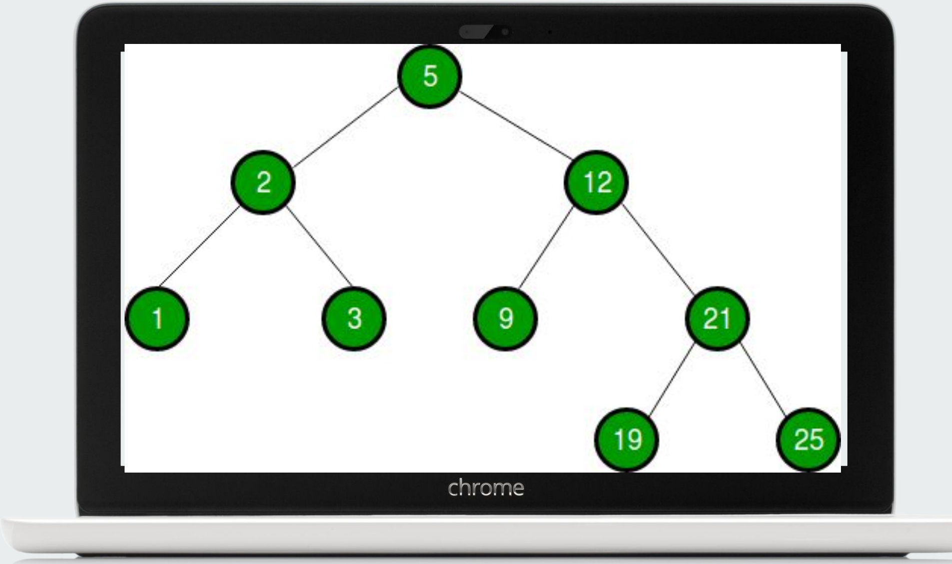
# Binary Search Tree (BST)

Group 1

Parikha Goyanka

Anushka Saxena

Aman Bahuguna




Summer Internship 2020, Internity Foundation

# Trees- a brief

## Properties

- Non linear data structure
- Inheriarcal Data structure
- Consist of nodes connected by edges

## Why tree?

- Quicker and easier access
- to data
- Lower Time complexity
- than linear data structure

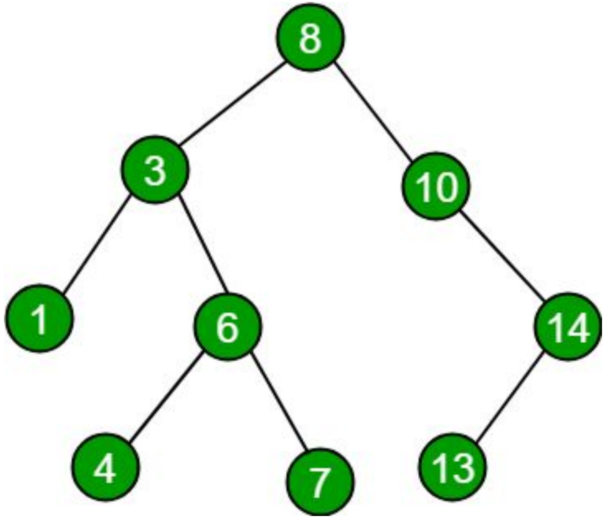
## Terminologies

- Edge
- Node
  - Root
  - Internal node
  - external(leaf) node
- Height/depth
- Degree of node

