**Assignment No. 01**

Name:- Anirudha Shripad Kale.

Roll No. :- 223029

PRN No :- 22210288

Batch: C1

**Problem Statement:** Implement binary search tree and perform following operations:

1. Insert
2. Delete
3. Display

**Objectives:**

1. To understand the concept of Binary Trees.
2. To implement binary search tree using another data structure.
3. To understand the different operations on binary search tree.

**Theory:**

Binary search tree: A Binary Search Tree (BST) is a type of binary tree in which every node has at most two children, referred to as the left child and the right child. The BST has the property that for each node:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtrees must also be binary search trees.

Operations:

Insert:

1. To insert a new node into a BST, compare the key of the new node with the key of the root node.
2. If the new node's key is less than the root's key, then recursively insert the new node in the left subtree.
3. If the new node's key is greater than the root's key, then recursively insert the new node in the right subtree.
4. Repeat this process until a suitable empty position is found, and insert the new node there.

Delete:

Deleting a node from a BST involves three cases:

1. If the node to be deleted has no children, simply remove the node from the tree.
2. If the node has only one child, replace the node with its child.
3. If the node has two children, find the node with the minimum key in its right subtree (or the maximum key in its left subtree), replace the node's key with that value, and then recursively delete that node in the right (or left) subtree.

Display:

There are several ways to display or traverse a binary search tree:

1. Inorder Traversal: Visit the left subtree, then the root, then the right subtree. This results in keys being visited in sorted order.
2. Preorder Traversal: Visit the root, then the left subtree, then the right subtree.
3. Postorder Traversal: Visit the left subtree, then the right subtree, then the root.

Insertion, deletion, and searching in a balanced BST take O(log n) time on average, where n is the number of nodes in the tree.

However, in the worst case (unbalanced tree), these operations can take O(n) time, where n is the number of nodes.

**Algorithm:**

1. Start.
2. Initialize root to null.
3. Take the input from user and call the respective function.
4. Implement the given function according to above mention algorithms.
5. When the user choice is exit then stop the process and return out of loop.
6. Stop.

**Code:**

import java.util.\*;

class Node {

    int data;

    Node left;

    Node right;

    Node(int data) {

        this.data = data;

        this.left = null;

        this.right = null;

    }

}

class Assignment1 {

    // Node root = null;

    public Node insertNode(Node root, int data) {

        Node new\_node = new Node(data);

        // base case

        if (root == null) {

            root = new\_node;

            return root;

        }

        if (data < root.data) {

            root.left = insertNode(root.left, data);

        }

        else if (data >= root.data) {

            root.right = insertNode(root.right, data);

        }

        return root;

    }

    public Node deleteNode(Node root, int value) {

        if (root == null) {

            System.out.println("The tree is empty.");

            return null;

        }

        if (root.data > value)

            root.left = deleteNode(root.left, value);

        else if (root.data < value)

            root.right = deleteNode(root.right, value);

        else {

            // Node to be deteted has both the child

            if (root.left != null && root.right != null) {

                Node temp = root;

                Node minNodeRight = minimum(temp.right);

                root.data = minNodeRight.data;

                root.right = deleteNode(root.right, minNodeRight.data);

            }

            // if the node has only left child

            else if (root.left != null)

                root = root.left;

            // if the node has only right child

            else if (root.right != null)

                root = root.right;

            // has no child

            else

                root = null;

        }

        return root;

    }

    public Node minimum(Node root) {

        if (root.left == null)

            return root;

        return minimum(root.left);

    }

    public Node searchNode(Node root, int data) {

        if (root == null || root.data == data)

            return root;

        if (data >= root.data)

            return searchNode(root.right, data);

        return searchNode(root.left, data);

    }

    public void Inorder(Node root) {

        if (root != null) {

            Inorder(root.left);

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

            Inorder(root.right);

        }

    }

    public void levelOrderPrint(Node root) {

        if (root == null) {

            return;

        }

        Queue<Node> queue = new LinkedList<>();

        queue.add(root);

        queue.add(null);

        while (!queue.isEmpty()) {

            Node temp = queue.poll();

            if (temp == null) {

                System.out.println();

                if (!queue.isEmpty()) {

                    queue.add(null);

                }

            } else {

                System.out.print(temp.data + " ");

                if (temp.left != null)

                    queue.add(temp.left);

                if (temp.right != null)

                    queue.add(temp.right);

            }

        }

    }

    public void findPredecessorAndSuccessor(Node root, int key) {

        int predecessor = -1;

        int successor = -1;

        Node temp = root;

        while (temp != null && temp.data != key) {

            if (key > temp.data) {

                predecessor = temp.data;

                temp = temp.right;

            } else { // (key < temp.data)

                successor = temp.data;

                temp = temp.left;

            }

        }

        // Key not found

        if (temp == null) {

            System.out.println("The node with key " + key + " doesn't present in the tree.");

            System.out.println("Inorder predecessor and successor cannot be determined.");

            return;

        }

        // now the key is found

        // find predecessor which is present in left subtree

        // take one left and then the rightmost element

        Node leftTree = temp.left;

        while (leftTree != null) {

            predecessor = leftTree.data;

            leftTree = leftTree.right;

