Data Structures & Algorithms

Prepared by: Mohamed Ayman

Machine Learning Researcher

spring 2019

sw.eng.MohamedAyman@gmail.com

facebook.com/sw.eng.MohamedAyman

linkedin.com/in/eng-mohamed-ayman

codeforces.com/profile/Mohamed_Ayman

Binary Search Tree

Agenda

1- Binary Search Tree Definition

2- Binary Search Tree Representation

3- Basics Operation

4- Insertion Operation

5- Deletion Operation

6- Search Operation

7- Balanced Binary Tree Property


Let's
STARTUP

Agenda

1- Binary Search Tree Definition

2- Binary Search Tree Representation

3- Basic Operations

4- Insertion Operation

5- Deletion Operation

6- Search Operation

7- Balanced Binary Tree Property

Binary Search Tree Definition

- 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 of interest because they have operations which are favorably fast: insertion, look up, and deletion can all be done in O(log n) time.

- It is important to note that the O(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.

Agenda

1- Binary Search Tree Definition

2- Binary Search Tree Representation

3- Basic Operations

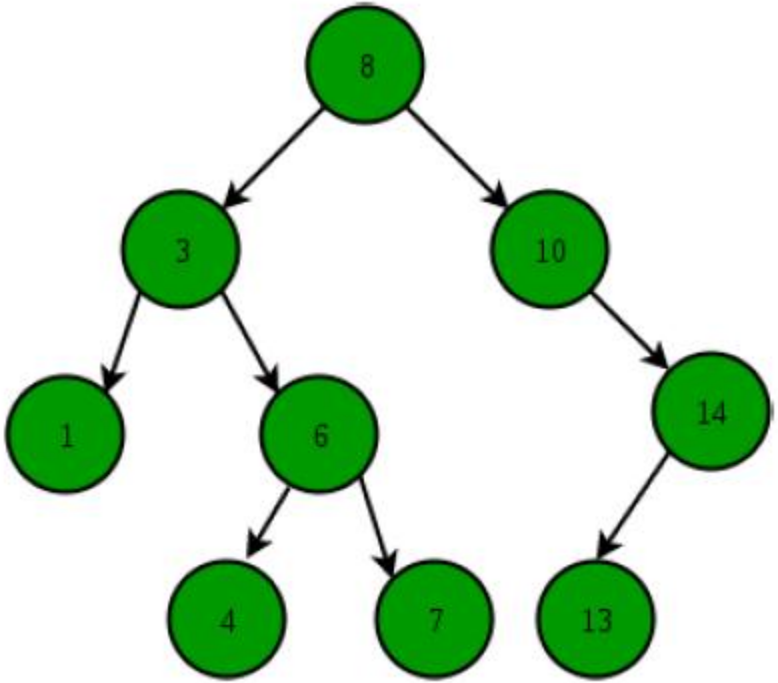
4- Insertion Operation

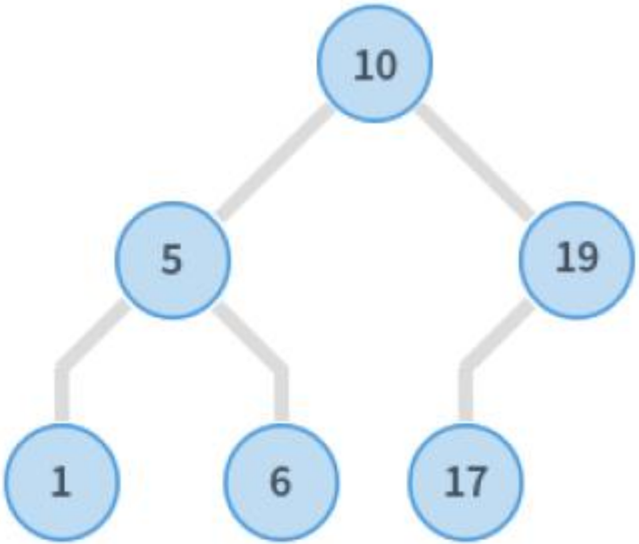
5- Deletion Operation

6- Search Operation

7- Balanced Binary Tree Property

Binary Search Tree Representation





Binary Search Tree Node in C++

Link: repl.it/repls/ZestyIndelibleQuotes

```cpp
// Binary Search Tree Node
struct node {
    int data;
    node *left;
    node *right;
};

node* root;
```

Agenda

1- Binary Search Tree Definition
2- Binary Search Tree Representation
3- Basic Operations
4- Insertion Operation
5- Deletion Operation
6- Search Operation
7- Balanced Binary Tree Property

Basics Operation

- Insert: Inserts an element in a tree/create a tree.
- Search: Searches an element in a tree.
- Delete: Deletes an element in a tree.
- Level-order Traversal: Traverses a tree in a level-order manner.
- Pre-order Traversal: Traverses a tree in a pre-order manner.
- In-order Traversal: Traverses a tree in an in-order manner.
- Post-order Traversal: Traverses a tree in a post-order manner.

