**Binary Tree and Binary Search Tree Implementation:**

|  |
| --- |
| **Part 1 - Class Attributes and Methods**:  **package** CtCILibrary;  **public** **class** BinaryTreeNode {  **public** **int** data;  **public** BinaryTreeNode left;  **public** BinaryTreeNode right;  **public** BinaryTreeNode parent;  **private** **int** size = 0;  **public** BinaryTreeNode(**int** d) {}  **private** **void** setLeftChild(BinaryTreeNode left) {}  **private** **void** setRightChild(BinaryTreeNode right) {}  **public** **int** size() {}    **public** **int** height() {}  **public** **void** insertInOrder(**int** d) {}  **public** **boolean** isBST() {}  **public** BinaryTreeNode find(**int** d) {}  **private** **static** BinaryTreeNode createMinimalBSTHelper(**int** arr[], **int** start, **int** end){}  **public** **static** BinaryTreeNode createMinimalBST(**int** array[]) {}  **public** **static** **void** visit(BinaryTreeNode node) {}  **public** **static** **void** inOrderTraversal(BinaryTreeNode node) {}  **public** **static** **void** preOrderTraversal(BinaryTreeNode node) {}  **public** **static** **void** postOrderTraversal(BinaryTreeNode node) {}  } |

|  |
| --- |
| **Part 2 – Attributes and Simple Methods:**  **public** **class** BinaryTreeNode {  **public** BinaryTreeNode(**int** d) {  data = d;  size = 1;  }  **private** **void** setLeftChild(BinaryTreeNode left) {  **this**.left = left;  **if** (left != **null**) {  left.parent = **this**;  }  }  **private** **void** setRightChild(BinaryTreeNode right) {  **this**.right = right;  **if** (right != **null**) {  right.parent = **this**;  }  }    **public** **int** size() {  **return** size;  }    **public** **int** height() {  **int** leftHeight;  **int** rightHeight;    **if**(left != **null**){  leftHeight = left.height();  }**else**{  leftHeight = 0;  }  **if**(right != **null**){  rightHeight = right.height();  }**else**{  rightHeight = 0;  }    **return** 1 + Math.*max*(leftHeight, rightHeight);  }  **Key Details:**   1. Before setting a child’s parent, ensure that child is not null or you will get a null pointer exception. 2. The height of a node is the (1 + the maximum of its two child node’s height). When you are getting the height of a child node, ensure that child is not null or you will get a null pointer exception. Always think about the null case when you use the “dot” operator in Java. |

|  |
| --- |
| **Part 3 – Logically Tricky Methods:**  **public** **void** insertInOrder(**int** d) {  **if**(d <= data && left != **null**){  left.insertInOrder(d);//insert on the left side  }**else** **if**(d > data && right != **null**){  right.insertInOrder(d);//insert on the right side  }**else** **if** (d <= data && left == **null**) {  setLeftChild(**new** BinaryTreeNode(d));//create a left tree node  }**else**{  setRightChild(**new** BinaryTreeNode(d));//create a right tree node  }  size++;  }  **public** **boolean** isBST() {  **if**(left != **null** && data < left.data){  **return** **false**;// parent's data shouldn't be < than its left child's  }**else** **if** (right != **null** && data >= right.data) {  **return** **false**;// parent's data shouldn't be >= than its right child's  }**else** **if**(left != **null** && !left.isBST()){  **return** **false**;//the left tree should be a BST for this to be a BST  }**else** **if**(right != **null** && !right.isBST()){  **return** **false**;//the right tree should be a BST for this to be a BST  }    **return** **true**;  }  **public** BinaryTreeNode find(**int** d) {  **if** (d == data) {  **return** **this**; //found it  } **else** **if** (d <= data && left!= **null**) {  **return** left.find(d);//search left side  } **else** **if** (d > data && right != **null**) {  **return** right.find(d);//search right side  }  **return** **null**; //not in the current node or left tree, or right tree  } |

|  |
| --- |
| **Part 3 – Creating Binary Search Trees from Arrays:**  **private** **static** BinaryTreeNode createMinimalBSTHelper(**int** arr[], **int** from, **int** to){  **if** (to < from) {  **return** **null**;  }    **int** mid = (from + to) / 2;  BinaryTreeNode binarySearchTree = **new** BinaryTreeNode(arr[mid]);  binarySearchTree.setLeftChild(*createMinimalBSTHelper*(arr, from, mid - 1));  binarySearchTree.setRightChild(*createMinimalBSTHelper*(arr, mid + 1, to));    **return** binarySearchTree;  }  **public** **static** BinaryTreeNode createMinimalBST(**int** array[]) {  **return** *createMinimalBSTHelper*(array, 0, array.length - 1);  }   1. You use a Binary Seach like algorithm to create the tree. You split the array in to parts. From {0 – mid -1} becomes the contents of the left tree, from {mid + 1, array.length -1} becomes the contents of the right tree, and the tree’s data value becomes mid. Eventually, the sub trees become null. |

