of a tree which can be
    // created using pre[] → spr to epr
    //                in[] : sin to ein
    if (spr > epr) return Null;

    TreeNode root = new TreeNode(pre[spr]);
    int idx = map.get(pre[spr]);
    int x = idx - sin
    root.left = buildTree(pre, in, sin, idx-1, spr+1, spr+x);
    root.right = buildTree(pre, in, idx+1, ein, spr+x+1, epr);

    return root;
}
```

TC : O(N)

SC : O(N)

↑ HashMap + recursive call stack.

---