# Data Structures and Algorithms

**Prepared by: Mohamed Ayman**

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

**spring 2020**

sw.eng.MohamedAyman@gmail.com

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

# Lecture 9

# Binary Search Tree

# Course Roadmap


## Part 2: Non-Linear Data Structures

Lecture 8: Binary Tree

**Lecture 9: Binary Search Tree**

Lecture 10: Self Balancing Binary Search Tree

Lecture 11: Binary Heap Tree

Lecture 12: Hash Table

Lecture 13: Graph

Lecture 14: STL in C++ (Non-Linear Data Structures)

# Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Binary Search Tree

2- Insertion Operation

3- Deletion Operation

4- Search Operation

5- Traverse Operation

6- Balanced Binary Tree Property

7- Time Complexity & Space Complexity


Let's
START UP

# Lecture Agenda

Section 1: Introduction to Binary Search Tree

Section 2: Insertion Operation

Section 3: Deletion Operation

Section 4: Search Operation

Section 5: Traverse Operation

Section 6: Balanced Binary Tree Property

Section 7: Time Complexity & Space Complexity

# Introduction to Binary Search Tree

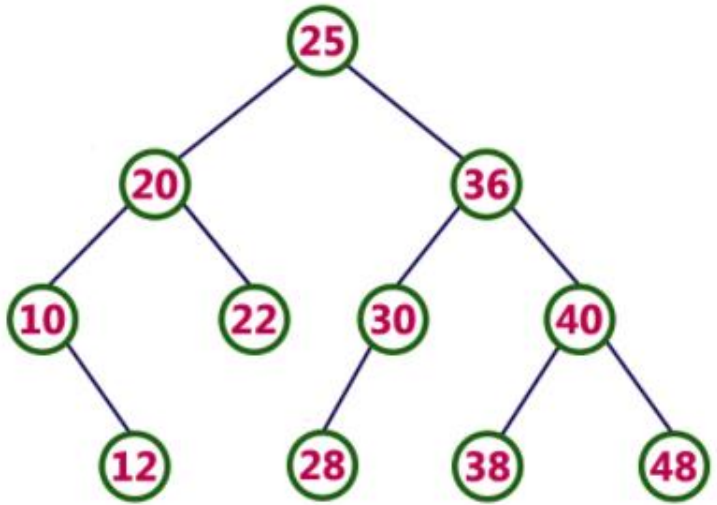
- Binary search trees (BSTs) are very simple to understand. We start with a root node with value x, where the left subtree of x contains nodes with values ‹ x and the right subtree contains nodes whose values are › x. Each node follows the same rules with respect to nodes in their left and right subtrees.
- BSTs are interest because they have operations which are favorably fast: insertion, look up, and deletion can all be done in Θ(log n) time. It is important to note that the Θ(log n) times for these operations can only be attained if the BST is reasonably balanced.
- A binary search tree is a good solution when you need to represent types that are ordered according to some custom rules inherent to that type. With logarithmic insertion, lookup, and deletion it is very efficient. Traversal remains linear, but there are many ways in which you can visit the nodes of a tree. Trees are recursive data structures, so typically you will find that many algorithms that operate on a tree are recursive.
- The run times presented in this topic are based on a pretty big assumption that the binary search tree's left and right subtrees are reasonably balanced. We can only attain logarithmic run times for the algorithms presented earlier when this is true. A binary search tree does not enforce such a property, and the run times for these operations on a pathologically unbalanced tree become linear: such a tree is effectively just a linked list.

# Introduction to Binary Search Tree

- A binary search tree is organized, as the name suggests, in a binary tree. We can represent such a tree by a linked data structure in which each node is an object. In addition to a key and satellite data, each node contains attributes left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL. The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property.

- A binary search tree node

```
struct node {
    int data;
    node* left;
    node* right;
};
```



# Lecture Agenda

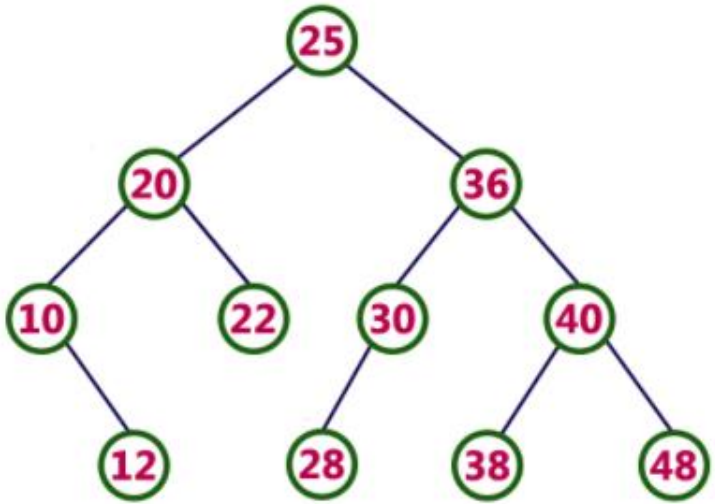
- ✓ Section 1: Introduction to Binary Search Tree
- **Section 2: Insertion Operation**
- Section 3: Deletion Operation
- Section 4: Search Operation
- Section 5: Traverse Operation
- Section 6: Balanced Binary Tree Property
- Section 7: Time Complexity & Space Complexity

# Insertion Operation - Binary Search Tree

- The insertion algorithm is split for a good reason. If the tree is empty then we simply create our root node and finish. In all other cases we invoke the recursive insert node algorithm which simply guides us to the first appropriate place in the tree to put value. Note that at each stage we perform a binary chop: we either choose to recursive into the left subtree or the right by comparing the new value with that of the current node. For any totally ordered type, no value can simultaneously satisfy the conditions to place it in both subtrees. left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

- ***Insert Algorithm***:

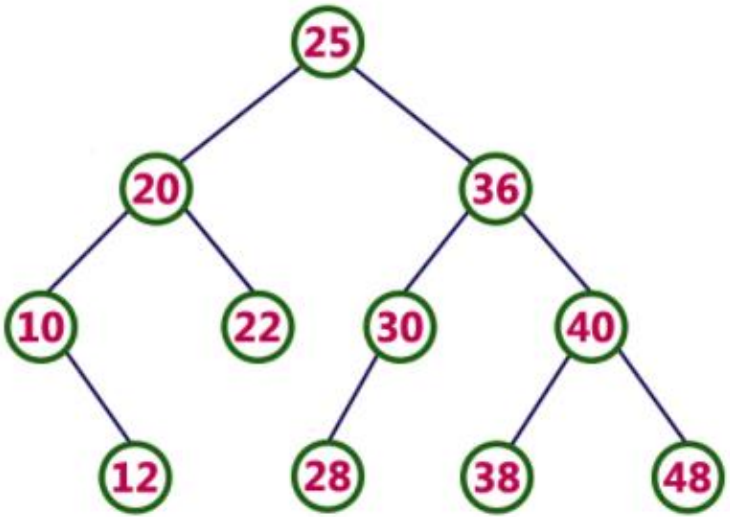
1. ***if curr == NULL then create a node***
2. ***if data < curr then***
3. ***go to step 1 with left node of curr***
4. ***if data > curr then***
5. ***go to step 1 with right node of curr***
6. ***return curr***

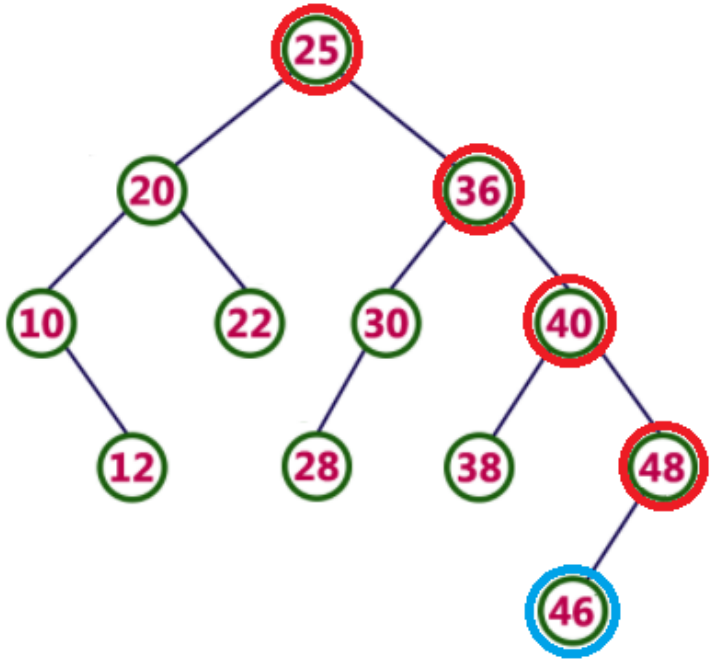


# Insertion Operation - Binary Search Tree

- Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

- Insert 46 in this binary search tree



# Insertion Operation - Binary Search Tree

```
// This function inserts a new node with given key in the binary search tree
node* insert_node(node* curr, int new_data) {
    // base case we reach a null node
    if (curr == NULL) {
        curr = new node();
        curr->data = new_data;
        return curr;
    }
    // repeat the same definition of insert at left or right subtrees
    if (new_data < curr->data)
        curr->left = insert_node(curr->left, new_data);
    else if (new_data > curr->data)
        curr->right = insert_node(curr->right, new_data);
    // return the unchanged node pointer
    return curr;
}
```

Binary-Search-Tree.cpp

# Lecture Agenda

- ✓ Section 1: Introduction to Binary Search Tree
- ✓ Section 2: Insertion Operation
- **Section 3: Deletion Operation**
- Section 4: Search Operation
- Section 5: Traverse Operation
- Section 6: Balanced Binary Tree Property
- Section 7: Time Complexity & Space Complexity

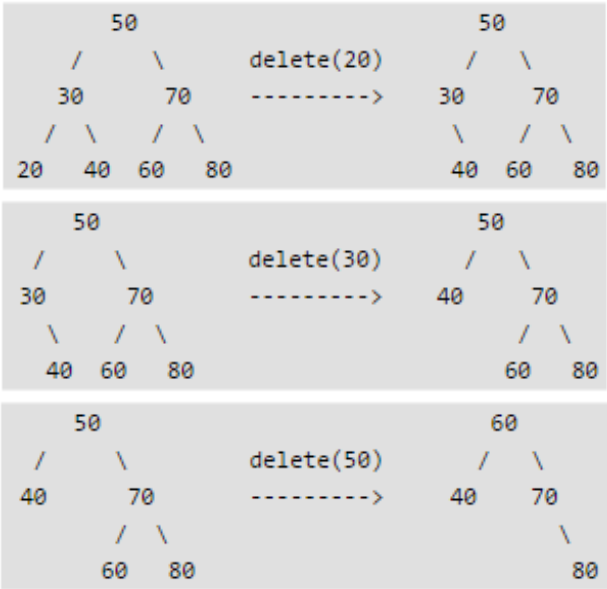
# Deletion Operation - Binary Search Tree

- Given a binary tree, delete a node from it by making sure that tree shrinks from the bottom (i.e. the deleted node is replaced by bottom most and rightmost node). This different from BST deletion. Here we do not have any order among elements, so we replace with last element.
- Removing a node from a BST is fairly straightforward, with four cases to consider:

Case 1: The value to remove is a leaf node

Case 2: The value to remove has a right subtree, but no left subtree or
The value to remove has a left subtree, but no right subtree

Case 3: The value to remove has both a left and right subtree in which case we promote the largest value in the left subtree or
the smallest value in the right subtree.

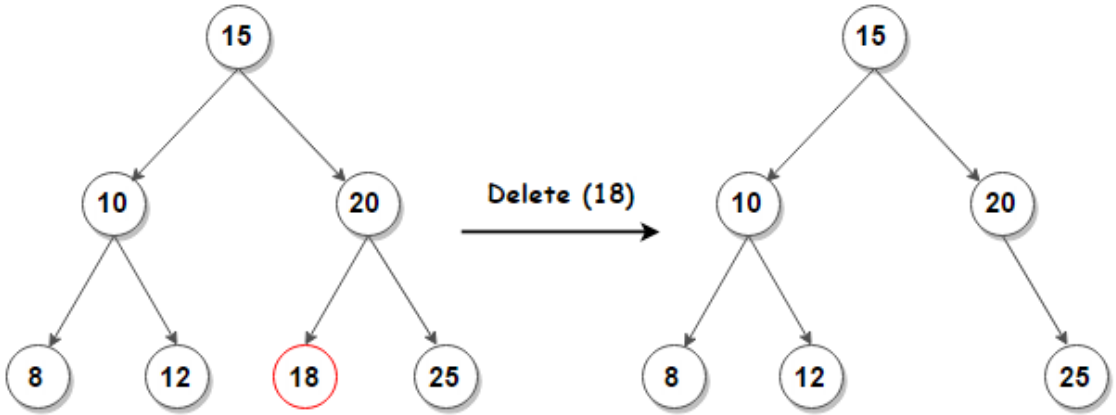


# Deletion Operation - Binary Search Tree

➢ Node to be deleted is leaf: Simply remove from the tree.

