BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Name	Debjit Ghosal
UID no.	2023300065
Experiment No.	5

AIM:	**Tree data structures: Write a program to construct a binary search tree, insert an element in BST, delete an element from BST and traverse it.**
THEORY:	1. ***Binary Trees*** A binary tree is a fundamental data structure where each node has at most two children, the left and right child. This hierarchical structure begins with a single node called the root. The binary tree can be categorized into different types, including: - **Full Binary Tree**: Every node has either 0 or 2 children. - **Complete Binary Tree**: All levels are completely filled except possibly for the last level, which is filled from left to right. - **Perfect Binary Tree**: All internal nodes have exactly two children, and all leaf nodes are at the same,m level. - **Balanced Binary Tree:** The height difference between any node's left and right subtrees is at most 1. **2. *Binary Tree Operations*** Key operations performed on binary trees include: - **Insertion:** This operation involves adding a new node to the tree. In a binary search tree (BST), this involves placing the node in the correct position to maintain the BST property. - **Deletion:** Removing a node requires ensuring the tree remains valid. There are three scenarios: - Node to be deleted is a leaf (no children). - Node to be deleted has one child. - Node to be deleted has two children, where the node is replaced by its inorder predecessor or successor. - **Search**: To find a specific node, the tree is traversed starting from the root. In a BST, this operation is efficient due to the ordered nature of the tree. - **Traversal**: This involves visiting all nodes systematically. Traversal methods are crucial for accessing and processing nodes.

3. ***Binary Tree Traversals:***

Traversal methods are strategies for visiting each node in a binary tree:

- **Preorder Traversal** (Root → Left → Right): This method visits the root node first, then recursively traverses the left subtree, followed by the right subtree. It is used to create a copy of the tree or to evaluate expressions.
- **Inorder Traversal** (Left → Root → Right): This method visits the left subtree first, then the root, and finally the right subtree. In a BST, this traversal yields the nodes in ascending order.
- **Postorder Traversal** (Left → Right → Root): Here, the left and right subtrees are visited before the root. This is useful for deleting nodes or evaluating post-fix expressions.
- **Level-order Traversal**: Nodes are visited level by level starting from the root. This method uses a queue and is useful for finding the shortest path in unweighted trees.

4. ***Binary Search Trees*** (BST):

A Binary Search Tree is a specialized binary tree where:

- For any node, all values in the left subtree are less than the node's value.
- All values in the right subtree are greater than the node's value.

This property ensures that operations such as search, insertion, and deletion are efficient. In a balanced BST, these operations have a time complexity of O(log n), where n is the number of nodes.

5. ***Binary Tree Traversals in BST*:**

In BSTs, traversal methods have specific advantages:

- **Inorder Traversal**: Produces a sorted sequence of nodes. This is particularly useful for operations that require sorted data.
- **Preorder Traversal**: Useful for creating a copy of the tree or for prefix notation in expressions.
- **Postorder Traversal**: Essential for operations that require deleting nodes or evaluating postfix expressions.
- **Level-order Traversal**: Helps in operations that involve processing nodes level by level, such as printing tree structure or finding shortest paths.

Conclusion:

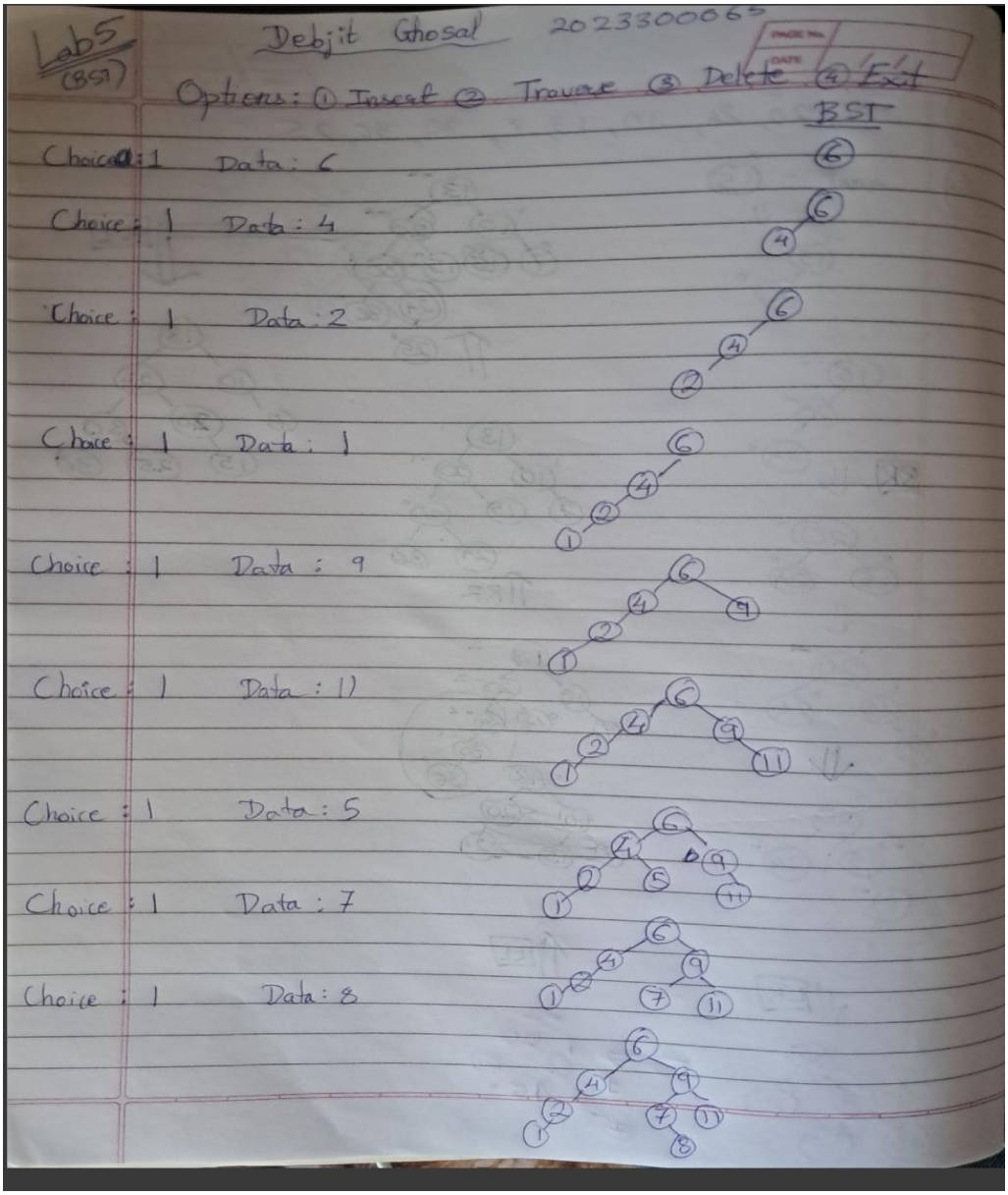
Binary trees are a crucial data structure with various forms and properties, such as full, complete, perfect, and balanced trees. Operations like insertion, deletion, and

	search are fundamental, with traversal methods (preorder, inorder, postorder, and level-order) providing different ways to process nodes. BSTs enhance binary trees by maintaining an ordered structure, enabling efficient operations. Understanding these concepts and their applications is vital for effective data management and manipulation.
ALGORITHM:	**Algorithm Steps** 1. **Define the Node Structure** o Create a structure Node that contains: ▪ An integer data field. ▪ Two pointers left and right pointing to the left and right child nodes, respectively. 2. **Create a Node** o Implement the function createNode(data): ▪ Allocate memory for a new node. ▪ Set the node's data to the provided value. ▪ Initialize the left and right pointers to NULL. ▪ Return the created node. 3. **Insert a Node in the BST** o Define the insert(root, data) function: ▪ If the root is NULL, create a new node with the given data and return it as the root. ▪ If data is less than the root's data, recursively insert the new node in the left subtree. ▪ If data is greater than the root's data, recursively insert the new node in the right subtree. ▪ Return the root after insertion. 4. **Find Minimum Value in a Subtree** o Define the function findMin(root): ▪ Traverse the left child nodes to find the node with the minimum value in the subtree. ▪ Return the node with the smallest value. 5. **Delete a Node in the BST** o Implement the deleteNode(root, data) function: ▪ If the root is NULL, return NULL (base case for recursion). ▪ If data is less than the root's data, recursively delete the node in the left subtree. ▪ If data is greater than the root's data, recursively delete the node in the right subtree.

