Click to verify

Read-only operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications have to observe and restore the height balance of the sub-trees. Searching for a specific key in an AVL tree can be done the same way as that of any balanced or unbalanced binary search tree.[8]: ch. 8 In order for search to work effectively it has to employ a comparison function which establishes a total order (or at least a total preorder) on the set of keys.[9]: 23 The number of comparisons required for successful search is limited by the height h and for unsuccessful search is very close to h, so both are in O(log n).[10]: 216 As a read-only operation the traversal of an AVL tree functions the same way as on any other binary tree. Exploring all n nodes of the tree visits each link exactly twice: one downward visit to enter the subtree rooted by that node, another visit upward to leave that node's subtree after having explored it. Once a node has been found in an AVL tree, the next or previous node can be accessed in amortized constant time.[11]: 58 Some instances of exploring these "nearby" nodes require traversing up to h ∝ log(n) links (particularly when navigating from the rightmost leaf of the root's left subtree to the root or from the root to the leftmost leaf of the root's right subtree; in the AVL tree of figure 1, navigating from node P to the next-to-the-right node Q takes 3 steps). Since there are n−1 links in any tree, the amortized cost is 2×(n−1)/n, or approximately 2. When inserting a node into an AVL tree, you initially follow the same process as inserting into a Binary Search Tree. If the tree is empty, then the node is inserted as the root of the tree. If the tree is not empty, then we go down the root, and recursively go down the tree searching for the location to insert the new node. This traversal is guided by the comparison function. In this case, the node always replaces a NULL reference (left or right) of an external node in the tree i.e., the node is either made a left-child or a right-child of the external node. After this insertion, if a tree becomes unbalanced, only ancestors of the newly inserted node are unbalanced. This is because only those nodes have their sub-trees altered.[12] So it is necessary to check each of the node's ancestors for consistency with the invariants of AVL trees: this is called "retracing". This is achieved by considering the balance factor of each node.[6]: 458–481 [11]: 108 Since with a single insertion the height of an AVL subtree cannot increase by more than one, the temporary balance factor of a node after an insertion will be in the range [–2,+2]. For each node checked, if the temporary balance factor remains in the range from –1 to +1 then only an update of the balance factor and no rotation is necessary. However, if the temporary balance factor is ±2, the subtree rooted at this node is AVL unbalanced, and a rotation is needed.[9]: 52 With insertion as the code below shows, the adequate rotation immediately perfectly rebalances the tree. In figure 1, by inserting the new node Z as a child of node X the height of that subtree Z increases from 0 to 1. Invariant of the retracing loop for an insertion The height of the subtree rooted by Z has increased by 1. It is already in AVL shape. More information Example code for an insert operation ... Example code for an insert operation for (X = parent(Z); X != null; X = parent(Z)) { // Loop (possibly up to the root) // BF(X) has to be updated: if (Z == right_child(X)) { // The right subtree increases if (BF(X) > 0) { // X is right-heavy // ==> the temporary BF(X) == +2 // ==> rebalancing is required. G = parent(X); // Save parent of X around rotations if (BF(Z) < 0) // Right Left Case (see figure 3) N = rotate_RightLeft(X, Z); // Double rotation: Right(Z) then Left(X) else // Right Right Case (see figure 2) N = rotate_Left(X, Z); // Single rotation Left(X) // After rotation adapt parent link } else { if (BF(X) < 0) { BF(X) = 0; // Z's height increase is absorbed at X. break; // Leave the loop } BF(X) = +1; Z = X; // Height(Z) increases by 1 continue; } } else { // Z == left_child(X): the left subtree increases if (BF(X) < 0) { // X is left-heavy // ==> the temporary BF(X) == -2 // ==> rebalancing is required. G = parent(X); // Save parent of X around rotations if (BF(Z) > 0) // Left Right Case N = rotate_LeftRight(X, Z); // Double rotation: Left(Z) then Right(X) else // Left Left Case N = rotate_Right(X, Z); // Single rotation Right(X) // After rotation adapt parent link } else { if (BF(X) > 0) { BF(X) = 0; // Z's height increase is absorbed at X. break; // Leave the loop } BF(X) = -1; Z = X; // Height(Z) increases by 1 continue; } } // After a rotation adapt parent link: // N is the new root of the rotated subtree // Height does not change: Height(N) == old Height(X) parent(N) = G; if (G != null) { if (X == left_child(G)) left_child(G) = N; else right_child(G) = N; } else tree->root = N; // N is the new root of the total tree break; // There is no fall thru, only break; or continue; } // Unless loop is left via break, the height of the total tree increases by 1. Close In order to update the balance factors of all nodes, first observe that all nodes requiring correction lie from child to parent along the path of the inserted leaf. If the above procedure is applied to nodes along this path, starting from the leaf, then every node in the tree will again have a balance factor of −1, 0, or 1. The retracing can stop if the balance factor becomes 0 implying that the height of that subtree remains unchanged. If the balance factor becomes ±1 then the height of the subtree increases by one and the retracing needs to continue. If the balance factor temporarily becomes ±2, this has to be repaired by an appropriate rotation after which the subtree has the same height as before (and its root the balance factor 0). The time required is O(log n) for lookup, plus a maximum of O(log n) retracing levels (O(1) on average) on the way back to the root, so the operation can be completed in O(log n) time.