- ***Delete Algorithm***:

1. ***if curr == NULL then return NULL***
2. ***if data < curr then***
3. ***go to step 1 with left node of curr***
4. ***if data > curr then***
5. ***go to step 1 with right node of curr***
6. ***if case 1 of deletion then delete it***

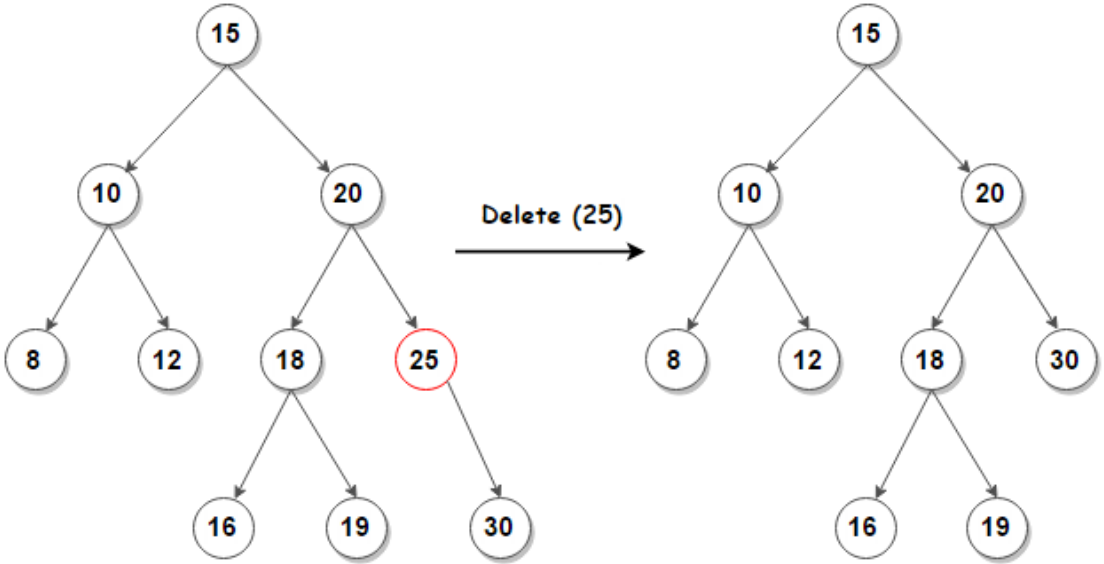


# Deletion Operation - Binary Search Tree

➢ Node to be deleted has only one child: Copy the child to the node and delete the child.

- ***Delete Algorithm***:

1. ***if curr == NULL then return NULL***
2. ***if data < curr then***
3. ***go to step 1 with left node of curr***
4. ***if data > curr then***
5. ***go to step 1 with right node of curr***
6. ***if case 1 of deletion then delete it***
7. ***if case 2 of deletion then***
8. ***reconnect the single child***

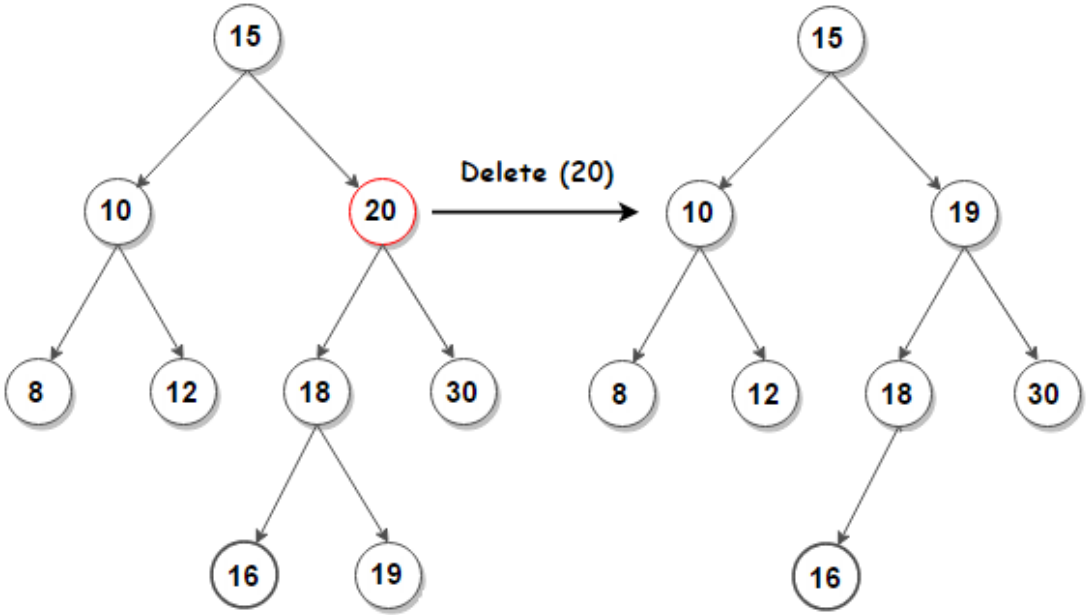


# Deletion Operation - Binary Search Tree

- Node to be deleted has two children: Find in-order successor of the node. Copy contents of the inorder successor to the node and delete the in-order successor.

- ***Delete Algorithm***:

1. ***if curr == NULL then return NULL***
2. ***if data < curr then***
3. ***go to step 1 with left node of curr***
4. ***if data > curr then***
5. ***go to step 1 with right node of curr***
6. ***if case 1 of deletion then delete it***
7. ***if case 2 of deletion then***
8. ***reconnect the single child***
9. ***if case 3 of deletion then***
10. ***get the nearest node***
11. ***swap deleted ndoe with the nearest node***
12. ***delete the nearest node***



# Min & Max Methods - Binary Search Tree

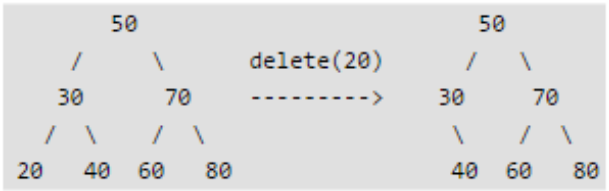
```
// This function returns the node with minimum value found in the given tree
node* min_node(node* curr) {
    node* res = curr;
    // loop down to find the leftmost leaf
    while (res->left != NULL)
        res = res->left;
    // return the nearest data node of curr data node
    return res;
}

// This function returns the node with maximum value found in the given tree
node* max_node(node* curr) {
    node* res = curr;
    // loop down to find the rightmost leaf
    while (res->right != NULL)
        res = res->right;
    // return the nearest data node of curr data node
    return res;
}
```

