**EXPERIMENT NO. 11**

| **Objective(s):**  Implement traversal algorithms (Preorder, Inorder, Postorder) on a Binary Search Tree (BST) to visit nodes in specific orders. |
| --- |
| **Outcome:**  Develop functions to traverse a BST and print nodes in Preorder (Root-Left-Right), Inorder (Left-Root-Right), and Postorder (Left-Right-Root) sequences, facilitating different ways to process or display tree elements. |
| **Problem Statement:**  Implement Traversals Preorder Inorder Postorder on BST. |
| **Background Study:**   * **Binary Search Tree (BST)** is a hierarchical data structure that organizes nodes in a binary tree format. Each node has at most two children, referred to as the left child and the right child. The structure follows a specific property:    + **Binary Search Property:** For every node n, all values in its left subtree are less than n, and all values in its right subtree are greater than n. * **Key Operations**    + **Creation:**     - Start with an empty tree or initialize with a root node.     - Insert nodes ensuring the binary search property is maintained.   + **Traversal:**     - Visit nodes in specific orders to access or process data.   + **Search:**     - Find a specific node based on its value using the BST properties.   + **Insertion:**     - Add new nodes while maintaining the BST properties.   + **Deletion:**     - Remove nodes while ensuring the BST properties are preserved. * **Traversal Algorithms**  Traversing a BST involves visiting nodes in a specific order. The three main traversal algorithms are:    + **Inorder Traversal (Left-Root-Right):**     - Visit the left subtree recursively.     - Visit the root node.     - Visit the right subtree recursively.     - In a BST, this traversal visits nodes in sorted order (ascending).   + **Preorder Traversal (Root-Left-Right):**     - Visit the root node.     - Visit the left subtree recursively.     - Visit the right subtree recursively.     - Preorder traversal is useful for creating a copy of the tree or evaluating prefix expressions.   + **Postorder Traversal (Left-Right-Root):**     - Visit the left subtree recursively.     - Visit the right subtree recursively.     - Visit the root node.     - Postorder traversal is useful for deleting a tree or evaluating postfix expressions. |

| **Algorithm:**  **Traversal**   * **Inorder Traversal:**   + Traverse the left subtree recursively.   + Visit the current node.   + Traverse the right subtree recursively.   + Used to print elements in non-decreasing order. * **Preorder Traversal:**   + Visit the current node.   + Traverse the left subtree recursively.   + Traverse the right subtree recursively.   + Used for creating a copy of the tree or prefix expression evaluation. * **Postorder Traversal:**   + Traverse the left subtree recursively.   + Traverse the right subtree recursively.   + Visit the current node.   + Used in deleting a tree or postfix expression evaluation. |
| --- |

| **Code:**  #include <stdio.h>  #include <stdlib.h>  // Definition of a BST node  struct TreeNode {  int data;  struct TreeNode\* left;  struct TreeNode\* right;  };  // Function to create a new node  struct TreeNode\* createNode(int data) {  struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));  if (newNode == NULL) {  fprintf(stderr, "Memory allocation failed\\n");  exit(EXIT\_FAILURE);  }  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  // Function to insert a new node in BST  struct TreeNode\* insertNode(struct TreeNode\* root, int data) {  // If the tree is empty, return a new node  if (root == NULL) {  return createNode(data);  }  // Otherwise, recur down the tree  if (data < root->data) {  root->left = insertNode(root->left, data);  } else if (data > root->data) {  root->right = insertNode(root->right, data);  }  // return the (unchanged) node pointer  return root;  }  // Function to find the inorder successor in BST  struct TreeNode\* minValueNode(struct TreeNode\* node) {  struct TreeNode\* current = node;  // Loop down to find the leftmost leaf  while (current && current->left != NULL) {  current = current->left;  }  return current;  }  // Function to delete a node from BST  struct TreeNode\* deleteNode(struct TreeNode\* root, int data) {  // Base case: If the tree is empty  if (root == NULL) {  return root;  }  // Recur down the tree  if (data < root->data) {  root->left = deleteNode(root->left, data);  } else if (data > root->data) {  root->right = deleteNode(root->right, data);  } else {  // Node found with the data  // Case 1: Node with only one child or no child  if (root->left == NULL) {  struct TreeNode\* temp = root->right;  free(root);  return temp;  } else if (root->right == NULL) {  struct TreeNode\* temp = root->left;  free(root);  return temp;  }  // Case 2: Node with two children  struct TreeNode\* temp = minValueNode(root->right);  // Copy the inorder successor's content to this node  root->data = temp->data;  // Delete the inorder successor  root->right = deleteNode(root->right, temp->data);  }  return root;  }  // Function to perform inorder traversal of BST  void inorderTraversal(struct TreeNode\* root) {  if (root != NULL) {  inorderTraversal(root->left);  printf("%d ", root->data);  inorderTraversal(root->right);  }  }  // Function to perform preorder traversal of BST  void preorderTraversal(struct TreeNode\* root) {  if (root != NULL) {  printf("%d ", root->data);  preorderTraversal(root->left);  preorderTraversal(root->right);  }  }  // Function to perform postorder traversal of BST  void postorderTraversal(struct TreeNode\* root) {  if (root != NULL) {  postorderTraversal(root->left);  postorderTraversal(root->right);  printf("%d ", root->data);  }  }  // Function to search for a node in BST  struct TreeNode\* search(struct TreeNode\* root, int data) {  // Base Cases: root is null or data is present at root  if (root == NULL || root->data == data) {  return root;  }  // data is greater than root's data  if (root->data < data) {  return search(root->right, data);  }  // data is smaller than root's data  return search(root->left, data);  }  // Example usage  int main() {  struct TreeNode\* root = NULL;  root = insertNode(root, 50);  insertNode(root, 30);  insertNode(root, 20);  insertNode(root, 40);  insertNode(root, 70);  insertNode(root, 60);  insertNode(root, 80);  printf("Inorder traversal of the BST: ");  inorderTraversal(root);  printf("\\n");  printf("Preorder traversal of the BST: ");  preorderTraversal(root);  printf("\\n");  printf("Postorder traversal of the BST: ");  postorderTraversal(root);  printf("\\n");  // Search for a node  int key = 40;  struct TreeNode\* result = search(root, key);  if (result) {  printf("Node %d found in the BST.\\n", key);  } else {  printf("Node %d not found in the BST.\\n", key);  }  // Delete a node  root = deleteNode(root, 20);  printf("Inorder traversal after deleting 20: ");  inorderTraversal(root);  printf("\\n");  return 0;  } |
| --- |
| **OUTPUT :** |