Agenda

- ✓ 1- Binary Search Tree Definition
- ✓ 2- Binary Search Tree Representation
- ✓ 3- Basic Operations
- 4- Insertion Operation
- 5- Deletion Operation
- 6- Search Operation
- 7- Balanced Binary Tree Property

Insertion Operation

- The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location.

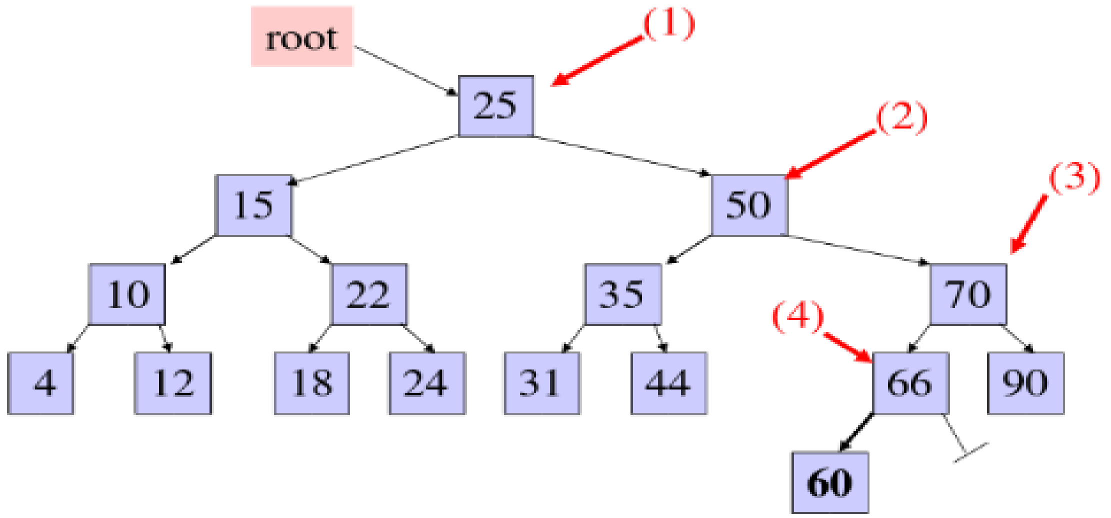
- Start searching from the root node, then if the data is less than the key value, search for the empty location in the left sub-tree and insert the data.

- Otherwise, search for the empty location in the right sub-tree and insert the data.

Insertion Operation

<u>Example: insert 60 in the tree:</u>

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



Insertion Operation in C++

Link: repl.it/repls/ZestyIndelibleQuotes

```cpp
// A utility function to insert a new node with given key in BST
node* insert(node *curr, int data) { // O(h)
    // If the tree is empty, return a new node
    if (curr == NULL) {
        curr = new node();
        curr->data = data;
        return curr;
    }
    // Otherwise, select path to go down the tree
    if (data < curr->data)
        curr->left = insert(curr->left, data);
    else if (data > curr->data)
        curr->right = insert(curr->right, data);
    // return the (unchanged) node pointer
    return curr;
}
```

Agenda

- ✓ 1- Binary Search Tree Definition
- ✓ 2- Binary Search Tree Representation
- ✓ 3- Basic Operations
- ✓ 4- Insertion Operation
- 5- Deletion Operation
- 6- Search Operation
- 7- Balanced Binary Tree Property

Deletion Operation

1) ***Node to be deleted is leaf:*** Simply remove from the tree.

```
              50                            50
           /     \         delete(20)      /   \
          30      70       --------->    30     70
         /  \    /  \                     \    /  \
       20   40  60   80                   40  60   80
```

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
              50                            50
           /     \         delete(30)      /   \
          30      70       --------->    40     70
            \    /  \                          /  \
            40  60   80                       60   80
```

3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
              50                            60
           /     \         delete(50)      /   \
          40      70       --------->    40    70
                 /  \                            \
                60   80                           80
```

