**Assignment Binary Search Tree**

Q1. Write an iterative program to search for an element in BST. Also construct a sample BST and try to search for elements in the same.

The for BST is : 15, 10, 20, 8, 12, 16, 25

Search for 25 in it. Left to right nodes arranged

The given key is the right node of the node with key 20

class Node {

int key;

Node left, right;

public Node(int item) {

key = item;

left = right = null;

}

}

class BinarySearchTree {

Node root;

// Constructor

BinarySearchTree() {

root = null;

}

// Insert a new node into the BST

void insert(int key) {

root = insertRec(root, key);

}

// A recursive function to insert a new key in BST

Node insertRec(Node root, int key) {

// If the tree is empty, return a new node

if (root == null) {

root = new Node(key);

return root;

}

// Otherwise, recur down the tree

if (key < root.key)

root.left = insertRec(root.left, key);

else if (key > root.key)

root.right = insertRec(root.right, key);

// return the (unchanged) node pointer

return root;

}

// Iterative function to search a key in BST

boolean search(int key) {

Node current = root;

while (current != null) {

if (current.key == key) {

return true; // Key found

} else if (key < current.key) {

current = current.left; // Go to the left subtree

} else {

current = current.right; // Go to the right subtree

}

}

return false; // Key not found

}

// Function to print inorder traversal of the BST (for visualization)

void inorder() {

inorderRec(root);

System.out.println();

}

// A utility function to do inorder traversal of BST

void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.print(root.key + " ");

inorderRec(root.right);

}

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

// Insert elements into the BST

int[] elements = {15, 10, 20, 8, 12, 16, 25};

for (int elem : elements) {

bst.insert(elem);

}

// Print the inorder traversal of the BST (left to right)

System.out.println("Inorder traversal of the BST:");

bst.inorder();

// Search for key 25 in the BST

int keyToSearch = 25;

if (bst.search(keyToSearch)) {

System.out.println("Key " + keyToSearch + " is found in the BST.");

} else {

System.out.println("Key " + keyToSearch + " is NOT found in the BST.");

}

}

}

Q2. Given a BST and a positive number k, find the k'th largest node in the BST.

The for BST is : 15, 10, 20, 8, 12, 16, 25

For example, consider the following binary search tree. If k = 2, the k'th largest node is 20.

class Node {

int key;

Node left, right;

public Node(int item) {

key = item;

left = right = null;

}

}

class BinarySearchTree {

Node root;

int count = 0;

Node kthLargestNode = null;

// Constructor

BinarySearchTree() {

root = null;

}

// Insert a new node into the BST

void insert(int key) {

root = insertRec(root, key);

}

// A recursive function to insert a new key in BST

Node insertRec(Node root, int key) {

if (root == null) {

root = new Node(key);

return root;

}

if (key < root.key)

root.left = insertRec(root.left, key);

else if (key > root.key)

root.right = insertRec(root.right, key);

return root;

}

// Function to find the k'th largest element

void findKthLargest(int k) {

findKthLargestRec(root, k);

if (kthLargestNode != null) {

System.out.println("The " + k + "'th largest node is " + kthLargestNode.key);

} else {

System.out.println("There are less than " + k + " nodes in the BST.");

}

}

// Recursive function to perform reverse inorder traversal

void findKthLargestRec(Node node, int k) {

if (node == null || count >= k) {

return;

}

// First, recurse on the right subtree (larger elements)

findKthLargestRec(node.right, k);

// Increment the count of visited nodes

count++;

// If count equals k, we've found the k'th largest element

if (count == k) {

kthLargestNode = node;

return;

}

// Otherwise, recurse on the left subtree (smaller elements)

findKthLargestRec(node.left, k);

}

// Function to print inorder traversal of the BST (for visualization)

void inorder() {

inorderRec(root);

System.out.println();

}

// A utility function to do inorder traversal of BST

void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.print(root.key + " ");

inorderRec(root.right);

}

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

// Insert elements into the BST

int[] elements = {15, 10, 20, 8, 12, 16, 25};

for (int elem : elements) {

bst.insert(elem)

}

// Print the inorder traversal of the BST

System.out.println("Inorder traversal of the BST:");

bst.inorder();

// Find the k'th largest element

int k = 2;

bst.findKthLargest(k); // Output should be 20

}

}

Q3. Given a binary search tree, find a pair with a given sum present in it.