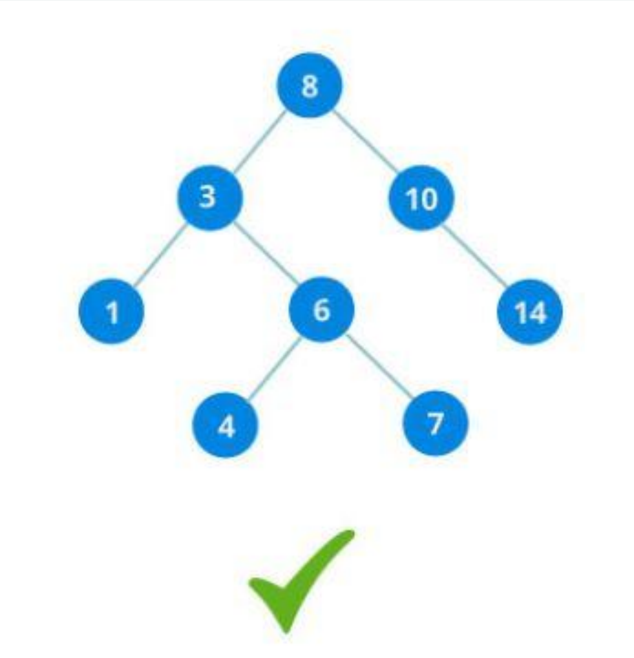
# What is a Binary Search Tree

A binary search tree is data structure that fall under ‘tree category’, which allow to maintain the sorted arrangement of numbers.

- It is called a ‘binary tree’ because each tree node can have maximum two children
- It is called a ‘search tree’ because it can be used to search for a number in O(log n) time.



