Continue

AVL Trees maintain balance through rotation operations, ensuring a minimum tree height. This differs from Binary Search Trees, which can become severely unbalanced. By balancing itself, an AVL Tree guarantees fast search, insert, and delete times. In contrast to the unbalanced Binary Search Tree above, the AVL Tree maintains a balanced structure, significantly reducing its height. To balance an AVL Tree, left or right rotations are performed as needed. This process is visualized in the animation below. The balance factor of each node is calculated by comparing the heights of its subtrees. If this factor exceeds 1 or falls below -1, a rotation operation is triggered to restore balance. There are four distinct "out-of-balance" cases that require specific rotations: - Left-Left (LL): Both the unbalanced node and its left child are left-heavy; a single right rotation rectifies the imbalance. - Right-Right (RR): The unbalanced node and its right child are both right-heavy; a single left rotation corrects the imbalance. - Left-Right (LR): The unbalanced node is left-heavy, while its left child is right-heavy; this necessitates a two-step process of left rotation on the left child followed by a right rotation on the unbalanced node. - Right-Left (RL): The unbalanced node is right-heavy, and its right child is left-heavy; a two-step approach involves first performing a right rotation on the right child and then a left rotation on the unbalanced node. To further understand AVL Trees' balancing mechanisms, refer to the provided animations for detailed explanations of these cases. The Left-Right (LR) Case in AVL Trees Explained When a node is left heavy, but its left child node is right heavy, a Left-Right rotation is performed to restore balance. This type of case occurs when the unbalanced node's left child node is right heavy. A single step through the animation reveals two instances where this happens: When K is inserted and nodes C, F, and G are added, resulting in Q becoming unbalanced with a balance factor of -2. Similarly, another instance occurs after nodes E, C, and F are inserted, where B becomes unbalanced due to its right child node D being left heavy. After inserting node B, a Right-Left case occurs due to node A becoming unbalanced and right heavy, with its right child being left heavy. To restore balance, a right rotation is performed on node F, followed by a left rotation on node A. The next Right-Left case emerges after nodes G, E, and D are added, as B becomes unbalanced and right heavy, while its right child F is left heavy. To correct this, a right rotation is initially done on node F, then followed by a left rotation on node B. To insert data into an AVL tree, we first compare the input data with the node's data. If the data is less than the node's data, we recursively call the insert function on the left subtree; otherwise, we call it on the right subtree. After insertion, we update the height of the node and check for balance. If the balance factor is greater than 1 or less than -1, we perform rotations to balance the tree. When deleting a node from an AVL tree, we first find the node to be deleted. If the node has no children, we simply remove it. If it has one child, we replace the node with its child. If it has two children, we find the node's in-order successor (the smallest node in the right subtree), replace the node's data with the successor's data, and then delete the successor. The time complexity of AVL tree operations such as search, insert, and delete is O(log n), where n is the number of nodes in the tree. This is because AVL trees are self-balancing, ensuring that the height of the tree remains relatively small by rotating nodes when the balance factor becomes too large. In contrast, unbalanced binary search trees can have a time complexity of O(n) in the worst case. To achieve this efficiency, AVL trees use a balance factor to determine when rotations are needed. The balance factor is calculated as the difference between the height of the left subtree and the height of the right subtree. If the balance factor becomes too large, the tree is rotated to restore balance. In addition to insert and delete operations, AVL trees also support in-order traversal, which visits nodes in ascending order. This is useful for applications where data needs to be processed in a specific order. Overall, AVL trees provide an efficient way to store and manipulate data, with fast search, insert, and delete operations. Their self-balancing property ensures that the tree remains approximately balanced, even after multiple insertions and deletions, which helps to maintain optimal performance. The time complexity of an AVL Tree can be broken down into two parts: slow operations like searching, inserting, and deleting nodes with a time complexity of O(h) = O(n), which is linear in the number of nodes. However, AVL Trees are self-balancing, meaning their height is kept to a minimum. To understand why this is faster than traditional Binary Search Trees (BSTs), imagine a perfect binary tree where each level has twice as many nodes as the previous one. The number of nodes at each level follows a power-of-2 pattern. For example, with 3 levels, you'd have 1 + 2 + 4 = 7 nodes. The key insight is that the total number of nodes in a perfect binary tree can be expressed as n_h = 2^(h+1) - 1. By using this formula, we can find