|  |
| --- |
| **Part A**  **Name:- Aryan Srivastava**  **Roll No:- A073**  **Subject:- Data Structures and Algorithms**  **Program: MCA Sem I** |
| **Aim:**  Implementation of Binary Search Tree: Insertion, deletion and Search operation on tree data structure |
| **Prerequisite:** C++ Programming |
| **Outcome:** Implementation of Binary Search Tree: Insertion, deletion and Search operation on tree data structure |
| * **Theory:** * **In binary search tree, each node has 0, 1, or at the most 2 children**   + **Key property: Value at node**     - **Smaller values in left subtree**     - **Larger values in right subtree**     **Binary tree- Node representation**   * struct node { * struct node \*left; * int data; * struct node \*right; * };     **Insert operation**  struct node \*insert(struct node \*root, int val)  {  if(root == NULL)  return getNewNode(val);  if(root->key < val)  root->right = insert(root->right,val);  else if(root->key > val)  root->left = insert(root->left,val);  return root;    }  struct node \*getNewNode(int val)  {  struct node \*newNode = new node;  newNode->key = val;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  **Delete operation**   * The delete function deletes a node from the binary search tree. * However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process.   Three cases:  (1) the node is a leaf   * + Delete it immediately   (2) the node has one child   * + Replace the node with its child   (3) the node has 2 children   * + i. replace the node with the minimum element at the right subtree (inorder successor)   + Ii. delete the minimum element by invoking case 1 or 2.   Node\* deleteNode(Node\* root, int k)  {      // Base case      if (root == NULL)          return root;        // Recursive calls for ancestors of      // node to be deleted      if (root->key > k) {          root->left = deleteNode(root->left, k);          return root;      }      else if (root->key < k) {          root->right = deleteNode(root->right, k);          return root;      }        // We reach here when root is the node      // to be deleted.  // If one of the children is empty      if (root->left == NULL) {          Node\* temp = root->right;          delete root;          return temp;      }      else if (root->right == NULL) {          Node\* temp = root->left;          delete root;          return temp;      }   // If both children exist      else {            Node\* succParent = root;            // Find successor          Node\* succ = root->right;          while (succ->left != NULL) {              succParent = succ;              succ = succ->left;          }            // Delete successor.  Since successor          // is always left child of its parent          // we can safely make successor's right          // right child as left of its parent.          // If there is no succ, then assign          // succ->right to succParent->right          if (succParent != root)              succParent->left = succ->right;          else              succParent->right = succ->right;            // Copy Successor Data to root          root->key = succ->key;            // Delete Successor and return root          delete succ;          return root;      }  }  **Search a node**  struct node\* search(struct node\* root, int key)  {      // Base Cases: root is null or key is present at root      if (root == NULL || root->key == key)          return root;        // Key is greater than root's key      if (root->key < key)          return search(root->right, key);        // Key is smaller than root's key     rReturn search(root->left, key);  }  **TASK 1:**  Write a C/C++ program to implement binary tree and perfrom the following operations:   * Insert * Delete * Search |
| **Procedure:**   1. Open CodeBlock editor or visual studio editor and write the code in C++. 2. Complile and run the code |
| **Instructions:**   1. Copy code & paste in code section and output of Part B. |
|  |

|  |
| --- |
| **Part B** |
| **Code**:  class Node {      int value;      Node left, right;      Node(int item) {          value = item;          left = right = null;      }  }  class BinaryTree {      Node root;      BinaryTree() {          root = null;      }      // Insert a new value into the tree      void insert(int value) {          root = insertingNode(root, value);      }      Node insertingNode(Node root, int value) {          if (root == null) {              root = new Node(value);              return root;          }          if (value < root.value) {              root.left = insertingNode(root.left, value);          } else if (value > root.value) {              root.right = insertingNode(root.right, value);          }          return root;      }      // Delete a value from the tree      void delete(int value) {          root = deletingNode(root, value);      }      Node deletingNode(Node root, int value) {          if (root == null) return root;          if (value < root.value) {              root.left = deletingNode(root.left, value);          } else if (value > root.value) {              root.right = deletingNode(root.right, value);          } else {              // Node with only one child or no child              if (root.left == null) return root.right;              else if (root.right == null) return root.left;              // Node with two children: Get the inorder successor (smallest in the right subtree)              root.value = minValue(root.right);              // Delete the inorder successor              root.right = deletingNode(root.right, root.value);          }          return root;      }      int minValue(Node root) {          int minValue = root.value;          while (root.left != null) {              minValue = root.left.value;              root = root.left;          }          return minValue;      }      // Search for a value in the tree      boolean search(int value) {          return searchingNode(root, value);      }      boolean searchingNode(Node root, int value) {          if (root == null) return false;          if (root.value == value) return true;          return value < root.value ? searchingNode(root.left, value) : searchingNode(root.right, value);      }      // Inorder traversal      void inorder() {          inordering(root);          System.out.println();      }      void inordering(Node root) {          if (root != null) {              inordering(root.left);              System.out.print(root.value + " ");              inordering(root.right);          }      }      // Preorder traversal      void preorder() {          preordering(root);          System.out.println();      }      void preordering(Node root) {          if (root != null) {              System.out.print(root.value + " ");              preordering(root.left);              preordering(root.right);          }      }      // Postorder traversal      void postorder() {          postordering(root);          System.out.println();      }      void postordering(Node root) {          if (root != null) {              postordering(root.left);              postordering(root.right);              System.out.print(root.value + " ");          }      }      public static void main(String[] args) {          BinaryTree bst = new BinaryTree();          bst.insert(50);          bst.insert(40);          bst.insert(30);          bst.insert(20);          bst.insert(70);          bst.insert(60);          bst.insert(80);          System.out.print("Inorder: ");          bst.inorder();          System.out.print("Preorder: ");          bst.preorder();          System.out.print("Postorder: ");          bst.postorder();          System.out.println("Search 40: " + bst.search(40));          System.out.println("Search 100: " + bst.search(100));          bst.delete(20);          System.out.print("Inorder after deleting 20: ");          bst.inorder();          bst.delete(30);          System.out.print("Inorder after deleting 30: ");          bst.inorder();          bst.delete(50);          System.out.print("Inorder after deleting 50: ");          bst.inorder();      }  } |
| **Output:** |
| **Observation & Learning:**  Write your Observations & Learning after performing task |
| Curiosity questions:  1. **Insert Operation**:   **Time Complexity**: O(h), where h is the height of the tree.   **Explanation**: In the worst case, the tree could be skewed (like a linked list), making the height equal to the number of nodes, n. Thus, the worst-case time complexity is O(n). In a balanced tree, the height is log(n), leading to an average-case time complexity of O(log n).  **Delete Operation**:   **Time Complexity**: O(h), where h is the height of the tree.   **Explanation**: Similar to the insert operation, the delete operation's time complexity depends on the tree's height. In the worst case, it is O(n) for a skewed tree and O(log n) for a balanced tree.  **Search Operation**:   **Time Complexity**: O(h), where h is the height of the tree.   **Explanation**: The search operation also depends on the tree's height. In the worst case, it is O(n) for a skewed tree and O(log n) for a balanced tree. |
| **Conclusion:**  We successfully implemented binary search tree and its operations: insert, delete and search using Java program. |