PW SKILLS

Lesson Plan

BST-2

Java

PW SKILLS

Today's Checklist

- Inorder predecessor and successor for a given key in BST
- Deletion in BST

Q. Inorder predecessor and successor for a given key in BST

Input: The input to the program is the key for which we want to find the inorder predecessor and successor in the BST.

Output: The output of the program is the inorder predecessor and successor of the given key in the BST.

Key = 60

```
Inorder predecessor is 50
Inorder successor is 70
```

Explanation: In a Binary Search Tree (BST), the inorder predecessor of a node is the node with the largest value smaller than the given node, and the inorder successor of a node is the node with the smallest value larger than the given node.

To find the inorder predecessor and successor of a given key in a BST, we can start by traversing the BST from the root node. If the key is found in the BST, we can find its predecessor and successor as follows:

1. If the left subtree of the key node is not empty, the predecessor of the key node is the rightmost node in the left subtree (i.e., the node with the largest value smaller than the key).
2. If the right subtree of the key node is not empty, the successor of the key node is the leftmost node in the right subtree (i.e., the node with the smallest value larger than the key).

If the key is not found in the BST, we can still find its predecessor and successor as follows:

1. If the key is smaller than the root node, we move to the left subtree of the root node and update the successor to be the root node. We repeat this process until we find the last node with a value smaller than the key, which is the inorder predecessor.
2. If the key is larger than the root node, we move to the right subtree of the root node and update the predecessor to be the root node. We repeat this process until we find the first node with a value larger than the key, which is the inorder successor.

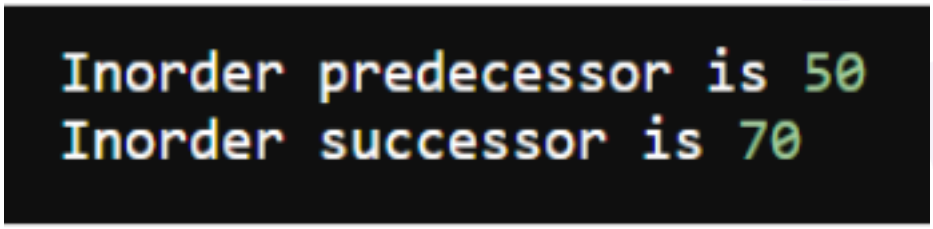
Overall, to find the inorder predecessor and successor of a given key in a BST, we need to traverse the tree once

Code:

```
class Node {
    int data;
    Node left;
    Node right;

    Node(int val) {
        data = val;
        left = right = null;
    }
}

public class Main {
    public static void findPreSuc(Node root, int key, Node[]
preSuc) {
        if (root == null) // If the root is null, return
            return;

        if (root.data == key) { // If the key is found, check for
its predecessor and successor
            if (root.left != null) {
                Node pre = root.left;
                while (pre.right != null)
                    pre = pre.right;
                preSuc[0] = pre;
            }
            if (root.right != null) {
                Node suc = root.right;
                while (suc.left != null)
                    suc = suc.left;
                preSuc[1] = suc;
            }
            return;
        }

        if (root.data > key) { // If the key is smaller, move to
the left subtree
            preSuc[1] = root;
            findPreSuc(root.left, key, preSuc);
        } else { // If the key is greater, move to the right
subtree
            preSuc[0] = root;
            findPreSuc(root.right, key, preSuc);
        }
    }

    public static void main(String[] args) {
        Node root = new Node(50);
        root.left = new Node(30);
        root.right = new Node(70);
        root.left.left = new Node(20);
```

```
        root.left.right = new Node(40);
        root.right.left = new Node(60);
        root.right.right = new Node(80);

        int key = 60;
        Node[] preSuc = new Node[2];
        preSuc[0] = null;
        preSuc[1] = null;
        findPreSuc(root, key, preSuc);

        System.out.println("Inorder predecessor is " +
((preSuc[0] ≠ null) ? preSuc[0].data : -1));
        System.out.println("Inorder successor is " + ((preSuc[1]
≠ null) ? preSuc[1].data : -1));
    }
}
```

Output:



Time Complexity: The time complexity of this program is O(h), where h is the height of the BST. In the worst case, when the BST is skewed (i.e., has only one child for each node), the time complexity is O(n), where n is the number of nodes in the BST.

Space Complexity: The space complexity of this program is O(1), since it uses only a constant amount of extra space.

Deletion

Approach:
Deletion in a Binary Search Tree (BST) involves removing a node from the tree while maintaining the properties of the BST. Here are the steps for deleting a node from a BST:

1. Find the node to be deleted: The first step in deleting a node from the BST is to find the node that needs to be deleted. We start at the root node and traverse the tree until we find the node that matches the value we want to delete.
2. Determine the type of node to be deleted: Once we have found the node to be deleted, we need to determine what type of node it is. There are three types of nodes we need to consider:
    - Leaf node: A node with no children.
    - Node with one child: A node with only one child.
    - Node with two children: A node with two children.
3. Delete the leaf node: If the node to be deleted is a leaf node, we can simply remove it from the tree.