Binary-Search-Tree.cpp

# Deletion Operation - Binary Search Tree

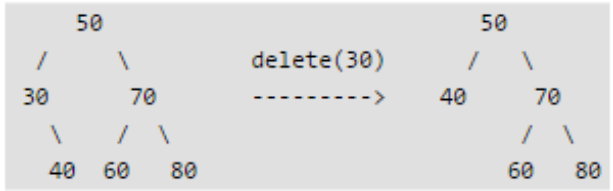
```
// This function deletes the given data in the binary search tree if exists
node* delete_node(node* curr, int data) {
    // base case we reach a null node
    if (curr == NULL)
        return curr;
    // repeat the same definition of insert at left or right subtrees
    if (data < curr->data)
        curr->left = delete_node(curr->left, data);
    else if (data > curr->data)
        curr->right = delete_node(curr->right, data);
    // if the given data is same as curr's data, then we will delete this node
    else {
        // node with no child
        if (curr->left == NULL && curr->right == NULL) {
            delete(curr);
            return NULL;
        }
```

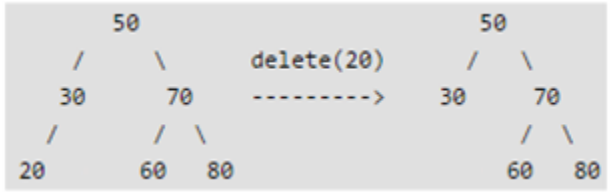



Binary-Search-Tree.cpp

# Deletion Operation - Binary Search Tree

```
// if the given data is same as curr's data, then we will delete this node
else {
    // node with no child
    if (curr->left == NULL && curr->right == NULL) {
        delete(curr);
        return NULL;
    }
    // node with only one child
    else if (curr->left == NULL) {
        node* temp = curr->right;
        delete(curr);
        return temp;
    }
    // node with only one child
    else if (curr->right == NULL) {
        node* temp = curr->left;
        delete(curr);
        return temp;
    }
```




Binary-Search-Tree.cpp

# Deletion Operation - Binary Search Tree

```
        // node with two children
        else {
            // get the inorder successor (smallest in the right subtree)
            node* temp = min_node(curr->right);
            // copy the inorder successor's content to this node
            curr->data = temp->data;
            // delete the inorder successor
            curr->right = delete_node(curr->right, temp->data);
        }
    }
    // return the unchanged node pointer
    return curr;
}
```

Binary-Search-Tree.cpp

# Lecture Agenda

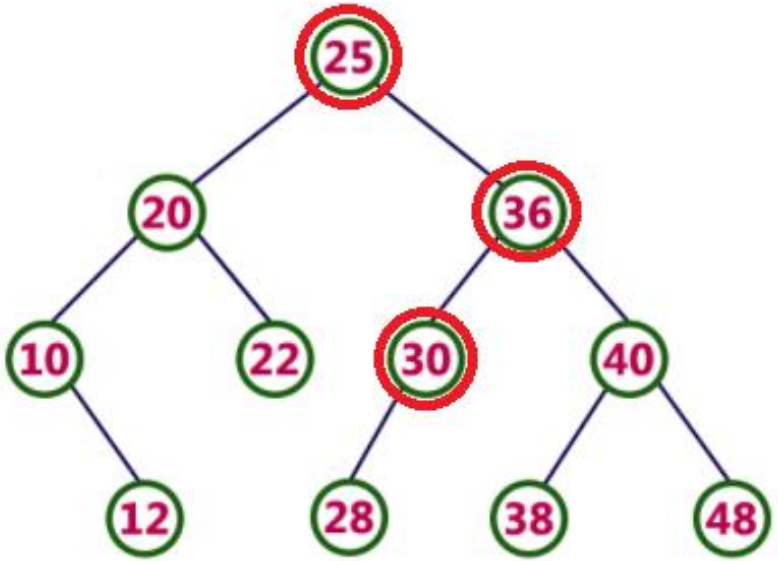
- ✓ Section 1: Introduction to Binary Search Tree
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- **Section 4: Search Operation**
- Section 5: Traverse Operation
- Section 6: Balanced Binary Tree Property
- Section 7: Time Complexity & Space Complexity

# Search Operation - Binary Search Tree

- Searching a BST is even simpler than insertion. The pseudo-code is self-explanatory but we will look briefly at the premise of the algorithm nonetheless. We have talked previously about insertion, we go either left or right with the right subtree containing values that are › x where x is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

- ***Search Algorithm***:

1. ***if curr == NULL then return false***
2. ***if curr == data then return true***
3. ***if data < curr then***
4. ***go to step 1 with left node of curr***
5. ***if data > curr then***
6. ***go to step 1 with right node of curr***




- Example Searching for 30.

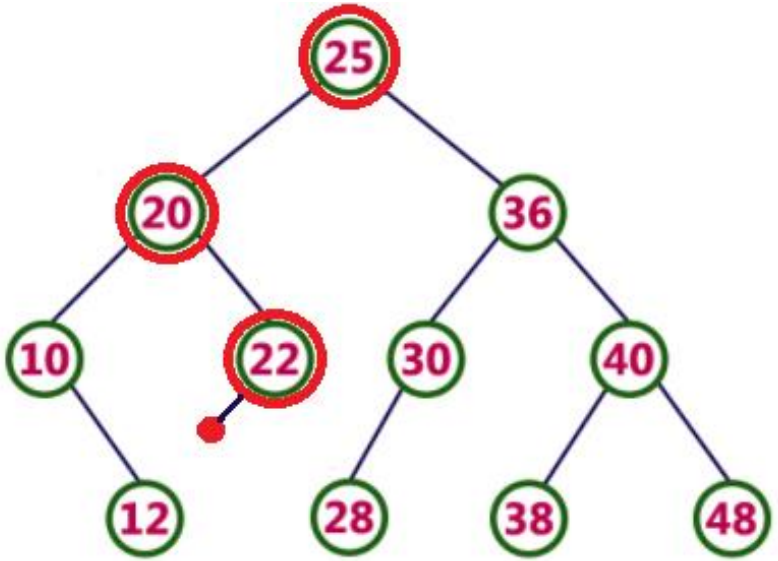
# Search Operation - Binary Search Tree

- Searching a BST is even simpler than insertion. The pseudo-code is self-explanatory but we will look briefly at the premise of the algorithm nonetheless. We have talked previously about insertion, we go either left or right with the right subtree containing values that are › x where x is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

- ***Search Algorithm***:

1. ***if curr == NULL then return false***
2. ***if curr == data then return true***
3. ***if data < curr then***
4. ***go to step 1 with left node of curr***
5. ***if data > curr then***
6. ***go to step 1 with right node of curr***

- Example Searching for 21.