BST 15, 10, 20, 8, 12, 20,30,6,9,18,22

For example, consider the following BST. If the given sum is 14, the pair is (8, 6).

import java.util.ArrayList;

import java.util.List;

class Node {

int key;

Node left, right;

public Node(int item) {

key = item;

left = right = null;

}

}

class BinarySearchTree {

Node root;

BinarySearchTree() {

root = null;

}

// Insert a new node into the BST

void insert(int key) {

root = insertRec(root, key);

}

// A recursive function to insert a new key in BST

Node insertRec(Node root, int key) {

if (root == null) {

root = new Node(key);

return root;

}

if (key < root.key)

root.left = insertRec(root.left, key);

else if (key > root.key)

root.right = insertRec(root.right, key);

return root;

}

// Inorder traversal to get elements in sorted order

void inorder(Node node, List<Integer> list) {

if (node != null) {

inorder(node.left, list);

list.add(node.key);

inorder(node.right, list);

}

}

// Function to find if there is a pair with a given sum

boolean findPairWithSum(int sum) {

// List to store inorder traversal (sorted elements)

List<Integer> inorderList = new ArrayList<>();

// Perform inorder traversal

inorder(root, inorderList);

// Use two-pointer technique to find the pair

int left = 0, right = inorderList.size() - 1;

while (left < right) {

int currentSum = inorderList.get(left) + inorderList.get(right);

if (currentSum == sum) {

System.out.println("Pair found: (" + inorderList.get(left) + ", " + inorderList.get(right) + ")");

return true;

}

if (currentSum < sum)

left++; // Move the left pointer to the right

else

right--; // Move the right pointer to the left

}

// If no pair is found

System.out.println("No pair found with the given sum.");

return false;

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

// Insert elements into the BST

int[] elements = {15, 10, 20, 8, 12, 20, 30, 6, 9, 18, 22};

for (int elem : elements) {

bst.insert(elem);

}

// Given sum

int sum = 14;

// Find if there is a pair with the given sum

bst.findPairWithSum(sum);

}

}

Q4. Given a BST, find the inorder predecessor of a given key in it. If the key does not lie in the BST, return the previous greater node (if any) present in the BST.

The for BST is : 15, 10, 20, 8, 12, 16, 25

The predecessor of node 15 is 12

The predecessor of node 10 is 8

The predecessor of node 20 is 16

The predecessor doesn’t exist for node 8

The predecessor of node 12 is 10

The predecessor of node 16 is 15

The predecessor of node 25 is 20

A node’s inorder predecessor is a node with maximum value in its left subtree, i.e., its left

subtree’s right-most child. If the left subtree of the node doesn’t exist, then the inorder

predecessor is one of its ancestors

class Node {

int key;

Node left, right;

public Node(int item) {

key = item;

left = right = null;

}

}

class BinarySearchTree {

Node root;

BinarySearchTree() {

root = null;

}

// Insert a new node into the BST

void insert(int key) {

root = insertRec(root, key);

}

// A recursive function to insert a new key in BST

Node insertRec(Node root, int key) {

if (root == null) {

root = new Node(key);

return root;

}

if (key < root.key)

root.left = insertRec(root.left, key);

else if (key > root.key)

root.right = insertRec(root.right, key);

return root;

}

// Function to find the inorder predecessor of a given key

Node findInorderPredecessor(Node root, int key) {

Node predecessor = null;

