Binary Search Tree (BST) Operations

A **Binary Search Tree (BST)** is a binary tree where each node has a value, and:

1. The left child of a node contains values **less than** the node's value.
2. The right child of a node contains values **greater than** the node's value.

1. Insertion Operation in BST

To insert a new value in a BST, the algorithm compares the new value with the values in the existing nodes and follows the properties of BST to place the new value in the correct position.

Steps:

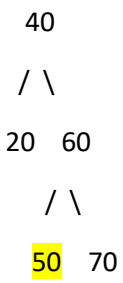
- Start at the root node.
- Compare the new value val with the current node's value:
    - If val is **less than** the current node, move to the left child.
    - If val is **greater than** the current node, move to the right child.
    - Repeat this until you find an empty spot (left or right child is null).
- Insert the new value at that spot.

Example: Insert 50 into the BST:

50 > 40, move right.

50 < 60, move left.

Insert 50 as the left child of 60.



2. Deletion Operation in BST

Deletion in a BST can be done in three cases:

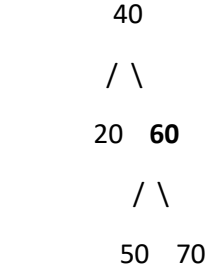
- **Case 1**: Node to be deleted has **no children** (leaf node).
- **Case 2**: Node to be deleted has **one child**.

- **Case 3**: Node to be deleted has **two children**.

Steps:

- Start at the root and search for the node to delete.
- Depending on the case:
    - **Case 1**: Simply remove the node.
    - **Case 2**: Replace the node with its only child.
    - **Case 3**: Find the **in-order successor** (smallest node in the right subtree) and replace the node to be deleted with the in-order successor.

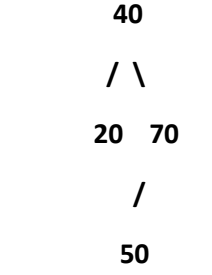
Example: **Delete 60 from the BST:**



60 has two children, so find the in-order successor (70).

Replace 60 with 70.

The final tree after deletion:



3. Searching Operation in BST

To search for a value in a BST, follow the same logic as insertion by comparing the value at each node.

Steps:

- Start at the root node.
- Compare the search value val with the current node:
    - If val is equal to the current node, the value is found.

    - If val is less than the current node, move to the left child.
    - If val is greater than the current node, move to the right child.
- Repeat until the value is found or the subtree is null (meaning the value is not in the tree).

Example: **Search for 50 in the BST:**

50 > 40, move right.

50 < 60, move left.

50 is found.

```
      40
     / \
   20   60
       / \
      50  70
```

4. Modification of Data in BST

To modify data in a BST, you need to first search for the node containing the data and then update its value.

Steps:

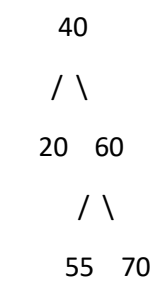
- Search for the node where the value needs to be modified using the search operation.
- Once found, update the node's value to the new data. After updating, check if the BST properties still hold. If the new value violates the BST property, you may need to delete the node and re-insert it with the new value.

Example: **Modify 50 to 55 in the BST:**

```
      40
     / \
   20   60
       / \
      50  70
```

- Search for 50.
- Modify 50 to 55.
- Check the tree's structure to ensure the BST properties are maintained.

New Tree:



Java Code for the Operation of Insertion, Deletion, Searching Modification of data in Binary Search Tree (BST).

```
import java.util.Scanner;

class Node {
   int data;
   Node left, right;

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

public class BinarySearchTree {
   Node root;

   BinarySearchTree() {
      root = null;
   }

   // Insertion Operation
   void insert(int data) {
      root = insertRec(root, data);
   }

   Node insertRec(Node root, int data) {
      if (root == null) {
         root = new Node(data);
         return root;
      }
      if (data < root.data) {
         root.left = insertRec(root.left, data);
```

```
    } else if (data > root.data) {
        root.right = insertRec(root.right, data);
    }
    return root;
}

// Searching Operation
boolean search(int data) {
    return searchRec(root, data);
}

boolean searchRec(Node root, int data) {
    if (root == null) {
        return false;
    }
    if (root.data == data) {
        return true;
    }
    if (data < root.data) {
        return searchRec(root.left, data);
    }
    return searchRec(root.right, data);
}

// Deletion Operation
void delete(int data) {
    root = deleteRec(root, data);
}

Node deleteRec(Node root, int data) {
    if (root == null) {
        return root;
    }

    if (data < root.data) {
        root.left = deleteRec(root.left, data);
    } else if (data > root.data) {
        root.right = deleteRec(root.right, data);
    } else {
        if (root.left == null) {
            return root.right;
        } else if (root.right == null) {
            return root.left;
        }
```

```
            root.data = minValue(root.right);
            root.right = deleteRec(root.right, root.data);
        }
        return root;
    }

    int minValue(Node root) {
        int minValue = root.data;
        while (root.left != null) {
            minValue = root.left.data;
            root = root.left;
        }
        return minValue;
    }

    // Modification Operation
    void modify(int oldData, int newData) {
        root = deleteRec(root, oldData);
        insert(newData);
    }

    // In-order Traversal (to visualize the tree structure)
    void inOrder() {
        inOrderRec(root);
        System.out.println();
    }

    void inOrderRec(Node root) {
        if (root != null) {
            inOrderRec(root.left);
            System.out.print(root.data + " ");
            inOrderRec(root.right);
        }
    }

    // Main method with menu options
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        Scanner sc = new Scanner(System.in);
        boolean exit = false;

        while (!exit) {
            System.out.println("\n--- Binary Search Tree Operations ---");
```

```
System.out.println("1. Insert");
System.out.println("2. Delete");
System.out.println("3. Search");
System.out.println("4. Modify");
System.out.println("5. In-Order Traversal");
System.out.println("6. Exit");
System.out.print("Choose an option (1-6): ");

int choice = sc.nextInt();

switch (choice) {
   case 1: // Insert
      System.out.print("Enter value to insert: ");
      int insertValue = sc.nextInt();
      bst.insert(insertValue);
      System.out.println(insertValue + " inserted.");
      break;

   case 2: // Delete
      System.out.print("Enter value to delete: ");
      int deleteValue = sc.nextInt();
      bst.delete(deleteValue);
      System.out.println(deleteValue + " deleted.");
      break;

   case 3: // Search
      System.out.print("Enter value to search: ");
      int searchValue = sc.nextInt();
      if (bst.search(searchValue)) {
         System.out.println(searchValue + " found in the BST.");
      } else {
         System.out.println(searchValue + " not found in the BST.");
      }
      break;

   case 4: // Modify
      System.out.print("Enter old value to modify: ");
      int oldValue = sc.nextInt();
      System.out.print("Enter new value to replace: ");
      int newValue = sc.nextInt();
      bst.modify(oldValue, newValue);
      System.out.println("Modified " + oldValue + " to " + newValue);
      break;
```

```
            case 5: // In-order Traversal
                System.out.println("In-Order Traversal of the tree:");
                bst.inOrder();
                break;

            case 6: // Exit
                System.out.println("Exiting...");
                exit = true;
                break;

            default:
                System.out.println("Invalid choice! Please select a valid option.");
        }
    }

    sc.close();
  }
}
```