the height (h) of a tree given its number of nodes (n). To derive O(log(n)) time complexity for operations like search, insert, and delete, we need to relate the height (h) to the number of nodes (n). By applying logarithmic properties, we get: * n = 2^(h+1) - 1 * n + 1 = 2^(h+1) * log2(n + 1) = log2(2^(h+1)) * h = log2(n + 1) - 1 This shows that O(h) is equivalent to O(log(n)) for search, insert, and delete operations in an AVL Tree. This is because the "+1" and "-1" terms become negligible when considering time complexity. In summary, AVL Trees are self-balancing Binary Search Trees with a height that remains relatively constant after insertion and deletion operations. As a result, their time complexity for operations like search, insert, and delete is much faster than traditional BSTs, making them more efficient. we can guarantee an upper bound of O(log(n)) for all these operations. The height of an AVL tree is always O(log(n)) where n is the number of nodes in the tree. Insertion in AVL Tree:To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). Left Rotation Right RotationT1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side) y x / \ Right Rotation / \ x T3 - - - - - - - > T1 y / \ < - - - - - - - / \ T1 T2 Left Rotation T2 T3 Keys in both of the above trees follow the following order keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)So BST property is not violated anywhere.Steps to follow for insertion: Let the newly inserted node be w Perform standard BST insert for w. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.Following are the possible 4 arrangements: y is the left child of z and x is the left child of y (Left Left Case) y is the left child of z and x is the right child of y (Left Right Case) y is the right child of z and x is the left child of y (Right Left Case) y is the right child of z and x is the right child of y (Right Right Case)Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of the subtree (After appropriate rotations) rooted with z becomes the same as it was before insertion. The Left Left case or left Right case is determined by comparing the newly inserted key with the key in the left subtree root. If the balance factor is less than -1, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root. Below is the implementation of the above approach using AVL tree: ```cpp // C++ program to insert a node in AVL tree #include using namespace std; struct Node { int key; Node *left; Node *right; int height; Node(int k) { key = k; left = nullptr; right = nullptr; height = 1; } }; int height(Node *N) { if (N == nullptr) return 0; return N->height; } Node *rightRotate(Node *y) { Node *x = y->left; Node *T2 = x->right; x->right = y; y->left = T2; y->height = 1 + max(height(y->left), height(y->right)); x->height = 1 + max(height(x->left), height(x->right)); return x; } Node *leftRotate(Node *x) { Node *y = x->right; Node *T2 = y->left; y->left = x; x->right = T2; x->height = 1 + max(height(x->left), height(x->right)); y->height = 1 + max(height(y->left), height(y->right)); return y; } int getBalance(Node *N) { if (N == nullptr) return 0; return height(N->left) - height(N->right); } Node* insert(Node* node, int key) { // Perform the normal BST insertion if (node == nullptr) return new Node(key); if (key < node->key) node->left = insert(node->left, key); else if (key > node->key) node->right = insert(node->right, key); else // Equal keys are not allowed in BST return node; node->height = 1 + max(height(node->left), height(node->right)); int balance = getBalance(node); if (balance > 1 && key < node->left->key) return rightRotate(node); if (balance < -1 && key > node->right->key) return leftRotate(node); if (balance > 1 && key > node->left->key) { node->left = leftRotate(node->left); return rightRotate(node); } if (balance < -1 && key < node->right->key) { node->right = rightRotate(node->right); return leftRotate(node); } return node; } void preOrder(Node *root) { if (root != nullptr) { cout key left); preOrder(root->right); } } int main() { Node *root = nullptr; root = insert(root, 10); root = insert(root, 20); root = insert(root, 30); root = insert(root, 40); root = insert(root, 50); root = insert(root, 25); cout height; } // A utility function to get maximum of two integers int max(int a, int b) { return (a > b)? a : b; } /* Helper function that allocates a new node with the given key and NULL left and right pointers. */ struct Node* newNode(int key) { struct Node* node = (struct Node*) malloc(sizeof(struct Node)); node->key = key; node->left = NULL; node->right = NULL; node->height = 1; // new node is initially added at leaf return(node); } // A utility function to right rotate subtree rooted with y // See the diagram given above. struct Node *rightRotate(struct Node *y) { struct Node *x = y->left; struct Node *T2 = x->right; // Perform rotation x->right = y; y->left = T2; // Update heights y->height = max(height(y->left), height(y->right)) + 1; x->height = max(height(x->left), height(x->right)) + 1; // Return new root return x; } // A utility function to left rotate subtree rooted with x // See the diagram given above. struct Node *leftRotate(struct Node *x) { struct Node *y = x->right; struct Node *T2 = y->left; // Perform rotation y->left = x; x->right = T2; // Update heights x->height = max(height(x->left), height(x->right)) + 1; y->height = max(height(y->left), height(y->right)) + 