Deletion Operation in C++

Link: repl.it/repls/ZestyIndelibleQuotes

```cpp
// Given a binary search tree and a data, this function deletes the data's node
node* deleteNode(node *curr, int data) { // O(h)
    // base case data not found
    if (curr == NULL)
        return curr;
    // If the data to be deleted is smaller than the curr's data,
    // then it lies in left subtree
    if (data < curr->data)
        curr->left = deleteNode(curr->left, data);
    // If the data to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (data > curr->data)
        curr->right = deleteNode(curr->right, data);
    // if key is same as curr's key,
    // then This is the node to be deleted
    else {
        // node with no child
        if (curr->left == NULL && curr->right == NULL) {
            delete(curr);
            return NULL;
        }
```

Deletion Operation in C++

Link: repl.it/repls/ZestyIndelibleQuotes

```
            // node with only one child
            if (curr->left == NULL) {
                node *temp = curr->right;
                delete(curr);
                return temp;
            }
            if (curr->right == NULL) {
                node *temp = curr->left;
                delete(curr);
                return temp;
            }
            // node with two children: Get the inorder successor (smallest in the right subtree)
            node *temp = minValueNode(curr->right);
            // Copy the inorder successor's content to this node
            curr->data = temp->data;
            // Delete the inorder successor
            curr->right = deleteNode(curr->right, temp->data);
        }
        return curr;
    }
```

Agenda

- ✓ 1- Binary Search Tree Definition
- ✓ 2- Binary Search Tree Representation
- ✓ 3- Basic Operations
- ✓ 4- Insertion Operation
- ✓ 5- Deletion Operation
- 6- Search Operation
- 7- Balanced Binary Tree Property

Search Operation

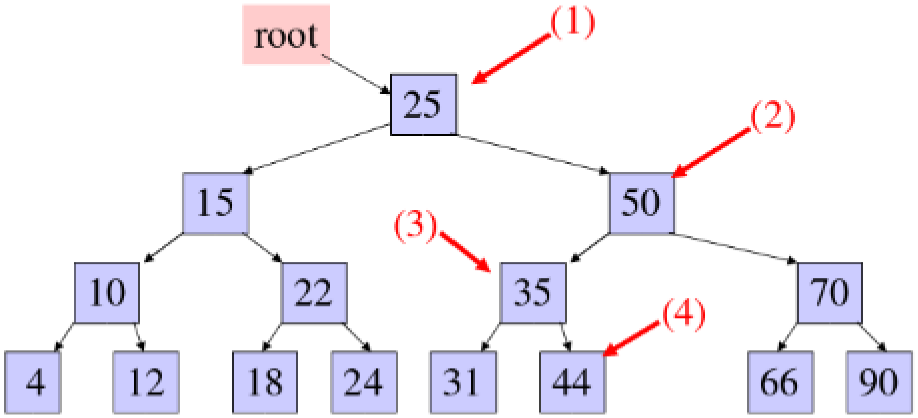
- Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left sub-tree. Otherwise, search for the element in the right sub-tree, finally if node not found return null.

Search Operation

Example: search for 45 in the tree

(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



Search Operation Algorithm

Algorithm

```
If root.data is equal to search.data
   return root
else
   while data not found
      If data is greater than node.data
         goto right subtree
      else
         goto left subtree
      If data found
         return node
   endwhile
   return data not found
end if
```

Search Operation in C++

Link: repl.it/repls/ZestyIndelibleQuotes

```
// function to search a given key in a given BST
node* search(node *curr, int data) { // O(h)
    // Base Cases: root is null or key is present at root
    if (curr == NULL || curr->data == data)
        return curr;
    // curr is greater than curr's data
    if (curr->data < data)
        return search(curr->right, data);
    // data is smaller than curr's data
    else
         return search(curr->left, data);
}
```

Agenda

- ✓ 1- Binary Search Tree Definition
- ✓ 2- Binary Search Tree Representation
- ✓ 3- Basic Operations
- ✓ 4- Insertion Operation
- ✓ 5- Deletion Operation
- ✓ 6- Search Operation
- 7- Balanced Binary Tree Property

Balanced Binary Tree Property

- Height for a Balanced Binary Tree is O(Log n).
- Worst case occurs for skewed tree and worst case height becomes O(n)

So in worst case extra space required is O(n) for both.

- Consider a height-balancing scheme where following conditions

should be checked to determine if a binary tree is balanced.