- If the node to be deleted is found:
  - If the node has no left child, return the right child and free the node.
  - If the node has no right child, return the left child and free the node.
  - If the node has both children:
    - Find the minimum value node in the right subtree.
    - Replace the root's data with the minimum value node's data.
    - Recursively delete the minimum value node in the right subtree.
- Return the root after deletion.

6. **Tree Traversals**
   - **Inorder Traversal (inorder(root))**:
     - Recursively traverse the left subtree.
     - Print the root's data.
     - Recursively traverse the right subtree.
   - **Preorder Traversal (preorder(root))**:
     - Print the root's data.
     - Recursively traverse the left subtree.
     - Recursively traverse the right subtree.
   - **Postorder Traversal (postorder(root))**:
     - Recursively traverse the left subtree.
     - Recursively traverse the right subtree.
     - Print the root's data.
7. **Main Function with Menu**
   - Start with an empty tree (root = NULL).
   - Continuously display a menu with the following options:

1. **Insert an Element**:
   - Input the element to insert.
   - Call the insert() function.
   - Perform and display an inorder traversal of the tree after insertion.
2. **Delete an Element**:
   - Input the element to delete.
   - Call the deleteNode() function.
   - Perform and display an inorder traversal of the tree

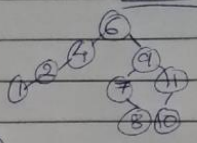
	after deletion. 3. **Traverse the Tree**: ▪ Perform and display the inorder, preorder, and postorder traversals of the tree. 4. **Exit the Program**. o Continue showing the menu until the user chooses to exit (option 4).
PROBLEM-SOLVING ON THE CONCEPT:	

Exp 5 (BST)

Debjit Ghosal 2023300065

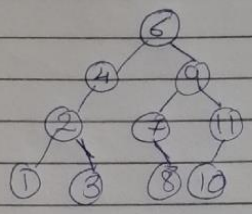
PAGE No. DATE / /

BST



Choice : 1 Data : 10

Choice : 1 Data 3



Preorder (VLR): Visit(print) left and then go to right

6 4 2 1 3 5 9 7 8 11 10

Inorder (LVR): go left, print and then go to right

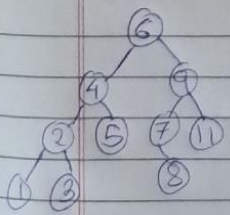
1 2 3 4 5 6 7 8 9 10 11

Postorder (LRV):

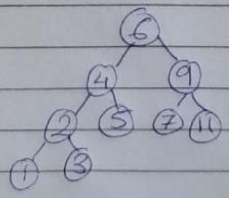
1 3 2 5 4 8 7 10 11 9 6

Deletion:

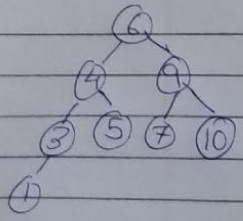
Delete: 11



Delete: 8



Delete: 2



CODE:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}
struct Node* insert(struct Node* root, int data) {
   if (root == NULL) {
      return createNode(data);
   }
   if (data < root->data) {
      root->left = insert(root->left, data);
   } else if (data > root->data) {
      root->right = insert(root->right, data);
   }
   return root;
}
struct Node* findMin(struct Node* root) {
   while (root->left != NULL) {
      root = root->left;
   }
   return root;
}
struct Node* deleteNode(struct Node* root, int data) {
   if (root == NULL) return root;

   if (data < root->data) {
      root->left = deleteNode(root->left, data);
   } else if (data > root->data) {
```