4. Delete a node with one child: If the node to be deleted has only one child, we can replace the node with its child. We simply connect the parent of the node to be deleted with its child node.

5. Delete a node with two children: If the node to be deleted has two children, we need to find the node with the next highest value in the tree. This node is called the successor node. We can replace the node to be deleted with the successor node and then delete the successor node using steps 3 or 4 above.

6. Update the tree: After deleting the node, we need to update the tree to maintain the BST properties. We need to ensure that all nodes to the left of a node have a value less than the node, and all nodes to the right of a node have a value greater than the node.

7. Repeat if necessary: If we have deleted a node with children, we need to repeat the process for those children until we have removed all the nodes that need to be deleted.

Overall, the deletion process in a BST can be complex, depending on the structure of the tree and the type of node to be deleted. However, by following the steps above, we can ensure that the tree remains a valid BST after the deletion.

Code:

```
class Node {
    int data;
    Node left;
    Node right;

    Node(int val) {
        data = val;
        left = right = null;
    }
}

public class Main {
    // Function to create a new node
    public static Node createNode(int val) {
        return new Node(val);
    }

    // Function to insert a value into the BST
    public static Node insert(Node root, int val) {
        if (root == null)
            return createNode(val);
        if (val < root.data)
            root.left = insert(root.left, val);
        else if (val > root.data)
            root.right = insert(root.right, val);
        return root;
    }

    // Function to find the minimum value in a BST
```

```
    public static Node minValueNode(Node node) {
        Node current = node;
        while (current ≠ null && current.left ≠ null)
            current = current.left;
        return current;
    }

    // Function to delete a value from the BST
    public static Node deleteNode(Node root, int key) {
        if (root == null)
            return root;
        if (key < root.data)
            root.left = deleteNode(root.left, key);
        else if (key > root.data)
            root.right = deleteNode(root.right, key);
        else {
            if (root.left == null) {
                Node temp = root.right;
                return temp;
            } else if (root.right == null) {
                Node temp = root.left;
                return temp;
            }
            Node temp = minValueNode(root.right);
            root.data = temp.data;
            root.right = deleteNode(root.right, temp.data);
        }
        return root;
    }

    // Function to print the preorder traversal of the BST
    public static void preorder(Node root) {
        if (root ≠ null) {
            System.out.print(root.data + " ");
            preorder(root.left);
            preorder(root.right);
        }
    }

    // Main function to test the program
    public static void main(String[] args) {
        Node root = null;
        int n, val;
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of nodes in the BST:
");
        n = scanner.nextInt();
        System.out.print("Enter the values of the nodes: ");
        for (int i = 0; i < n; i++) {
            val = scanner.nextInt();
            root = insert(root, val);
```

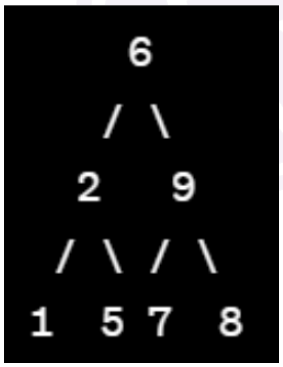
```
        }
        System.out.println("\nPreorder traversal of the BST: ");
        preorder(root);
        System.out.println();
        System.out.print("Enter the value to be deleted: ");
        val = scanner.nextInt();
        root = deleteNode(root, val);
        System.out.println("Preorder traversal of the modified
BST: ");
        preorder(root);
        System.out.println();
        scanner.close();
    }
}
```

Output:

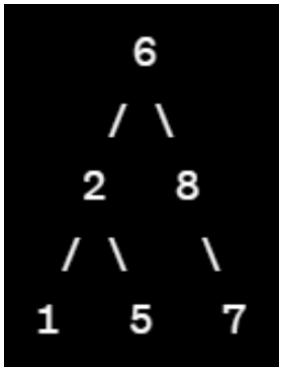
```
Enter the number of nodes in the BST: 7
Enter the values of the nodes: 6 2 9 8 5 1 7

Preorder traversal of the BST: 6 2 1 5 9 8 7
Enter the value to be deleted: 9
Preorder traversal of the modified BST: 6 2 1 5 8 7
```

BST Diagram:

Before deletion



After deletion



Time Complexity:

- Insertion and deletion operations in a BST have a time complexity of O(log n) in the average case, where n is the number of nodes in the BST.
- However, in the worst case, the time complexity can be O(n) if the BST is degenerate (e.g. all nodes have only one child) and resembles a linked list.

Space Complexity: The space complexity of the deleteHelper() function depends on the height of the BST. In the worst case, it is O(n) i.e. when the BST is skewed and in the average case, it is O(log n).

PW SKILLS

THANK YOU !

PW SKILLS