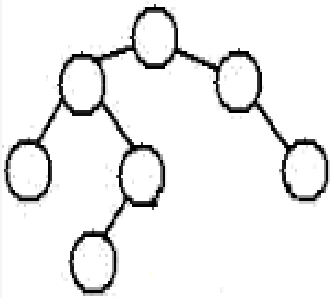
- A non-empty binary tree T is balanced if:

  1) Left sub-tree of T is balanced

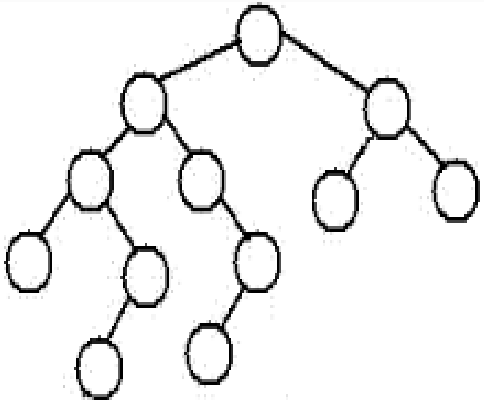
  2) Right sub-tree of T is balanced

  3) The difference between heights of left sub-tree and right sub-tree is not more than 1.

Balanced Binary Tree Property



A height-balanced Tree



Not a height-balanced tree

Agenda

- ✓ 1- Binary Search Tree Definition
- ✓ 2- Binary Search Tree Representation
- ✓ 3- Basic Operations
- ✓ 4- Insertion Operation
- ✓ 5- Deletion Operation
- ✓ 6- Search Operation
- ✓ 7- Balanced Binary Tree Property


DO
MORE.

Practice

Practice

1- Level Order Traversal Line by Line

2- Print all root-to-leaf paths one per line in BST

3- Find maximum value of all nodes in BST

4- Find minimum value of all nodes in BST

5- Check if BST contain value k

6- Calculate height of each node in BST

7- Check if binary search tree is balanced

8- Check if binary tree is BST

9- Find lowest common ancestor LCA in a BST

10- Binary Tree to Binary Search Tree Conversion

Practice

11- Construct BST from its given level order traversal

12- Construct BST from given preorder traversal

13- Construct BST from given inorder traversal

14- Construct BST from given postorder traversal

15- Reverse a path in BST using queue

16- Check if the given array can represent level order traversal of BST

17- Check if the given array can represent preorder traversal of BST

18- Check if the given array can represent inorder traversal of BST

19- Check if two BSTs contain same set of elements

20- Find the largest number in BST which is less than or equal to N

Practice

21- Find median of BST
22- Remove BST keys outside the given range
23- Print BST keys in the given range
24- Count BST nodes that lie in a given range
25- Count BST subtrees that lie in given range
26- Remove all leaf nodes from the BST
27- Sum of k smallest elements in BST
28- Inorder predecessor and successor for a given key in BST by iterative
29- Inorder predecessor and successor for a given key in BST by recursive
30- Maximum element between two nodes of BST

Practice

31- Find pairs with 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

36- Shortest distance between two nodes in BST

37- Count pairs from two BSTs whose sum is equal to a given x

38- Iterative Inorder Traversal

39- Iterative Preorder Traversal

40- Iterative Postorder Traversal


Let's
STARTUP

Practice

1- Level Order Traversal Line by Line

2- Print all root-to-leaf paths one per line in BST

3- Find maximum value of all nodes in BST

4- Find minimum value of all nodes in BST

5- Check if BST contain value k

6- Calculate height of each node in BST

7- Check if binary search tree is balanced

8- Check if binary tree is BST

9- Find lowest common ancestor LCA in a BST

10- Binary Tree to Binary Search Tree Conversion

Level Order Traversal Line by Line

- Implement function which print nodes value of each level in separate line at Binary Search Tree
- Function Name: print level order lines
- Parameters: None
- Return: None