Node current = root;

while (current != null) {

if (key > current.key) {

// If key is greater, the current node could be a potential predecessor

predecessor = current;

current = current.right;

} else {

// If key is smaller or equal, move to the left

current = current.left;

}

}

return predecessor;

}

// Function to get the maximum node in a given subtree

Node getMaxNode(Node node) {

while (node.right != null) {

node = node.right;

}

return node;

}

// Function to find the inorder predecessor of a given key

Node getInorderPredecessor(int key) {

Node current = root;

Node predecessor = null;

while (current != null) {

if (key > current.key) {

// Potential predecessor found, move right

predecessor = current;

current = current.right;

} else if (key < current.key) {

// Move left

current = current.left;

} else {

// If the node has a left subtree, the predecessor is the rightmost node of the left subtree

if (current.left != null) {

predecessor = getMaxNode(current.left);

}

break;

}

}

return predecessor;

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

// Insert elements into the BST

int[] elements = {15, 10, 20, 8, 12, 16, 25};

for (int elem : elements) {

bst.insert(elem);

}

// Test cases

int[] keys = {15, 10, 20, 8, 12, 16, 25};

for (int key : keys) {

Node predecessor = bst.getInorderPredecessor(key);

if (predecessor != null) {

System.out.println("Inorder predecessor of " + key + " is " + predecessor.key);

} else {

System.out.println("Inorder predecessor of " + key + " doesn't exist.");

}

}

}

}

Q5. Given a BST and two nodes x and y in it, find the lowest common ancestor (LCA) of x and y.

The solution should return null if either x or y is not the actual node in the tree.

LCA (6, 12) = 10

LCA (10, 12) = 10

LCA (20, 6) = 15

LCA (18, 22)= 20

LCA (30, 30) = 30

BST 15, 10, 20, 8, 12, 20,30,6,9,18,22

class Node {

int key;

Node left, right;

public Node(int item) {

key = item;

left = right = null;

}

}

class BinarySearchTree {

Node root;

BinarySearchTree() {

root = null;

}

// Insert a new node into the BST

void insert(int key) {

root = insertRec(root, key);

}

// A recursive function to insert a new key in BST

Node insertRec(Node root, int key) {

if (root == null) {

root = new Node(key);

return root;

}

if (key < root.key)

root.left = insertRec(root.left, key);

else if (key > root.key)

root.right = insertRec(root.right, key);

return root;

}

// Function to find the LCA of two nodes

Node findLCA(int x, int y) {

// Check if both nodes are present in the BST

if (!findNode(root, x) || !findNode(root, y)) {

return null; // One or both nodes are not present in the tree

}

return findLCARec(root, x, y);

}

// Function to find the LCA of two nodes recursively

Node findLCARec(Node root, int x, int y) {

if (root == null) {

return null;

}

if (x < root.key && y < root.key) {

// Both nodes are in the left subtree

return findLCARec(root.left, x, y);

}

if (x > root.key && y > root.key) {

// Both nodes are in the right subtree

return findLCARec(root.right, x, y);

}

// One node is on the left, and the other is on the right, or one of the nodes is the root

return root;

}

// Function to check if a node with the given key is present in the BST

boolean findNode(Node root, int key) {

if (root == null) {

return false;

}

if (root.key == key) {

return true;

}

return key < root.key ? findNode(root.left, key) : findNode(root.right, key);

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

// Insert elements into the BST

int[] elements = {15, 10, 20, 8, 12, 20, 30, 6, 9, 18, 22};

for (int elem : elements) {

bst.insert(elem);

}

// Test cases

int[][] queries = {

{6, 12}, {10, 12}, {20, 6}, {18, 22}, {30, 30}

};

for (int[] query : queries) {

int x = query[0];

int y = query[1];

Node lca = bst.findLCA(x, y);

if (lca != null) {

System.out.println("LCA of " + x + " and " + y + " is " + lca.key);

} else {

System.out.println("One or both nodes are not present in the tree.");

}

}

}

}