T



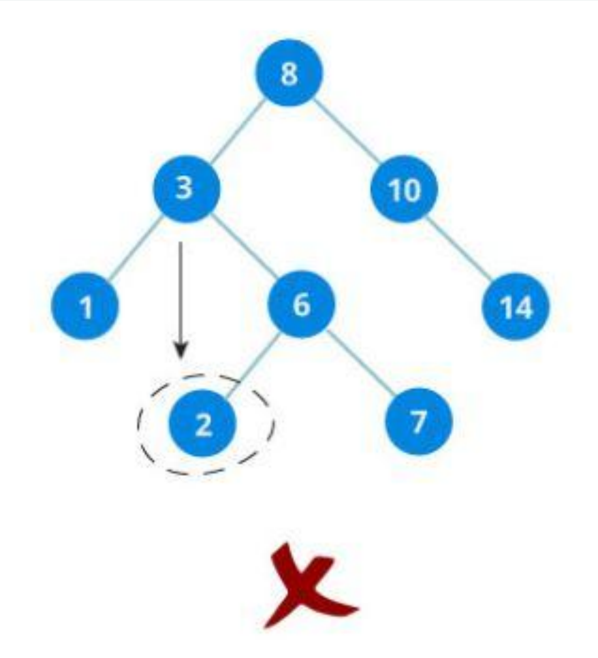

search

than roo

ore than

also BST

les/keys



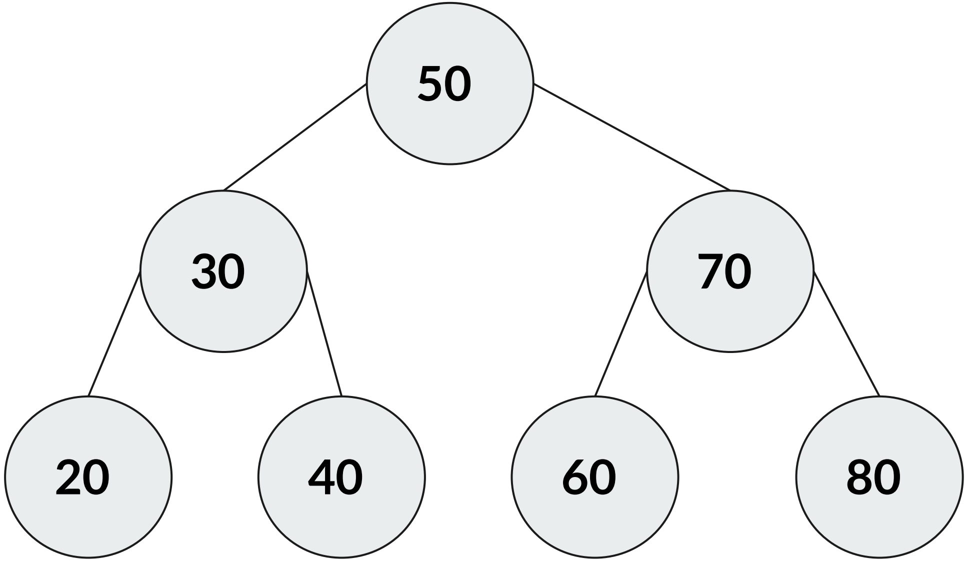

perties

# Basic operations

| | |
|---|---|
| **Insertion** | Insertion of of elements in a Binary Search Tree<br>O(log n) |
| **Searching** | Searching elements in a Binary Search Tree whether they are present in it or not<br>O(log n) |
| **Deletion** | Deleting elements from Binary Search Tree in such a way that it donot loss its properties<br>O(log n) |
| **Traversing** | Visiting every node of tree<br>O(n) |

# Insertion in BST

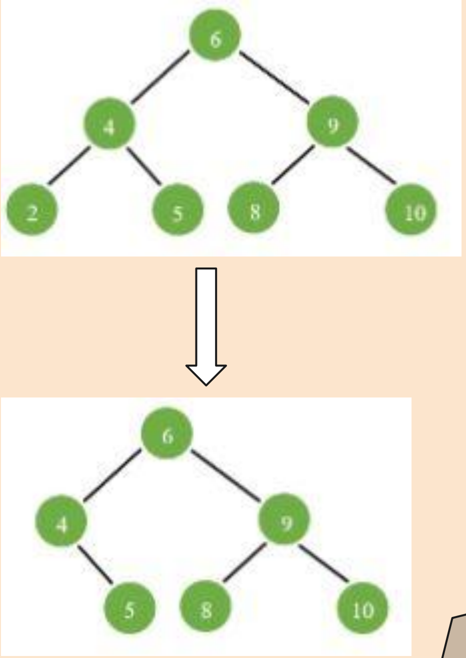
```
Struct nodeStruct newNode(int
key)
struct node* insert(struct node*
node, int key)

{node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);}
```



# Deletion in BST

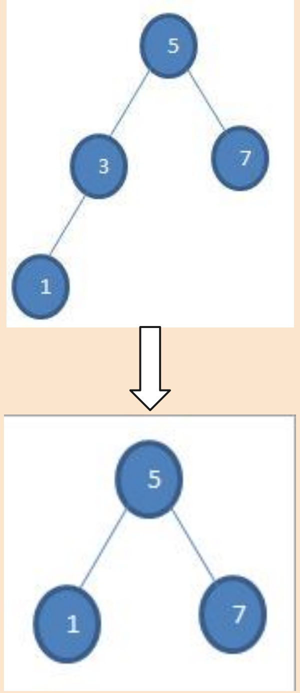
## CASE 1: Node is root with no child

(Dlt(2))



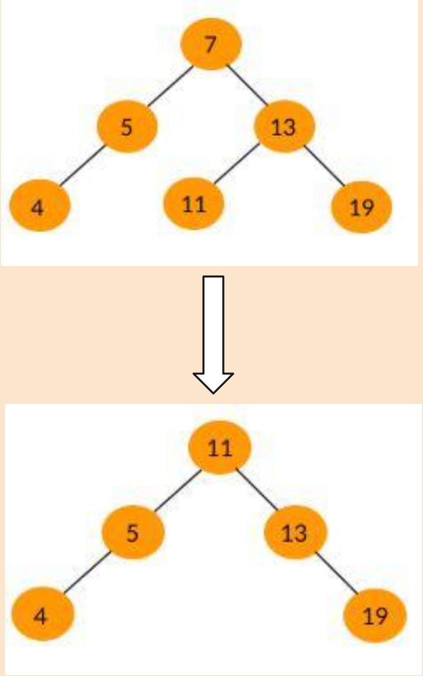

## CASE 2: Node is root with one child

(Dlt(3))




## CASE 3: Node is root with two children

(Dlt(7))



# Binary Search Tree Traversal

### In-Order Traversal

```
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);

    cout << node->data << " ";

    printInorder(node->right);
}
```