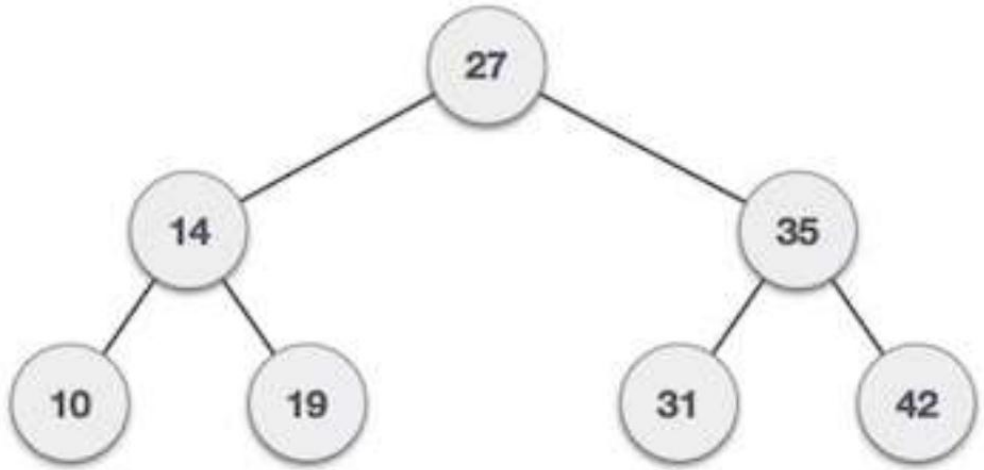
Level Order Traversal Line by Line

This Binary Tree should print:

27

14 35

10 19 31 42



Level Order Traversal Line by Line

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Iterative method to do level order traversal line by line
void printLevelOrderLines() { // O(n)
    // check if tree is empty
    if (root == NULL)
        return;
    // Create an empty queue for level order tarversal and Enqueue Root
    queue<node*> q;
    q.push(root);
    // nodeCount (queue size) indicates number of nodes at current lelvel.
    int nodeCount = q.size();
    while (nodeCount > 0) {
        // Dequeue all nodes of current level and Enqueue all nodes of next level
        while (nodeCount > 0) {
            node *curr = q.front();
            cout << curr->data << ' ';
            q.pop();
            if (curr->left != NULL)
                q.push(curr->left);
            if (curr->right != NULL)
                q.push(curr->right);
            nodeCount--;
        }
        cout << '\n';
        nodeCount = q.size();
    }
}
```

Print all root-to-leaf paths one per line in BST

- Implement function which print nodes values of each path in separate line at Binary Search Tree
- Function Name: print paths recursion
- Parameters: (*curr, path, path size)

  *curr: pointer to current node

  path: array of current path nodes

  path size: size of current path nodes
- Return: None

Print all root-to-leaf paths one per line in BST

Algorithm:

```
initialize: pathlen = 0, path[1000]
/*1000 is some max limit for paths, it can change*/

/*printPathsRecur traverses nodes of tree in preorder */
printPathsRecur(tree, path[], pathlen)
   1) If node is not NULL then
         a) push data to path array:
                path[pathlen] = node->data.
         b) increment pathlen
                pathlen++
   2) If node is a leaf node then print the path array.
   3) Else
        a) Call printPathsRecur for left subtree
                 printPathsRecur(node->left, path, pathLen)
        b) Call printPathsRecur for right subtree.
                printPathsRecur(node->right, path, pathLen)
```

Print all root-to-leaf paths one per line in BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Recursive helper function given a node, and an array containing the path
// from the root node up to but not including this node,
// print out all the root-leaf paths
void print_paths_recursion(node *curr, int path[], int pathLen) { // O(n)
    if (curr == NULL)
        return;
    // append this node to the path array
    path[pathLen] = curr->data;
    pathLen++;
    // it's a leaf, so print the path that led to here
    if (curr->left == NULL && curr->right == NULL) {
        for (int i=0; i<pathLen; i++)
            cout << path[i] << ' ';
        cout << '\n';
    }
    else {
        print_paths_recursion(curr->left, path, pathLen);
        print_paths_recursion(curr->right, path, pathLen);
    }
}
```

Print all root-to-leaf paths one per line in BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Given a binary tree, print out all of its root-to-leaf paths, one per line.
void printPaths() {
    int maxSize = 1e5;
    int path[maxSize];
    print_paths_recursion(root, path, 0);
}
```

Find maximum value of all nodes in BST

- Implement function which get maximum node value in Binary Tree
- Function Name: find max value
- Parameters: (*curr) pointer to current node
- Return: int number which mean maximum node value in Binary Tree

Find maximum value of all nodes in BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Given a non-empty binary search tree,
// return the node with maximum key value found in that tree.
node* maxValueNode(node *curr) { // O(h)
    node *result = curr;
    // loop down to find the rightmost leaf
    while (result->right != NULL)
        result = result->right;
    // the nearest data node of curr data node
    return result;
}
```

Find minimum value of all nodes in BST

- Implement function which get minimum node value in Binary Tree
- Function Name: find min value
- Parameters: (*curr) pointer to current node
- Return: int number which mean minimum node value in Binary Tree