```
        root->right = deleteNode(root->right, data);
    } else {
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

void inorder(struct Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

void preorder(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

void postorder(struct Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
```

```
}

int main() {
    struct Node* root = NULL;
    int choice, element;

    do {
        printf("\n--- Binary Search Tree Operations ---\n");
        printf("1. Insert an element (Inorder)\n");
        printf("2. Delete an element (Inorder)\n");
        printf("3. Traverse the tree\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to insert: ");
                scanf("%d", &element);
                root = insert(root, element);
                printf("Inorder traversal after insertion: ");
                inorder(root);
                printf("\n");
                break;

            case 2:
                printf("Enter the element to delete: ");
                scanf("%d", &element);
                root = deleteNode(root, element);
                printf("Inorder traversal after deletion: ");
                inorder(root);
                printf("\n");
                break;

            case 3:
                printf("\nInorder traversal: ");
                inorder(root);
                printf("\nPreorder traversal: ");
```

```
            preorder(root);
            printf("\nPostorder traversal: ");
            postorder(root);
            printf("\n");
            break;

        case 4:
            printf("Exiting...\n");
            break;

        default:
            printf("Invalid choice! Please try again.\n");
      }
   } while (choice != 4);

   return 0;
}
```

OUTPUT SCREENSHOT:

```
--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 50
Inorder traversal after insertion: 50

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 40
Inorder traversal after insertion: 40 50

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 60
Inorder traversal after insertion: 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 30
```

```
Enter the element to insert: 60
Inorder traversal after insertion: 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 30
Inorder traversal after insertion: 30 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 25
Inorder traversal after insertion: 25 30 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 28
Inorder traversal after insertion: 25 28 30 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
```

```
Inorder traversal after deletion: 28 30 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 2
Enter the element to delete: 30
Inorder traversal after deletion: 28 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 10
Inorder traversal after insertion: 10 28 40 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 45
Inorder traversal after insertion: 10 28 40 45 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
```

```
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 23
Inorder traversal after insertion: 10 23 28 40 45 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 22
Inorder traversal after insertion: 10 22 23 28 40 45 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 21
Inorder traversal after insertion: 10 21 22 23 28 40 45 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 1
Enter the element to insert: 17
Inorder traversal after insertion: 10 17 21 22 23 28 40 45 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
```

```
Enter your choice: 2
Enter the element to delete: 28
Inorder traversal after deletion: 10 21 22 23 40 45 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 2
Enter the element to delete: 21
Inorder traversal after deletion: 10 22 23 40 45 50 60

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 3

Inorder traversal: 10 22 23 40 45 50 60
Preorder traversal: 50 40 10 23 22 45 60
Postorder traversal: 22 23 10 45 40 60 50

--- Binary Search Tree Operations ---
1. Insert an element (Inorder)
2. Delete an element (Inorder)
3. Traverse the tree
4. Exit
Enter your choice: 4
Exiting...


...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:	In this experiment, I gained a thorough understanding of Binary Search Trees (BSTs). I learned how nodes are structured with left and right children, ensuring values in the left subtree are smaller and values in the right subtree are larger. I

	also learned insertion which involves recursively finding the correct position, while deletion handles various cases: removing leaf nodes, replacing nodes with one child, or finding the in-order successor for nodes with two children. Tree traversals (inorder, preorder, and postorder) provided insight into different ways of visiting nodes. Additionally, recursion played a crucial role in implementing these operations. Finally, I understood the time complexity, where balanced trees ensure operations occur in **O(log n)** time, though unbalanced trees may degrade to **O(n)**. This experiment greatly enhanced my grasp of tree data structures and algorithms.