1; // Return new root return y; } // Get Balance factor of node N int getBalance(struct Node *N) { if (N == NULL) return 0; return height(N->left) - height(N->right); } // Recursive function to insert a key in the subtree rooted // with node and returns the new root of the subtree. struct Node* insert(struct Node* node, int key) { /* 1. Perform the normal BST insertion */ if (node == NULL) return(newNode(key)); if (key < node->key) node->left = insert(node->left, key); else if (key > node->key) node->right = insert(node->right, key); else // Equal keys are not allowed in BST return node; /* 2. Update height of this ancestor node */ node->height = 1 + max(height(node->left), height(node->right)); /* 3. Get the balance factor of this ancestor node to check whether this node became unbalanced */ int balance = getBalance(node); // If this node becomes unbalanced, then // there are 4 cases // Left Left Case if (balance > 1 && key < node->left->key) return rightRotate(node); // Right Right Case if (balance < -1 && key > node->right->key) return leftRotate(node); // Left Right Case if (balance > 1 && key > node->left->key) { node->left = leftRotate(node->left); return rightRotate(node); } // Right Left Case if (balance < -1 && key < node->right->key) { node->right = rightRotate(node->right); return leftRotate(node); } /* return the (unchanged) node pointer */ return node; } // A utility function to print preorder traversal // of the tree. // The function also prints height of every node void preOrder(struct Node *root) { if(root != NULL) { printf("%d ", root->key); preOrder(root->left); preOrder(root->right); } } /* Driver program to test above function*/ int main() { struct Node *root = NULL; /* Constructing tree given in the above figure */ root = insert(root, 10); root = # Constructing an AVL Tree in Java ## Overview This article demonstrates how to construct and balance an AVL (Adelson-Velskii-Landis) tree in Java. An AVL tree is a self-balancing binary search tree that ensures the height of the tree remains relatively constant even after insertions or deletions. ## Code ```java import java.util.*; class Node { int key; Node left; Node right; int height; public Node(int k) { key = k; left = null; right = null; height = 1; } } public class GfG { // A utility function to get the height of the tree static int height(Node N) { if (N == null) return 0; return N.height; } // A utility function to right rotate subtree rooted with y static Node rightRotate(Node y) { Node x = y.left; Node T2 = x.right; x.right = y; y.left = T2; y.height = 1 + Math.max(height(y.left), height(y.right)); x.height = 1 + Math.max(height(x.left), height(x.right)); return x; } // A utility function to left rotate subtree rooted with x static Node leftRotate(Node x) { Node y = x.right; Node T2 = y.left; y.left = x; x.right = T2; x.height = 1 + Math.max(height(x.left), height(x.right)); y.height = 1 + Math.max(height(y.left), height(y.right)); return y; } // Get balance factor of node N static int getBalance(Node N) { if (N == null) return 0; return height(N.left) - height(N.right); } // Recursive function to insert a key in the subtree rooted with node static Node insert(Node node, int key) { if (node == null) return new Node(key); if (key < node.key) node.left = insert(node.left, key); else if (key > node.key) node.right = insert(node.right, key); else // Equal keys are not allowed in BST return node; node.height = 1 + Math.max(height(node.left), height(node.right)); int balance = getBalance(node); if (balance > 1 && key < node.left.key) return rightRotate(node); if (balance < -1 && key > node.right.key) return leftRotate(node); if (balance > 1 && key > node.left.key) { node.left = leftRotate(node.left); return rightRotate(node); } if (balance < -1 && key < node.right.key) { node.right = rightRotate(node.right); return leftRotate(node); } return node; } // A utility function to print preorder traversal of the tree static void preOrder(Node root) { if (root != null) { System.out.print(root.key + " "); preOrder(root.left); preOrder(root.right); } } public static void main(String[] args) { Node root = null; // Constructing tree given in the above figure root = insert(root, 10); root = insert(root, 20); root = insert(root, 30); root = insert(root, 40); root = insert(root, 50); root = insert(root, 25); /* The constructed AVL Tree would be 30 / \ 20 40 / \ \ 10 25 50 */ System.out.println("Preorder traversal : "); preOrder(root); } } ``` ## Explanation This Java program constructs an AVL tree and performs a preorder traversal of the tree. 1. **Node Class**: The `Node` class represents a node in the AVL tree, containing an integer key, references to its left and right children, and its height. 2. **Height Function**: The `height` function calculates the height of a given node in the tree by recursively calling itself until it reaches a leaf node. 3. **Right Rotate** and **Left Rotate**: These functions perform rotations on the tree to balance it after insertion or deletion. 4. **Get Balance Factor**: This function returns the balance factor of a node, which is essential for maintaining the AVL tree's balance property. 5. **Insert Function**: The `insert` function inserts a key into the AVL tree while balancing the tree accordingly. 