Find minimum value of all nodes in BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Given a non-empty binary search tree,
// return the node with minimum key value found in that tree.
node* minValueNode(node *curr) { // O(h)
    node *result = curr;
    // loop down to find the leftmost leaf
    while (result->left != NULL)
        result = result->left;
    // the nearest data node of curr data node
    return result;
}
```

Check if BST contain value k

- Implement function which check if value k in BST or not
- Function Name: search
- Parameters: (*curr, key)   pointer to current node and key
- Return: boolean, true if BST contain value k, otherwise false

Check if BST contain value k

Link: repl.it/repls/ZestyIndelibleQuotes

```cpp
// function to search a given key in a given BST
node* search(node *curr, int data) { // O(h)
    // Base Cases: root is null or key is present at root
    if (curr == NULL || curr->data == data)
        return curr;
    // curr is greater than curr's data
    if (curr->data < data)
        return search(curr->right, data);
    // data is smaller than curr's data
    else
         return search(curr->left, data);
}
```

Calculate height of each node in BST

- Implement function which calculate height of each node in Binary Search Tree
- Function Name: height
- Parameters: (*curr) pointer to current node
- Return: int number which mean height of Binary Tree

Calculate height of each node in BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// The function Compute the "height" of a tree. Height is the number f nodes along
// the longest path from the root node down to the farthest leaf node.
int height(node *curr) { // O(n)
    // base case tree is empty
    if(curr == NULL)
        return 0;
    // If tree is not empty then height = 1 + max of left height & right heights
    return 1 + max(height(curr->left), height(curr->right));
}
```

Check if binary search tree is balanced

- Implement function which check if Binary Search Tree balanced or not
- Function Name: isBalanced
- Parameters: (*curr) pointer to current node
- Return: boolean true if Binary Search Tree is balanced, false otherwise

Check if binary search tree is balanced

Link: repl.it/repls/ZestyIndelibleQuotes

```cpp
// Returns true if binary tree with root as root is height-balanced
bool isBalanced(node *curr) { // O(n ^ 2)
    // If tree is empty then return true
    if(curr == NULL)
        return true;
    // Get the height of left and right sub trees
    int left_height = height(curr->left);
    int right_height = height(curr->right);

    return  abs(left_height - right_height) <= 1 &&
            isBalanced(curr->left) &&
            isBalanced(curr->right);
}
```

Check if binary tree is BST

- Implement function which check if a binary tree is Binary Search Tree
- Function Name: isBST
- Parameters: (*curr) pointer to current node
- Return: boolean true if a binary tree is Binary Search Tree, false otherwise

Check if binary tree is BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Returns true if the given tree is a BST and its values are >= min and <= max
bool isBST(node *curr, int min, int max) { // O(n)
    // an empty tree is BST
    if (curr==NULL)
        return true;
    // false if this node violates the min/max constraint
    if (curr->data < min || curr->data > max)
        return false;
    // otherwise check the subtrees recursively,
    // tightening the min or max constraint
    return  isBST(curr->left,  min, curr->data-1) &&
            isBST(curr->right, curr->data+1, max);
}
```

Check if binary tree is BST

- Method above runs slowly since it traverses over some parts of the tree many times.

A better solution looks at each node only once. The trick is to a function that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX they narrow from there.

Check if binary tree is BST

```
/* Returns true if the given tree is a binary search tree
 (efficient version). */
int isBST(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
 values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
```

Check if binary tree is BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Returns true if a binary tree is a binary search tree
bool isBST(node *curr) { // O(n * h)
    if (curr == NULL)
        return true;
    // false if the max of the left is > than us
    if (curr->left != NULL && maxValueNode(curr->left)->data > curr->data)
        return false;
    // false if the min of the right is <= than us
    if (curr->right != NULL && minValueNode(curr->right)->data < curr->data)
        return false;
    // false if, recursively, the left or right is not a BST
    if (!isBST(curr->left) || !isBST(curr->right))
        return false;
    // passing all that, it's a BST
    return true;
}
```

Find lowest common ancestor LCA in a BST

- Implement function which get Lowest Common Ancestor of two nodes in Binary Search Tree
- Function Name: find LCA
- Parameters: (*curr, n1, n2)

  *curr: pointer to current node

  n1: first node

  n2: second node
- Return: pointer of node which is lowest common ancestor of n1 and n2

Find lowest common ancestor LCA in a BST

- We can solve this problem using BST properties. We can recursively traverse the BST from root.
- The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2,
- i.e., n1 < n < n2 or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that n1 < n2).
- So just recursively traverse the BST in, if node value is greater than both n1 and n2 then our LCA lies in left side of the node, if it is smaller than both n1 and n2, then LCA lies on right side.
- Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