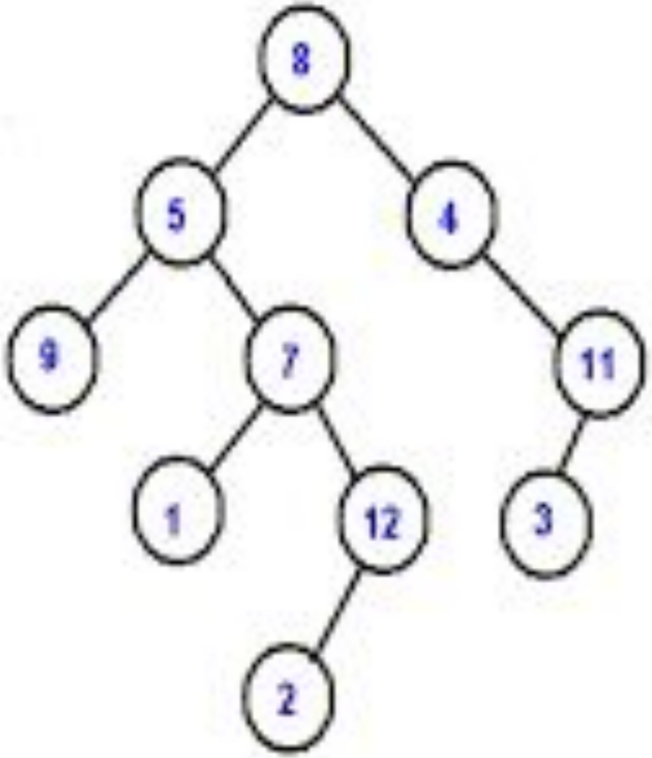
### Pre-Order Traversal

```
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    cout << node->data << " ";

    printPreorder(node->left);

    printPreorder(node->right);
}
```

### Post-Order Traversal

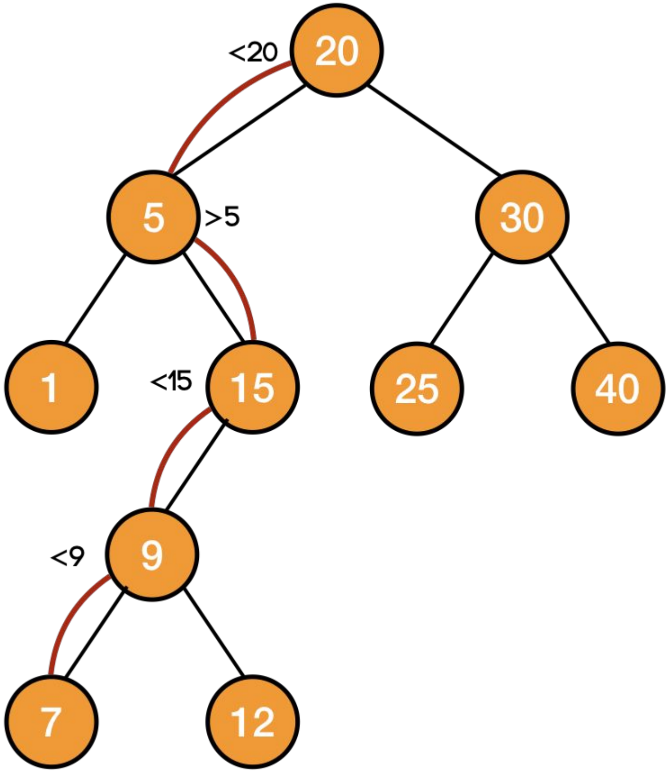
```
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    printPostorder(node->left);

    printPostorder(node->right);

    cout << node->data << " ";
}
```

PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2



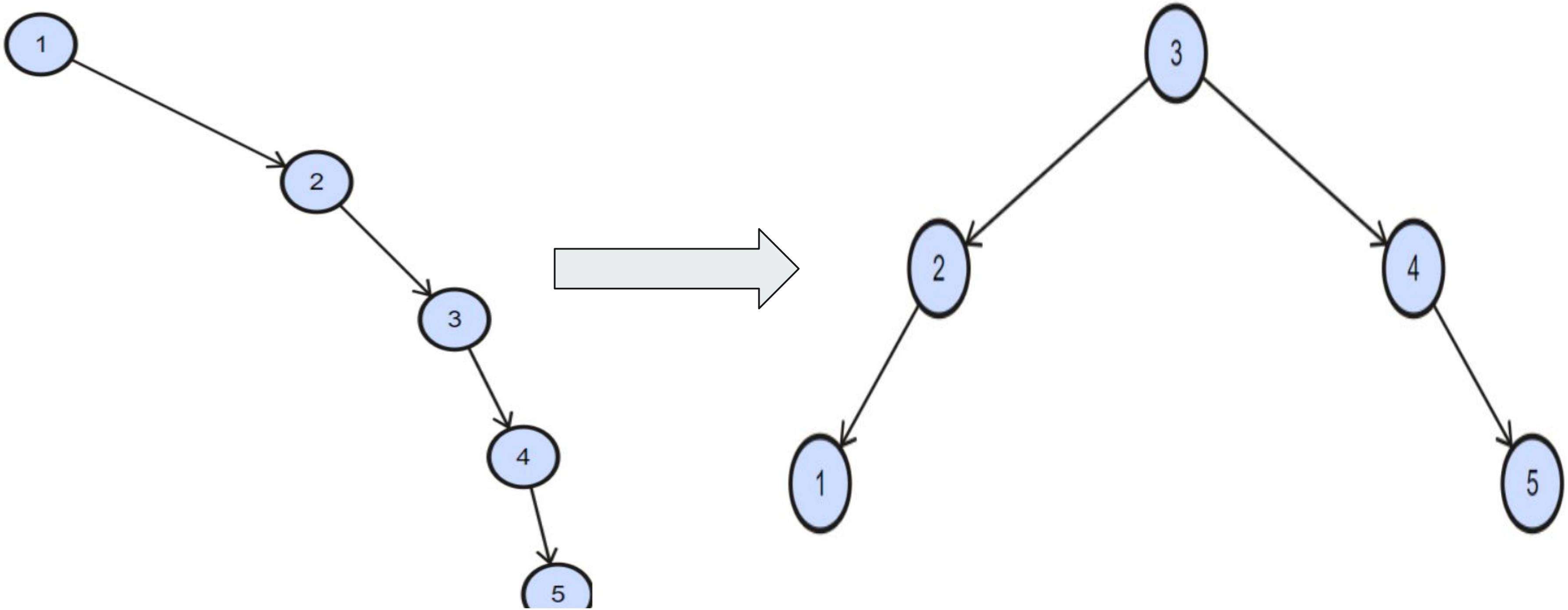
# Searching in BST

**Search(7) and Search(8)**




- 7 is found
- 8 is not found

# Balanced Binary Search Tree



# Conversion of BST to Balanced BST

### Storing its Nodes Pointers in Vector Nodes

```
void storeBSTNodes(Node* root,
        vector<Node*> &nodes)
        {
          if (root==NULL)
                return;
            storeBSTNodes(root->left,
                nodes);
        nodes.push_back(root);
          storeBSTNodes(root->right,
                nodes);
        }
```

### Constructing BST by Recursion

```
Node* buildTreeUtil(vector<Node*>
        &nodes, int start,int end)
        {
          if (start > end)
                return NULL;

        int mid = (start + end)/2;
        Node *root = nodes[mid];

    root->left  = buildTreeUtil(nodes,
                start, mid-1);
   root->right = buildTreeUtil(nodes,
                mid+1, end);
                 return root;
        }
```