6. **Preorder Traversal**: The `preOrder` function prints the keys in the preorder traversal order. ## Example Output ``` Preorder traversal : 30 / \ 20 40 / \ \ 10 25 50 ``` Given article text here public class AVLTree { public class Node { public int Key; public Node Left; public Node Right; public int Height; public Node(int key) { Key = key; Left = null; Right = null; Height = 1; } } // Utility function to right rotate # subtree rooted with y def right_rotate(y): x = y.left T2 = x.right # Perform rotation x.right = y y.left = T2 # Update heights y.height = 1 + max(height(y.left), height(y.right)) x.height = 1 + max(height(x.left), height(x.right)) # Return new root return x # A utility function to left rotate # subtree rooted with x def left_rotate(x): y = x.right T2 = y.left # Perform rotation y.left = x x.right = T2 # Update heights x.height = 1 + max(height(x.left), height(x.right)) y.height = 1 + max(height(y.left), height(y.right)) # Return new root return y # Get balance factor of node N def get_balance(node): if not node: return 0 return height(node.left) - height(node.right) # Recursive function to insert a key in # the subtree rooted with node def insert(node, key): # Perform the normal BST insertion if not node: return Node(key) if key < node.key: node.left = insert(node.left, key) elif key > node.key: node.right = insert(node.right, key) else: # Equal keys are not allowed in BST return node # Update height of this ancestor node node.height = 1 + max(height(node.left), height(node.right)) # Get the balance factor of this ancestor node balance = get_balance(node) # If this node becomes unbalanced, # then there are 4 cases if balance > 1 and key < node.left.key: return right_rotate(node) if balance < -1 and key > node.right.key: return left_rotate(node) if balance > 1 and key > node.left.key: node.left = left_rotate(node.left) return right_rotate(node) if balance < -1 and key < node.right.key: node.right = right_rotate(node.right) return left_rotate(node) # Return the (unchanged) node pointer return node # A utility function to print preorder # traversal of the tree def pre_order(root): if root: print(root.key, end=" ") pre_order(root.left) pre_order(root.right) # Driver code root = None # Constructing tree given in the above figure root = insert(root, 10) root = insert(root, 20) root = insert(root, 30) root = insert(root, 40) root = insert(root, 50) root = insert(root, 25) # The constructed AVL Tree would be # 30 # / \ # 20 40 # / \ \ # 10 25 50 print("Preorder traversal :") pre_order(root) public static Node Insert(Node node, int key) { if (node == null) return new Node(key); if (key < node.Key) node.Left = Insert(node.Left, key); else if (key > node.Key) node.Right = Insert(node.Right, key); else return node; node.Height = 1 + Math.Max(Height(node.Left), Height(node.Right)); int balance = GetBalance(node); if (balance > 1 && key < node.Left.Key) return RightRotate(node); if (balance < -1 && key > node.Right.Key) return LeftRotate(node); if (balance > 1 && key > node.Left.Key) { node.Left = LeftRotate(node.Left); return RightRotate(node); } if (balance < -1 && key < node.Right.Key) { node.Right = RightRotate(node.Right); return LeftRotate(node); } return node; } static void PreOrder(Node root) { if (root != null) { Console.Write(root.Key + " "); PreOrder(root.Left); PreOrder(root.Right); } } public static void Main() { Node root = null; root = Insert(root, 10); root = Insert(root, 20); root = Insert(root, 30); root = Insert(root, 40); root = Insert(root, 50); root = Insert(root, 25); Console.WriteLine("Preorder traversal :"); PreOrder(root); } Given text: "the balance factor of this ancestor node const balance = getBalance(node); // If this node becomes unbalanced, then there are 4 cases // Left Left Case if (balance > 1 && key < node.left.key) { return rightRotate(node); } // Right Right Case if (balance < -1 && key > node.right.key) { return leftRotate(node); } // Left Right Case if (balance > 1 && key > node.left.key) { node.left = leftRotate(node.left); return rightRotate(node); } // Right Left Case if (balance < -1 && key < node.right.key) { node.right = rightRotate(node.right); return leftRotate(node); } // Return the (unchanged) node pointer return node; } // A utility function to print preorder // traversal of the tree function preOrder(root) { if (root !== null) { console.log(root.key + " "); preOrder(root.left); preOrder(root.right); } } // Driver code let root = null; // Constructing tree given in the above figure root = insert(root, 10); root = insert(root, 20); root = insert(root, 30); root = insert(root, 40); root = insert(root, 50); root = insert(root, 25); /* The constructed AVL Tree would be 30 / \ 20 40 / \ \ 10 25 50 */ console.log("Preorder traversal :"); preOrder(root); OutputPreorder traversal : 30 20 10 25 40 50 Time Complexity: O(log(n)), For InsertionAuxiliary Space: O(Log n) for recursion call stack as we have written a recursive method to insert The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of the AVL insert remains the same as the BST insert which is O(h) where h is the height of the tree. Since the AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn). Comparison with Red Black Tree:The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then the AVL tree should be preferred over Red Black Tree."

Avl tree explained. Avl tree deletion example step by step. Avl tree insertion example step by step. Avl tree example with solution. Explain avl tree with example. Avl tree step by step. Avl tree example. Avl-tree.