Find lowest common ancestor LCA in a BST

Link: repl.it/repls/ZestyIndelibleQuotes

```
// Function to find LCA of n1 and n2
// The function assumes that both n1 and n2 are present in BST
node* find_LCA(node* curr, int n1, int n2) { // O(h)
    if (curr == NULL)
        return NULL;
    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (curr->data > n1 && curr->data > n2)
        return find_LCA(curr->left, n1, n2);
    // If both n1 and n2 are greater than root, then LCA lies in right
    if (curr->data < n1 && curr->data < n2)
        return find_LCA(curr->right, n1, n2);
    return curr;
}
```

Binary Tree to Binary Search Tree Conversion

Problem Link: geeksforgeeks.org/binary-tree-to-binary-search-tree-conversion

Practice

1- ~~Level Order Traversal Line by Line~~

2- ~~Print all root-to-leaf paths one per line in BST~~

3- ~~Find maximum value of all nodes in BST~~

4- ~~Find minimum value of all nodes in BST~~

5- ~~Check if BST contain value k~~

6- ~~Calculate height of each node in BST~~

7- ~~Check if binary search tree is balanced~~

8- ~~Check if binary tree is BST~~

9- ~~Find lowest common ancestor LCA in a BST~~

10- ~~Binary Tree to Binary Search Tree Conversion~~


DO
MORE.

Practice

11- Construct BST from its given level order traversal

12- Construct BST from given preorder traversal

13- Construct BST from given inorder traversal

14- Construct BST from given postorder traversal

15- Reverse a path in BST using queue

16- Check if the given array can represent level order traversal of BST

17- Check if the given array can represent preorder traversal of BST

18- Check if the given array can represent inorder traversal of BST

19- Check if two BSTs contain same set of elements

20- Find the largest number in BST which is less than or equal to N

Construct BST from its given level order traversal

Problem Link: geeksforgeeks.org/construct-bst-given-level-order-traversal

Construct BST from given preorder traversal

Problem Link: geeksforgeeks.org/construct-bst-from-given-preorder-traversa

Problem Link: geeksforgeeks.org/construct-bst-from-given-preorder-traversal-set-2

Construct BST from given inorder traversal

Problem Link: geeksforgeeks.org/construct-binary-tree-from-inorder-traversal

Construct BST from given postorder traversal

Problem Link: geeksforgeeks.org/construct-a-binary-search-tree-from-given-postorder

Reverse a path in BST using queue

Problem Link: geeksforgeeks.org/reverse-path-bst-using-queue

Check if the given array can represent level order traversal of BST

Problem Link: geeksforgeeks.org/check-given-array-of-size-n-can-represent-bst-of-n-levels-or-not

Check if the given array can represent preorder traversal of BST

Problem Link: geeksforgeeks.org/check-if-a-given-array-can-represent-preorder-traversal-of-binary-search-tree

Check if the given array can represent inorder traversal of BST

Problem Link: geeksforgeeks.org/check-array-represents-inorder-binary-search-tree-not

Check if two BSTs contain same set of elements

Problem Link: geeksforgeeks.org/check-two-bsts-contain-set-elements

Find the largest number in BST which is less than or equal to N

Problem Link: geeksforgeeks.org/largest-number-bst-less-equal-n

Practice

11- ~~Construct BST from its given level order traversal~~

12- ~~Construct BST from given preorder traversal~~

13- ~~Construct BST from given inorder traversal~~

14- ~~Construct BST from given postorder traversal~~

15- ~~Reverse a path in BST using queue~~

16- ~~Check if the given array can represent level order traversal of BST~~

17- ~~Check if the given array can represent preorder traversal of BST~~

18- ~~Check if the given array can represent inorder traversal of BST~~

19- ~~Check if two BSTs contain same set of elements~~

20- ~~Find the largest number in BST which is less than or equal to N~~


DO
MORE.