### Unbalanced BST to Balanced BST

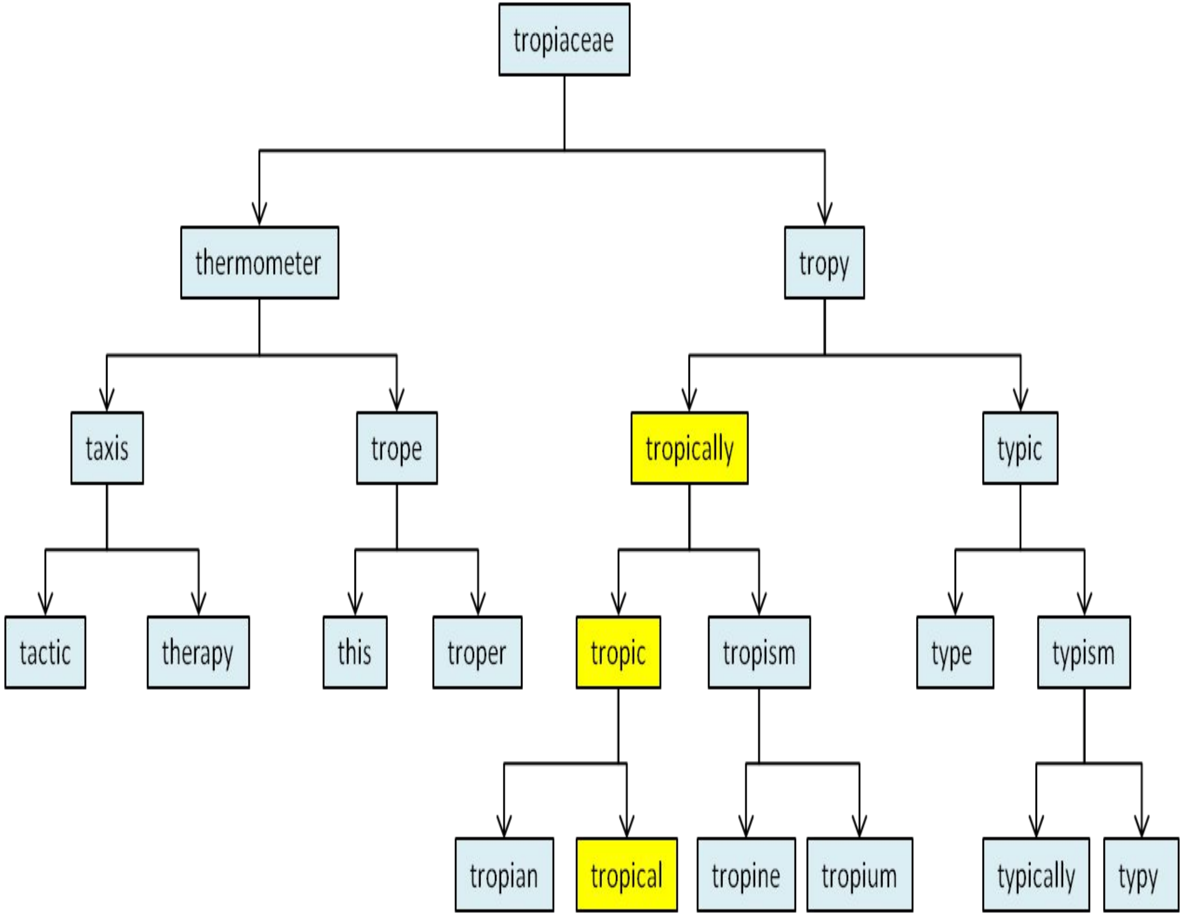
```
Node* buildTree(Node* root)
{
    vector<Node *> nodes;
    storeBSTNodes(root, nodes);

    // Constucts BST from nodes[]
    int n = nodes.size();
    return buildTreeUtil(nodes, 0,
n-1);
}
```

# Real life application of BST

1. It is used to implement dictionary.
2. It is used to implement multilevel indexing in DATABASE.
3. File compression using Huffman Coding Algorithm.
4. It is used to implement searching Algorithm.
5. Implementing routing table in router.
6. There are many more algorithm problems where a Self-Balancing BST is the best suited data structure, like count smaller elements

# Use of BST in dictionary

- In **dictionary** entries are maintained in alphabetical order (ie. sorted order)
- So when to search a word the  inorder traversal of a binary search **tree** is taken place.



# Applications contd.

There are many more algorithm problems where a Self-Balancing BST is the best suited data structure, like count smaller elements on right, Smallest Greater Element on Right Side, etc.


THANK
YOU!