# Search Operation - Binary Search Tree

```
// This function searches if the given data in the binary search tree
bool search(node* curr, int data) {
    // base case we reach a null node
    if (curr == NULL)
        return false;
    // check if we find the data at the curr node
    if (curr->data == data)
        return true;
    // repeat the same definition of search at left or right subtrees
    if (data < curr->data)
        return search(curr->left, data);
    else
        return search(curr->right, data);
}
```

Binary-Search-Tree.cpp

# Lecture Agenda

- ✓ Section 1: Introduction to Binary Search Tree
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Search Operation
- **Section 5: Traverse Operation**
- Section 6: Balanced Binary Tree Property
- Section 7: Time Complexity & Space Complexity

# Traverse Operation - Binary Search Tree


- There are various strategies which can be employed to traverse the items in a tree. the choice of strategy depends on which node visitation order you require. In this section we will touch on the traversals that DSA provides on all data structures that derive from Binary Search Tree.
- Preorder: When using this algorithm, you visit the root first, then traverse the left subtree and finally traverse the right subtree.
- Inorder: When using this algorithm, you traverse the left subtree first, then visit the root and finally traverse the right subtree.
- Postorder: When using this algorithm, you traverse the left subtree first, then traverse the right subtree and finally visit the root.
- Breadth First: Traversing a tree in breadth first order yields the values of all nodes of a particular depth in the tree before any deeper ones. In other words, given a depth d we would visit the values of all nodes at d in a left to right fashion, then we would proceed to d + 1 and so on until we had no more nodes to visit. Traditionally breadth first traversal is implemented using a list (vector, resizable array, etc) to store the values of the nodes visited in breadth first order and then a queue to store those nodes that have yet to be visited.

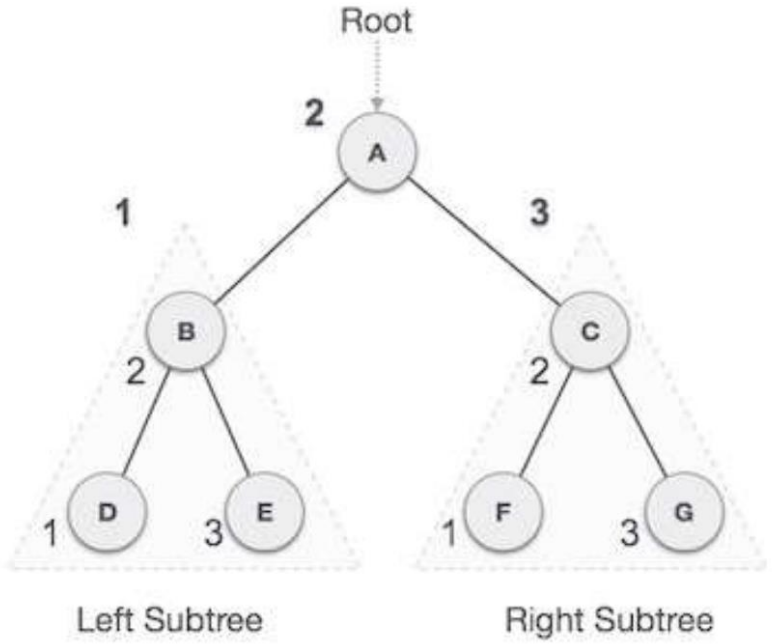
# Traverse Operation - Binary Search Tree (in-order)

- In-order Traversal is the one of DFS Traversal of the tree. Therefore, we will start from the root node of the tree and go deeper-and-deeper into the left subtree with recursive manner. When we will reach to the left-most node with the above steps, then we will visit that current node and go to the left-most node of its right subtree (if exists).

- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***go to step 1 with left node of curr***
3. ***print curr node data***
4. ***go to step 1 with right node of curr***

```
void inOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition
    inOrder(curr->left);
    cout << curr->data << ' ';
    inOrder(curr->right);
}
```



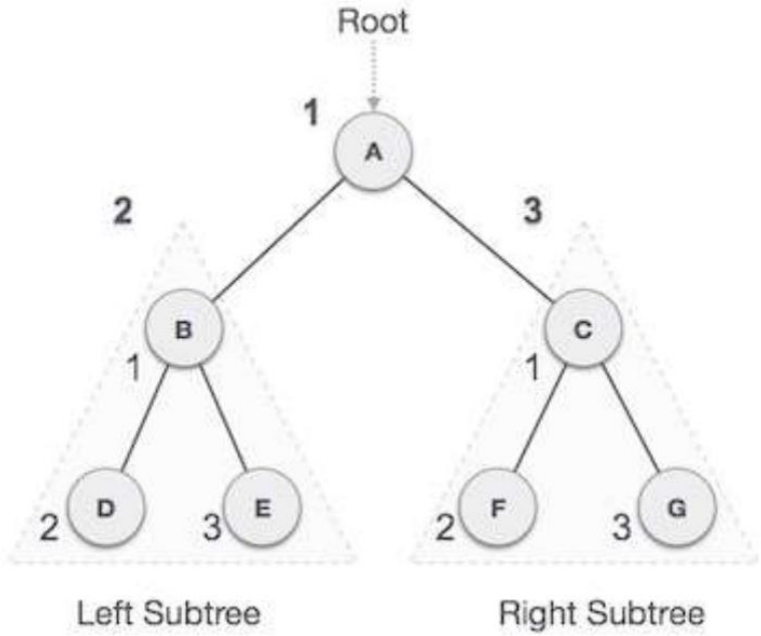
# Traverse Operation - Binary Search Tree (pre-order)

- Pre-order Traversal: is the one of DFS Traversal of the tree. Therefore, we visit the current node first and then goes to the left sub-tree. After covering every node of the left sub-tree, we will move towards the right sub-tree and visit in a similar fashion.

- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***print curr node data***
3. ***go to step 1 with left node of curr***
4. ***go to step 1 with right node of curr***

```
void preOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition
    cout << curr->data << ' ';
    preOrder(curr->left);
    preOrder(curr->right);
}
```



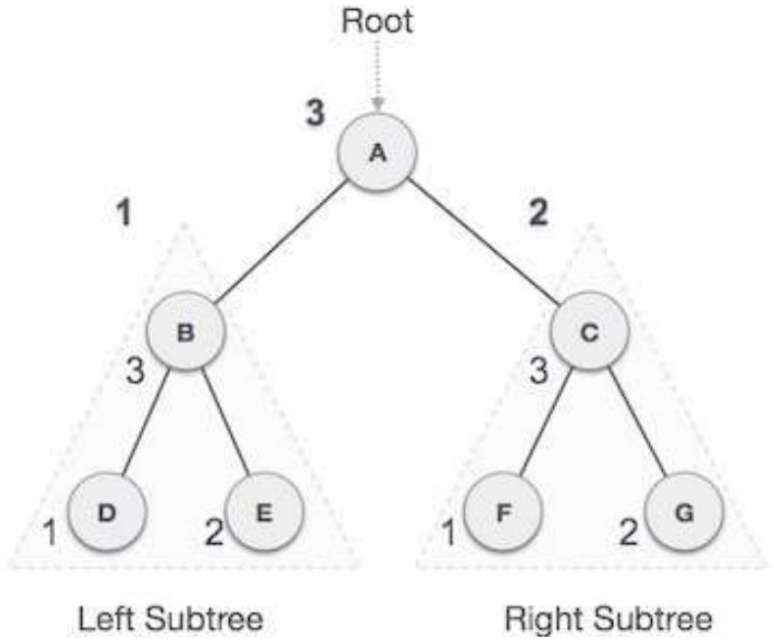
# Traverse Operation - Binary Search Tree (post-order)

- Post-order Traversal: is the one of DFS Traversal of the tree. Therefore, we will start from the root node of the tree and go deeper-and-deeper into the left subtree with recursive manner. When we will reach to the left-most node with the above steps, then we will go to the left-most node of its right subtree (if exists) and finally we will visit that current node.

- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***go to step 1 with left node of curr***
3. ***go to step 1 with right node of curr***
4. ***print curr node data***

```
void postOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition
    postOrder(curr->left);
    postOrder(curr->right);
    cout << curr->data << ' ';
}
```



# Functionality Testing - Binary Search Tree

- Initialize a global struct

```cpp
#include <bits/stdc++.h>
using namespace std;

// A binary search tree node
struct node {
    int data;
    node* left;
    node* right;
};

// Initialize a global pointer for root
node* root;
```

Binary-Search-Tree.cpp

# Functionality Testing - Binary Search Tree

- In the Main function:

```
// first level of the binary search tree (root only)
root = insert_node(root, 100);

cout << "Binary Search Tree in-order   : ";
inOrder(root);
cout << "\n";
cout << "Binary Search Tree pre-order  : ";
preOrder(root);
cout << "\n";
cout << "Binary Search Tree post-order : ";
postOrder(root);
```

- Tree Diagram

100

- Expected Output:

```
Binary Search Tree in-order   : 100
Binary Search Tree pre-order  : 100
Binary Search Tree post-order : 100
```

Binary-Search-Tree.cpp

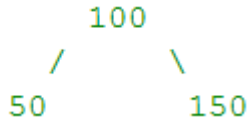
# Functionality Testing - Binary Search Tree

- In the Main function:

```
// second level of the binary search tree
root = insert_node(root, 50);
root = insert_node(root, 150);

cout << "Binary Search Tree in-order   : ";
inOrder(root);
cout << "\n";
cout << "Binary Search Tree pre-order  : ";
preOrder(root);
cout << "\n";
cout << "Binary Search Tree post-order : ";
postOrder(root);
```

- Tree Diagram




- Expected Output:

```
Binary Search Tree in-order   : 50 100 150
Binary Search Tree pre-order  : 100 50 150
Binary Search Tree post-order : 50 150 100
```

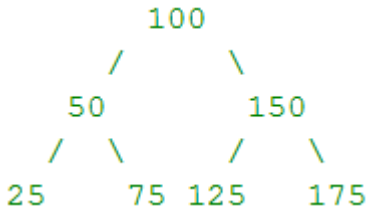
Binary-Search-Tree.cpp

# Functionality Testing - Binary Search Tree

- In the Main function:

```
// third level of the binary search tree
root = insert_node(root, 25);
root = insert_node(root, 75);
root = insert_node(root, 125);
root = insert_node(root, 175);
cout << "Binary Search Tree in-order   : ";
inOrder(root);
cout << "\n";
cout << "Binary Search Tree pre-order  : ";
preOrder(root);
cout << "\n";
cout << "Binary Search Tree post-order : ";
postOrder(root);
```

- Tree Diagram




- Expected Output:

```
Binary Search Tree in-order   : 25 50 75 100 125 150 175
Binary Search Tree pre-order  : 100 50 25 75 150 125 175
Binary Search Tree post-order : 25 75 50 125 175 150 100
```

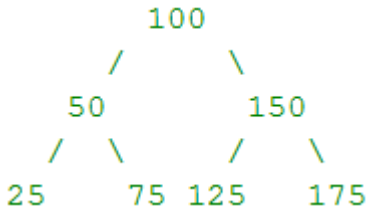
Binary-Search-Tree.cpp

# Functionality Testing - Binary Search Tree

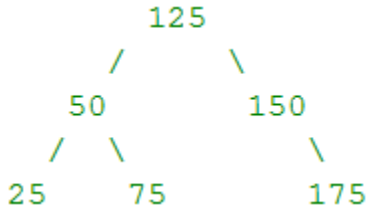
- In the Main function:

```
cout << "deleting node 100 \n";
root = delete_node(root, 100);
cout << "Binary Search Tree in-order   : ";
inOrder(root);
cout << "\n";
cout << "Binary Search Tree pre-order  : ";
preOrder(root);
cout << "\n";
cout << "Binary Search Tree post-order : ";
postOrder(root);
```

- Current Tree Diagram




- Now Tree Diagram




- Expected Output:

```
deleting node 100
Binary Search Tree in-order   : 25 50 75 125 150 175
Binary Search Tree pre-order  : 125 50 25 75 150 175
Binary Search Tree post-order : 25 75 50 175 150 125
```

Binary-Search-Tree.cpp

# Functionality Testing - Binary Search Tree

- In the Main function:

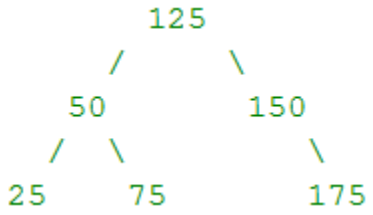
```
if (search(root, 150))
    cout << "element " << 150 << " in the tree\n";
else
    cout << "element " << 150 << " not in the tree\n";

if (search(root, 100))
    cout << "element " << 100 << " in the tree\n";
else
    cout << "element " << 100 << " not in the tree\n";

if (search(root, 75))
    cout << "element " << 75 << " in the tree\n";
else
    cout << "element " << 75 << " not in the tree\n";

if (search(root, 95))
    cout << "element " << 95 << " in the tree\n";
else
    cout << "element " << 95 << " not in the tree\n";
```

- Tree Diagram




- Expected Output:

```
element 150 in the tree
element 100 not in the tree
element 75 in the tree
element 95 not in the tree
```

Binary-Search-Tree.cpp

# Functionality Testing - Binary Search Tree

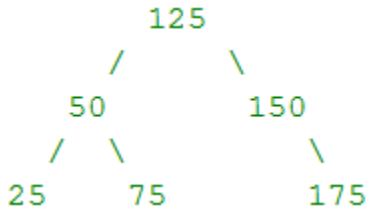
- In the Main function:

```
cout << "adding the following elements 135 80 130\n";
root = insert_node(root, 135);
root = insert_node(root, 80);
root = insert_node(root, 130);
cout << "Binary Search Tree in-order   : ";
inOrder(root);
cout << "\n";
cout << "Binary Search Tree pre-order  : ";
preOrder(root);
cout << "\n";
cout << "Binary Search Tree post-order : ";
postOrder(root);
```

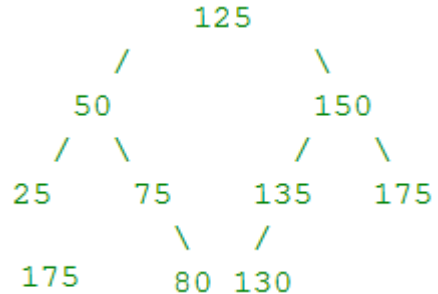
- Expected Output:

```
adding the following elements 135 80 130
Binary Search Tree in-order   : 25 50 75 80 125 130 135 150 175
Binary Search Tree pre-order  : 125 50 25 75 80 150 135 130 175
Binary Search Tree post-order : 25 80 75 50 130 135 175 150 125
```

- Current Tree Diagram




- Now Tree Diagram



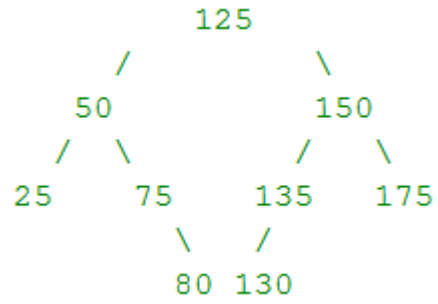

Binary-Search-Tree.cpp

# Functionality Testing - Binary Search Tree

- In the Main function:

```
cout << "The minimum value in the tree of root 125 is: ";
cout << min_node(root)->data << '\n';
cout << "The maximum value in the tree of root 125 is: ";
cout << max_node(root)->data << '\n';
cout << "The minimum value in the tree of root 50  is: ";
cout << min_node(root->left)->data << '\n';
cout << "The maximum value in the tree of root 50  is: ";
cout << max_node(root->left)->data << '\n';
cout << "The minimum value in the tree of root 150 is: ";
cout << min_node(root->right)->data << '\n';
cout << "The maximum value in the tree of root 150 is: ";
cout << max_node(root->right)->data << '\n';
```

- Tree Diagram




- Expected Output:

```
The minimum value in the tree of root 125 is: 25
The maximum value in the tree of root 125 is: 175
The minimum value in the tree of root 50  is: 25
The maximum value in the tree of root 50  is: 80
The minimum value in the tree of root 150 is: 130
The maximum value in the tree of root 150 is: 175
```

Binary-Search-Tree.cpp

# Functionality Testing - Binary Search Tree


- In the Main function:

```
delete_tree(root);
root = NULL;

cout << "Binary Search Tree in-order   : ";
inOrder(root);
cout << "\n";
cout << "Binary Search Tree pre-order  : ";
preOrder(root);
cout << "\n";
cout << "Binary Search Tree post-order : ";
postOrder(root);
```

- Expected Output:

```
Binary Search Tree in-order   :
Binary Search Tree pre-order  :
Binary Search Tree post-order :
```

Binary-Search-Tree.cpp

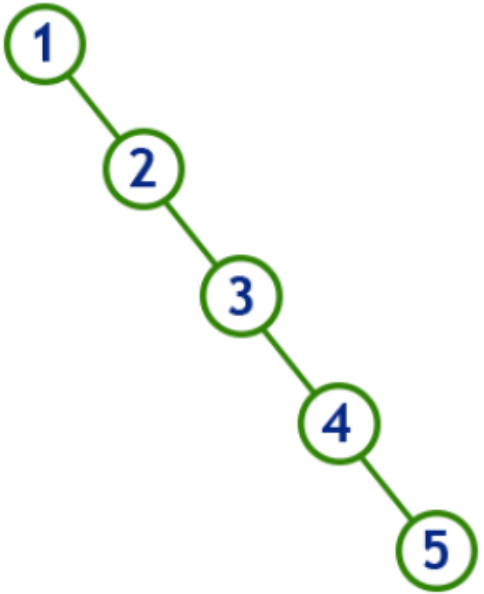
# Lecture Agenda

- ✓ Section 1: Introduction to Binary Search Tree
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Search Operation
- ✓ Section 5: Traverse Operation
- **Section 6: Balanced Binary Tree Property**
- Section 7: Time Complexity & Space Complexity