        }

        // find successor which is present in right subtree

        // take one right and then the leftmost element

        Node rightTree = temp.right;

        while (rightTree != null) {

            successor = rightTree.data;

            rightTree = rightTree.left;

        }

        System.out.println("The inorder predecessor of " + key + " is : " + predecessor);

        System.out.println("The inorder successor of " + key + " is : " + successor);

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        Node root = null;

        Assignment1 tree1 = new Assignment1();

        System.out.println("Welcome user !! \nThis is the system which performs Binary search Tree Opeartions : ");

        System.out.println("-----------------------------------------");

        boolean x = true;

        while (x) {

            System.out.println("Operations Available  : ");

            System.out.println(

                    "1. Insert the Node \n2. Delete the Node \n3. Search the Node \n4. Inorder print \n5. Level order print \n6. Find the inorder predecessor and sucessor of the node you want \n7.Exit");

            System.out.println("-----------------------------------------");

            System.out.print("Enter your choice : ");

            int choice = sc.nextInt();

            switch (choice) {

                case 1:

                    System.out.println("Enter the data to create the BST : ");

                    int data = sc.nextInt();

                    while (data != -1) {

                        root = tree1.insertNode(root, data);

                        data = sc.nextInt();

                    }

                    System.out.println("The data get successfully inserted.");

                    System.out.println("\n-----------------------------------------");

                    break;

                case 2:

                    System.out.println("Enter the element that you want to delete : ");

                    int element = sc.nextInt();

                    root = tree1.deleteNode(root, element);

                    System.out.println("The element " + element + " get successfully deleted.");

                    System.out.println("\n-----------------------------------------");

                    break;

                case 3:

                    System.out.print("Enter the element that you want to search : ");

                    int search = sc.nextInt();

                    if (tree1.searchNode(root, search) == null)

                        System.out.println(search + " is not fount in BST.");

                    else

                        System.out.println(search + " is present in the BST.");

                    System.out.println("\n-----------------------------------------");

                    break;

                case 4:

                    System.out.println("The Inorder representation of BST is : ");

                    tree1.Inorder(root);

                    System.out.println("\n-----------------------------------------");

                    break;

                case 5:

                    System.out.println("The level order representation of BST is : ");

                    tree1.levelOrderPrint(root);

                    System.out.println("\n-----------------------------------------");

                    break;

                case 6:

                    System.out.print("Enter the node whose predecessor and successor you want to find : ");

                    int key = sc.nextInt();

                    tree1.findPredecessorAndSuccessor(root, key);

                    System.out.println("\n-----------------------------------------");

                    break;

                case 7:

                    x = false;

                    break;

            }

        }

    }

}

**Output:**

Welcome user !!

This is the system which performs Binary search Tree Opeartions :

-----------------------------------------

Operations Available :

1. Insert the Node

2. Delete the Node

3. Search the Node

4. Inorder print

5. Level order print

6. Find the inorder predecessor and sucessor of the node you want

7.Exit

-----------------------------------------

Enter your choice : 1

Enter the data to create the BST :

23 56 47 8 96 54 12 32 41 -1

The data get successfully inserted.

-----------------------------------------

Operations Available :

1. Insert the Node

2. Delete the Node

3. Search the Node

4. Inorder print

5. Level order print

6. Find the inorder predecessor and sucessor of the node you want

7.Exit

-----------------------------------------

Enter your choice : 4

The Inorder representation of BST is :

8 12 23 32 41 47 54 56 96

-----------------------------------------

Operations Available :

1. Insert the Node

2. Delete the Node

3. Search the Node

4. Inorder print

5. Level order print

6. Find the inorder predecessor and sucessor of the node you want

7.Exit

-----------------------------------------

Enter your choice : 5

The level order representation of BST is :

23

8 56

12 47 96

32 54

41

-----------------------------------------

Operations Available :

1. Insert the Node

2. Delete the Node

3. Search the Node

4. Inorder print

5. Level order print

6. Find the inorder predecessor and sucessor of the node you want

7.Exit

-----------------------------------------

Enter your choice : 2

Enter the element that you want to delete :

23

The element 23 get successfully deleted.

-----------------------------------------

Operations Available :

1. Insert the Node

2. Delete the Node

3. Search the Node

4. Inorder print

5. Level order print

6. Find the inorder predecessor and sucessor of the node you want

7.Exit

-----------------------------------------

Enter your choice : 5

The level order representation of BST is :

32

8 56

12 47 96

41 54

-----------------------------------------

Operations Available :

1. Insert the Node

2. Delete the Node

3. Search the Node

4. Inorder print

5. Level order print

6. Find the inorder predecessor and sucessor of the node you want

7.Exit

-----------------------------------------

Enter your choice : 6

Enter the node whose predecessor and successor you want to find : 47

The inorder predecessor of 47 is : 41

The inorder successor of 47 is : 54

-----------------------------------------

Operations Available :

1. Insert the Node

2. Delete the Node

3. Search the Node

4. Inorder print

5. Level order print

6. Find the inorder predecessor and sucessor of the node you want

7.Exit

-----------------------------------------

Enter your choice : 7