Practice

21- Find median of BST

22- Remove BST keys outside the given range

23- Print BST keys in the given range

24- Count BST nodes that lie in a given range

25- Count BST subtrees that lie in given range

26- Remove all leaf nodes from the BST

27- Sum of k smallest elements in BST

28- Inorder predecessor and successor for a given key in BST by iterative

29- Inorder predecessor and successor for a given key in BST by recursive

30- Maximum element between two nodes of BST

Find median of BST

Problem Link: geeksforgeeks.org/find-median-bst-time-o1-space

Remove BST keys outside the given range

Problem Link: geeksforgeeks.org/remove-bst-keys-outside-the-given-range

Print BST keys in the given range

Problem Link: geeksforgeeks.org/print-bst-keys-in-the-given-range

Count BST nodes that lie in a given range

Problem Link: geeksforgeeks.org/count-bst-nodes-that-are-in-a-given-range

Count BST subtrees that lie in given range

Problem Link: geeksforgeeks.org/count-bst-subtrees-that-lie-in-given-range

Remove all leaf nodes from the BST

Problem Link: geeksforgeeks.org/remove-leaf-nodes-binary-search-tree

Sum of k smallest elements in BST

Problem Link: geeksforgeeks.org/sum-k-smallest-elements-bst

Inorder predecessor and successor for a given key in BST by iterative

Problem Link: geeksforgeeks.org/inorder-predecessor-and-successor-for-a-given-key-in-bst-iterative-approach

Inorder predecessor and successor for a given key in BST by recursive

Problem Link: geeksforgeeks.org/inorder-predecessor-successor-given-key-bst

Maximum element between two nodes of BST

Problem Link: geeksforgeeks.org/maximum-element-two-nodes-bst

Practice

21- ~~Find median of BST~~

22- ~~Remove BST keys outside the given range~~

23- ~~Print BST keys in the given range~~

24- ~~Count BST nodes that lie in a given range~~

25- ~~Count BST subtrees that lie in given range~~

26- ~~Remove all leaf nodes from the BST~~

27- ~~Sum of k smallest elements in BST~~

28- ~~Inorder predecessor and successor for a given key in BST by iterative~~

29- ~~Inorder predecessor and successor for a given key in BST by recursive~~

30- ~~Maximum element between two nodes of BST~~


DO
MORE.

Practice

31- Find pairs with 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

36- Shortest distance between two nodes in BST

37- Count pairs from two BSTs whose sum is equal to a given x

38- Iterative Inorder Traversal

39- Iterative Preorder Traversal

40- Iterative Postorder Traversal

Find pairs with given sum such that pair elements lie in different BSTs

Problem Link: geeksforgeeks.org/find-pairs-with-given-sum-such-that-pair-elements-lie-in-different-bsts

Find the largest BST subtree in a given binary tree

Problem Link: geeksforgeeks.org/find-the-largest-subtree-in-a-tree-that-is-also-a-bst

Replace every element with the least greater element on its right

Problem Link: geeksforgeeks.org/replace-every-element-with-the-least-greater-element-on-its-right

Add all greater values to every node in a given BST

Problem Link: geeksforgeeks.org/add-greater-values-every-node-given-bst

Check for identical BSTs without building the trees

Problem Link: geeksforgeeks.org/check-for-identical-bsts-without-building-the-trees

Shortest distance between two nodes in BST

Problem Link: geeksforgeeks.org/shortest-distance-between-two-nodes-in-bst

Count pairs from two BSTs whose sum is equal to a given x

Problem Link: geeksforgeeks.org/count-pairs-from-two-bsts-whose-sum-is-equal-to-a-given-value-x

Iterative Inorder Traversal

Problem Link: geeksforgeeks.org/inorder-tree-traversal-without-recursion

Iterative Preorder Traversal

Problem Link: geeksforgeeks.org/iterative-preorder-traversal

Iterative Postorder Traversal

Problem Link: geeksforgeeks.org/iterative-postorder-traversal-using-stack

Practice

~~31- Find pairs with 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~~

~~36- Shortest distance between two nodes in BST~~

~~37- Count pairs from two BSTs whose sum is equal to a given x~~

~~38- Iterative Inorder Traversal~~

~~39- Iterative Preorder Traversal~~

~~40- Iterative Postorder Traversal~~


DO
MORE.

Assignment

References

[01]	Online Course YouTube Playlists	https://bit.ly/2Pq88rN
[02]	Introduction to Algorithms Thomas H. Cormen	https://bit.ly/2ONhuSn
[03]	Competitive Programming 3 Steven Halim	https://nus.edu/2z40vyK
[04]	Fundamental of Algorithmics Gilles Brassard and Paul Bartley	https://bit.ly/2QuvwbM
[05]	Analysis of Algorithms An Active Learning Approach	http://bit.ly/2EgCCYX
[06]	Data Structures and Algorithms Annotated Reference	http://bit.ly/2c37XEv
[07]	Competitive Programmer's Handbook	https://bit.ly/2APAbeG
[08]	GeeksforGeeks	https://geeksforgeeks.org
[09]	Codeforces Online Judge	http://codeforces.com
[10]	HackerEarth Online Judge	https://hackerearth.com
[11]	TopCoder Online Judge	https://topcoder.com

Questions ?