# Balanced Binary Tree Property

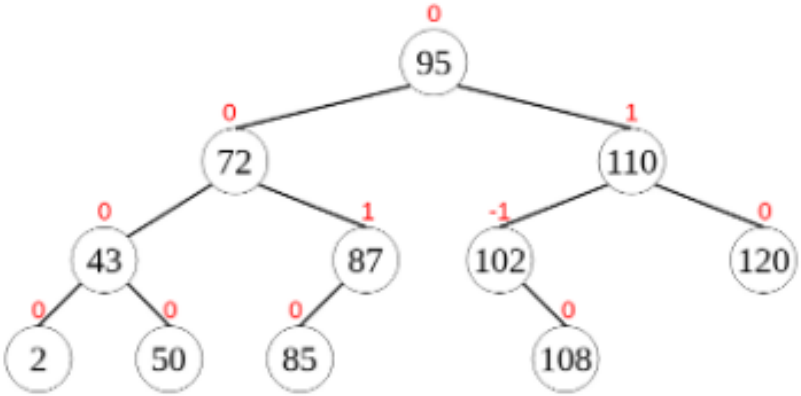
- It is important to note that the Θ(log n) times for these operations can only be attained if the BST is reasonably balanced; for a tree data structure with self balancing properties see AVL tree. Height for a Balanced Binary Tree is Θ(log n). Worst case occurs for skewed tree and worst case height becomes Θ(n), So in worst case extra space required is Θ(n) for both.
- Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced. A non-empty binary tree T is balanced if: Left sub-tree of T is balanced or Right sub-tree of T is balanced or The difference between heights of left sub-tree and right sub-tree is not more than 1.
- The average time complexity of BST operations is Θ(h), where h is the height of the tree. Hence having the height as small as possible is better when it comes to performing a large number of operations. Hence, self-balancing BSTs were introduced which automatically maintain the height at a minimum. However, you may think having to self-balance every time an operation is performed is not efficient, but this is compensated by ensuring a large number of fast operations which will be performed later on the BST.

- Inserting the following Items in a BST (1, 2, 3, 4, 5)

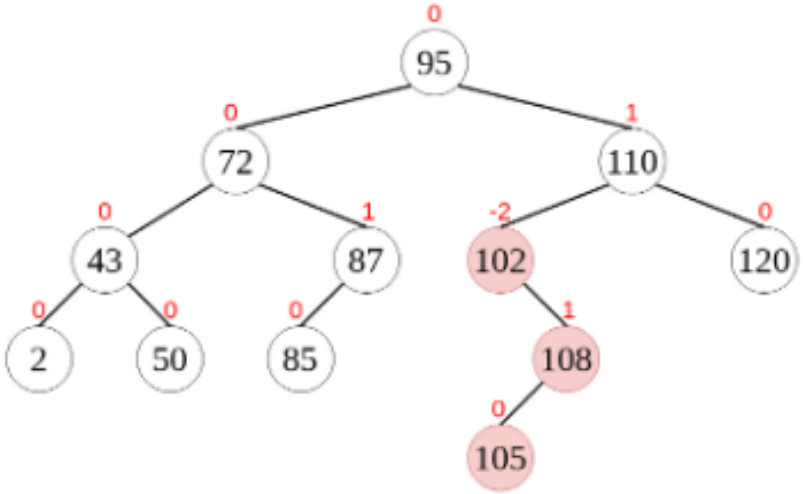


# Balanced Binary Tree Property

- A self-balancing binary search tree (BST) is a binary search tree that automatically tries to keep its height as minimal as possible at all times (even after performing operations such as insertions or deletions).




The tree is balanced




The tree is not balanced

# Lecture Agenda

- ✓ Section 1: Introduction to Binary Search Tree
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Search Operation
- ✓ Section 5: Traverse Operation
- ✓ Section 6: Balanced Binary Tree Property

**Section 7: Time Complexity & Space Complexity**

# Time Complexity & Space Complexity


- Time Analysis

| | Worst Case | Average Case |
|---|---|---|
| • Insert | Θ(n) | θ(log n) |
| • Delete | Θ(n) | θ(log n) |
| • Search | Θ(n) | θ(log n) |
| • Traverse | Θ(n) | θ(n) |

# Lecture Agenda

- ✓ Section 1: Introduction to Binary Search Tree
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Search Operation
- ✓ Section 5: Traverse Operation
- ✓ Section 6: Balanced Binary Tree Property
- ✓ Section 7: Time Complexity & Space Complexity

# Practice

# Practice


1- Traversal a BST in-order in (recursive/iterative) ways
2- Traversal a BST pre-order in (recursive/iterative) ways
3- Traversal a BST post-order in (recursive/iterative) ways
4- Traversal a BST level Order (BFS) one per line
5- Traversal a BST all root-to-leaf paths one per line
6- Check if BST contain a given value in (recursive/iterative) ways
7- Find the maximum value of all nodes in a BST in (recursive/iterative) ways
8- Find the minimum value of all nodes in a BST in (recursive/iterative) ways
9- Calculate height of each node in a BST
10- Check if a binary search tree is balanced
11- Check if a binary tree is BST
12- Find the largest number in a BST which is less than or equal to a given number
13- Print BST keys in a given range in (recursive/iterative) ways
14- Count BST nodes that lie in a given range in (recursive/iterative) ways
15- Count BST subtrees that lie in given range in (recursive/iterative) ways

# Practice

16- Check if the given array can represent level order traversal of BST
17- Check if the given array can represent pre-order traversal of BST
18- Check if the given array can represent in-order traversal of BST
19- Check if the given array can represent post-order traversal of BST
20- Construct BST from its given level order traversal
21- Construct BST from given pre-order traversal
22- Construct BST from given in-order traversal
23- Construct BST from given post-order traversal
24- Binary Tree to Binary Search Tree Conversion
25- Find median of BST
26- Remove BST keys outside a given range in (recursive/iterative) ways
27- Remove all leaf nodes from the BST
28- Sum of k smallest elements in BST in (recursive/iterative) ways
29- Inorder predecessor and successor for a given key in BST by (recursive/iterative) ways
30- Find the maximum element between two nodes of BST in (recursive/iterative) ways

# Practice


31- Find pairs with a given sum such that pair elements lie in different BSTs
32- Find the largest BST subtree in a given binary tree
33- Replace every element with the least greater element on its right
34- Add all greater values to every node in a given BST
35- Check for identical BSTs without building the trees in (recursive/iterative) ways
36- Shortest distance between two nodes in BST
37- Count pairs from two BSTs whose sum is equal to a given x
38- Check if two BSTs contain same set of elements in (recursive/iterative) ways
39- Find lowest common ancestor LCA in a BST
40- Reverse a path in BST using queue
41- Convert a normal BST to balanced BST
42- Merge two balanced BST
43- Calculate the maximum path sum in a BST in (recursive/iterative) ways
44- Calculate the minimum path sum in a BST in (recursive/iterative) ways
45- Maximum difference between node and its ancestor in a BST


DO
MORE.