**Binary Tree**

**Binary Tree**

|  |
| --- |
| **Binary Tree** combines advantages of an ordered array [fast search] and a linked list [fast insertion, deletion]. |

**Efficiency of Binary Tree**

|  |
| --- |
| Search, insertion, deletion operations are proportional to the length of the path. Therefore it is desirable to have binary search trees that are “shallow” rather than “deep.” A binary tree height is **0**, empty binary tree is **-1**. |

**Terminology: path, root, parent, child, leaf, binary tree.**

|  |
| --- |
| **Path –** the resulting sequence of nodes.  **Root** – the node at the top of the tree, there must be only one path from the root to any other node.  **Parent** – the node above. **Child** – the node below.  **Leaf** – a node that has no children. [One root, many leaves]  **Binary search tree** – *left child* is less than its parent, *right child* is greater than its parent. |

**The AVL Tree**

|  |
| --- |
| **An AVL tree** is a binary search tree in which the heights of the left and right subtree of every node differ by at most 1. |

**Adding Nodes to the Tree**

|  |
| --- |
| Node left = **new** Node(2);  Node right = **new** Node(3, **new** Node(4), **null**);  root = **new** Node(1, left, right); |

**Class Node**

|  |
| --- |
| **class** Node  {  **public** **int** value;  **public** Node left;  **public** Node right;    **public** Node(**int** val, Node l, Node r)  {  value = val;  left = l;  right = r;  }    **public** Node(**int** val)  {  **this**(val, **null**, **null**);  }  } |

**Class Tree**

|  |
| --- |
| **class** BinaryTree  {  **private** Node root = **null**;  } |

**Class RemovalResult**

|  |
| --- |
| **class** RemovalResult  {  **public** Node node; // removed node  **public** Node tree; // Remaining tree    **public** RemovalResult(Node node, Node tree)  {  **this**.node = node;  **this**.tree = tree;  }  } |

**Traversal**

|  |  |  |
| --- | --- | --- |
| **public** **void** preOrder()  {  preOrder(root);  }  **private** **void** preOrder(Node btree)  {  **if**(btree != **null**)  {  System.*out*.println(btree.value);  preOrder(btree.left);  preOrder(btree.right);  }  } | **public** **void** inOrder()  {  inOrder(root);  }  **private** **void** inOrder(Node btree)  {  **if**(btree != **null**)  {  inOrder(btree.left);  System.*out*.println(btree.value);  inOrder(btree.right);  }  } | **public** **void** postOrder()  {  postOrder(root);  }  **private** **void** postOrder(Node btree)  {  **if**(btree != **null**)  {  postOrder(btree.left);  postOrder(btree.right);  System.*out*.print(btree.value);  }  } |
| **• Preorder traversal:** process the data at the **root node**, traverse the **left subtree**, and then traverse the **right subtree**  **• Inorder traversal:** traverse the **left subtree**, process the data at the **root node**, and then traverse the **right subtree**  **• Postorder traversal:** traverse the **left subtree**, traverse the **right subtree**, and then process the data at the **root node** | | |

**The add() Method – O(log2N) provided that the BST is well-balanced (O(n) – worst case)**

|  |
| --- |
| **public** **boolean** add(**int** val)  {  root = add(val, root);  **return** **true**;  }  **private** Node add(**int** val, Node btree)  {  **if**(btree == **null**)  **return** **new** Node(val);  **if**(val < btree.value)  btree.left = add(val, btree.left);  **else**  btree.right = add(val, btree.right);  **return** btree;  } |

**The contains() Method – O(log2N) provided that the BST is well-balanced (O(n) – worst case)**

|  |
| --- |
| **public** **boolean** **contains**(**int** val)  {  **return** **contains**(val, root);  }  **private** **boolean** contains(**int** val, **Node** btree)  {  **if**(btree == **null**)  **return** **false**;  **if**(val == btree.value)  **return** **true**;  **if**(val < btree.value)  **return** contains(val, btree.left);  **else**  **return** contains(val, btree.right);  } |

**The remove() Method – O(log2N) provided that the BST is well-balanced (O(n) – worst case)**

|  |
| --- |
| **Sidestep deletion**: Add a new Boolean field to the node class. To delete a node, set the field to true. Then other operations, like find() check this field to be sure the node isn’t marked down before working with it. |
| **Deleting a leaf (node with no children):** simply remove a leaf from the tree by setting the node to null. |
| **Deleting a node with one child:** remove the node and replace it with its child. |
| **Deleting a node with two children:** Call the node to be deleted N. [do not delete it] Choose either its in-order successor node or its in-order predecessor node, R. Replace the value N with the value of R, then delete R. |
| **public** **boolean** remove(**int** x)  {  RemovalResult result = remove(root, x);  **if**(result == **null**)  **return** **false**;  **else**  {  root = result.tree;  **return** **true**;  }  }  **private** RemovalResult remove(Node bTree, **int** val)  {  **if**(bTree == **null**)  **return** **null**;  **if**(val < bTree.value) // Remove val from the left subtree  {  RemovalResult result = remove(bTree.left, val);  **if**(result == **null**)  **return** **null**;  bTree.left = result.tree;  result.tree = bTree;  **return** result;  }    **if**(val > bTree.value) // Remove val from the right subtree  {  RemovalResult result = remove(bTree.right, val);  **if**(result == **null**)  **return** **null**;  bTree.right = result.tree;  result.tree = bTree;  **return** result;  }  **if**(bTree.left == **null** && bTree.right == **null**) // val is in this node, is it a leaf?  **return** **new** RemovalResult(bTree, **null**);    **if**(bTree.left != **null** && bTree.right != **null**) // Does the node have two children?  {  RemovalResult remResult = removeLargest(bTree.left);  Node newRoot = remResult.node;  newRoot.left = remResult.tree;  newRoot.right = bTree.right;    bTree.left = **null**; // Prepare the result to be returned.  bTree.right = **null**;  **return** **new** RemovalResult(bTree, newRoot);  }    Node node = bTree; // The node has one child  Node tree;  **if**(bTree.left != **null**)  tree = bTree.left;  **else**  tree = bTree.right;  node.left = **null**;  node.right = **null**;  **return** **new** RemovalResult(node, tree);  }  **private** RemovalResult removeLargest(Node bTree)  {  **if**(bTree == **null**)  **return** **null**;  **if**(bTree.right == **null**)  {  Node tree = bTree.left; // Root is the largest node  bTree.left = **null**;  **return** **new** RemovalResult(bTree, tree);  }  **else**  {  RemovalResult remResult = removeLargest(bTree.right); // Remove the largest from the right subtree  bTree.right = remResult.tree;  remResult.tree = bTree;  **return** remResult;  }  } |