|  |
| --- |
| **Part 4 – Binary Tree Traversals:**  **public** **static** **void** visit(BinaryTreeNode node) {  **if** (node != **null**) {  System.*out*.println(node.data);  }  }  **public** **static** **void** inOrderTraversal(BinaryTreeNode node) {  **if** (node != **null**) {  *inOrderTraversal*(node.left);  *visit*(node);  *inOrderTraversal*(node.right);  }  }  **public** **static** **void** preOrderTraversal(BinaryTreeNode node) {  **if** (node != **null**) {  *visit*(node);  *inOrderTraversal*(node.left);  *inOrderTraversal*(node.right);  }  }  **public** **static** **void** postOrderTraversal(BinaryTreeNode node) {  **if** (node != **null**) {  *inOrderTraversal*(node.left);  *inOrderTraversal*(node.right);  *visit*(node);  }  }   * **Pre Order:** Visit the node first, then visit its left tree, then visit its right tree (VLR). * **In Order:** Visit the left tree, then visit the node, then visit the right tree. In a BST, this prints th elements in ascending order (LVR). * **Post Order:** Visit the left tree, visit the right tree, THEN visit the node (LRV).   Always check if the node is null! |

|  |
| --- |
| **Part 5 – Complete Source Code:**  **public** **class** BinaryTreeNode {  **public** **int** data;  **public** BinaryTreeNode left;  **public** BinaryTreeNode right;  **public** BinaryTreeNode parent;  **private** **int** size = 0;  **public** BinaryTreeNode(**int** d) {  data = d;  size = 1;  }  **private** **void** setLeftChild(BinaryTreeNode left) {  **this**.left = left;  **if** (left != **null**) {  left.parent = **this**;  }  }  **private** **void** setRightChild(BinaryTreeNode right) {  **this**.right = right;  **if** (right != **null**) {  right.parent = **this**;  }  }  **public** **int** size() {  **return** size;  }  **public** **int** height() {  **int** leftHeight;  **int** rightHeight;  **if**(left != **null**){  leftHeight = left.height();  }**else**{  leftHeight = 0;  }  **if**(right != **null**){  rightHeight = right.height();  }**else**{  rightHeight = 0;  }  **return** 1 + Math.*max*(leftHeight, rightHeight);  }  **public** **void** insertInOrder(**int** d) {  **if**(d <= data && left != **null**){  left.insertInOrder(d);//insert on the left side  }**else** **if**(d > data && right != **null**){  right.insertInOrder(d);//insert on the right side  }**else** **if** (d <= data && left == **null**) {  setLeftChild(**new** BinaryTreeNode(d));//create a left tree node  }**else**{  setRightChild(**new** BinaryTreeNode(d));//create a right tree node  }  size++;  }  **public** **boolean** isBST() {  **if**(left != **null** && data < left.data){  **return** **false**;// parent's data shouldn't be < than its left child's  }**else** **if** (right != **null** && data >= right.data) {  **return** **false**;// parent's data shouldn't be >= than its right child's  }**else** **if**(left != **null** && !left.isBST()){  **return** **false**;//the left tree should be a BST for this to be a BST  }**else** **if**(right != **null** && !right.isBST()){  **return** **false**;//the right tree should be a BST for this to be a BST  }    **return** **true**;  }  **public** BinaryTreeNode find(**int** d) {  **if** (d == data) {  **return** **this**; //found it  } **else** **if** (d <= data && left!= **null**) {  **return** left.find(d);//search left side  } **else** **if** (d > data && right != **null**) {  **return** right.find(d);//search right side  }  **return** **null**; //not in the current node or left tree, or right tree  }  **private** **static** BinaryTreeNode createMinimalBSTHelper(**int** arr[], **int** start, **int** end){  **if** (end < start) {  **return** **null**;  }    **int** mid = (start + end) / 2;  BinaryTreeNode binarySearchTree = **new** BinaryTreeNode(arr[mid]);  binarySearchTree.setLeftChild(*createMinimalBSTHelper*(arr, start, mid - 1));  binarySearchTree.setRightChild(*createMinimalBSTHelper*(arr, mid + 1, end));    **return** binarySearchTree;  }  **public** **static** BinaryTreeNode createMinimalBST(**int** array[]) {  **return** *createMinimalBSTHelper*(array, 0, array.length - 1);  }  **public** **static** **void** visit(BinaryTreeNode node) {  **if** (node != **null**) {  System.*out*.println(node.data);  }  }  **public** **static** **void** inOrderTraversal(BinaryTreeNode node) {  **if** (node != **null**) {  *inOrderTraversal*(node.left);  *visit*(node);  *inOrderTraversal*(node.right);  }  }  **public** **static** **void** preOrderTraversal(BinaryTreeNode node) {  **if** (node != **null**) {  *visit*(node);  *inOrderTraversal*(node.left);  *inOrderTraversal*(node.right);  }  }  **public** **static** **void** postOrderTraversal(BinaryTreeNode node) {  **if** (node != **null**) {  *inOrderTraversal*(node.left);  *inOrderTraversal*(node.right);  *visit*(node);  }  }  } |