[9]: 53 The preliminary steps for deleting a node are described in section Binary search tree#Deletion. There, the effective deletion of the subject node or the replacement node decreases the height of the corresponding child tree either from 1 to 0 or from 2 to 1, if that node had a child. Starting at this subtree, it is necessary to check each of the ancestors for consistency with the invariants of AVL trees. This is called "retracing". Since with a single deletion the height of an AVL subtree cannot decrease by more than one, the temporary balance factor of a node will be in the range from −2 to +2. If the balance factor remains in the range from −1 to +1 it can be adjusted in accord with the AVL rules. If it becomes ±2 then the subtree is unbalanced and needs to be rotated. (Unlike insertion where a rotation always balances the tree, after delete, there may be BF(Z) ≠ 0 (see figures 2 and 3), so that after the appropriate single or double rotation the height of the rebalanced subtree decreases by one meaning that the tree has to be rebalanced again on the next higher level.) The various cases of rotations are described in section Rebalancing. Invariant of the retracing loop for a deletion The height of the subtree rooted by N has decreased by 1. It is already in AVL shape. More information Example code for a delete operation ... Example code for a delete operation for (X = parent(N); X != null; X = G) { // Loop (possibly up to the root) G = parent(X); // Save parent of X around rotations // BF(X) has not yet been updated! if (N == left_child(X)) { // the left subtree decreases if (BF(X) > 0) { // X is right-heavy // ==> the temporary BF(X) == +2 // ==> rebalancing is required. Z = right_child(X); // Sibling of N (higher by 2) b = BF(Z); if (b < 0) // Right Left Case (see figure 3) N = rotate_RightLeft(X, Z); // Double rotation: Right(Z) then Left(X) else // Right Right Case (see figure 2) N = rotate_Left(X, Z); // Single rotation Left(X) // After rotation adapt parent link } else { if (BF(X) == 0) { BF(X) = +1; // N's height decrease is absorbed at X. break; // Leave the loop } N = X; BF(N) = 0; // Height(N) decreases by 1 continue; } } else { // (N == right_child(X)): The right subtree decreases if (BF(X) < 0) { // X is left-heavy // ==> the temporary BF(X) == -2 // ==> rebalancing is required. Z = left_child(X); // Sibling of N (higher by 2) b = BF(Z); if (b > 0) // Left Right Case N = rotate_LeftRight(X, Z); // Double rotation: Left(Z) then Right(X) else // Left Left Case N = rotate_Right(X, Z); // Single rotation Right(X) // After rotation adapt parent link } else { if (BF(X) == 0) { BF(X) = -1; // N's height decrease is absorbed at X. break; // Leave the loop } N = X; BF(N) = 0; // Height(N) decreases by 1 continue; } } // After a rotation adapt parent link: // N is the new root of the rotated subtree parent(N) = G; if (G != null) { if (X == left_child(G)) left_child(G) = N; else right_child(G) = N; } else tree->root = N; // N is the new root of the total tree if (b == 0) break; // Height does not change: Leave the loop // Height(N) decreases by 1 (== old Height(X)-1) } // If (b != 0) the height of the total tree decreases by 1. Close The retracing can stop if the balance factor becomes ±1 (it must have been 0) meaning that the height of that subtree remains unchanged. If the balance factor becomes 0 (it must have been ±1) then the height of the subtree decreases by one and the retracing needs to continue. If the balance factor temporarily becomes ±2, this has to be repaired by an appropriate rotation. It depends on the balance factor of the sibling Z (the higher child tree in figure 2) whether the height of the subtree decreases by one –and the retracing needs to continue– or does not change (if Z has the balance factor 0) and the whole tree is in AVL-shape. The time required is O(log n) for lookup, plus a maximum of O(log n) retracing levels (O(1) on average) on the way back to the root, so the operation can be completed in O(log n) time. In addition to the single-element insert, delete and lookup operations, several set operations have been defined on AVL trees: union, intersection and set difference. Then fast bulk operations on insertions or deletions can be implemented based on these set functions. These set operations rely on two helper operations, Split and Join. With the new operations, the implementation of AVL trees can be more efficient and highly-parallelizable.[13] The function Join on two AVL trees t1 and t2 and a key k will return a tree containing all elements in t1, t2 as well as k. It requires k to be greater than all keys in t1 and smaller than all keys in t2. If the two trees differ by height at most one, Join simply create a new node with left subtree t1, root k and right subtree t2. Otherwise, suppose that t1 is higher than t2 for more than one (the other case is symmetric). Join follows the right spine of t1 until a node c which is balanced with t2. At this point a new node with left child c, root k and right child t2 is created to replace c. The new node satisfies the AVL invariant, and its height is one greater than c. The increase in height can increase the height of its ancestors, possibly invalidating the AVL invariant of those nodes. This can be fixed either with a double rotation if invalid at the parent or a single left rotation if invalid higher in the tree, in both cases restoring the height for any further ancestor nodes. Join will therefore require at most two rotations. The cost of this function is the difference of the heights between the two input trees. More information Pseudocode implementation for the Join algorithm ... Pseudocode implementation for the Join algorithm function JoinRightAVL(TL, k, TR) (l, k', c) = expose(TL) if (Height(c) node.data: node.right = delete(node.right, data) else: if node.left is None: temp = node.right node = None return temp elif node.right is None: temp = node.left node = None return temp temp = minValueNode(node.right) node.data = temp.data node.right = delete(node.right, temp.data) if node is None: return node # Update the balance factor and balance the tree node.height = 1 + max(getHeight(node.left), getHeight(node.right)) balance = getBalance(node) # Balancing the tree # Left Left if balance > 1 and getBalance(node.left) >= 0: return rightRotate(node) # Left Right if balance > 1 and getBalance(node.left) < 0: node.left = leftRotate(node.left) return rightRotate(node) # Right Right if balance < -1 and getBalance(node.right) 0: node.right = rightRotate(node.right) return leftRotate(node) return node Run Example » Time Complexity for AVL Trees Take a look at the unbalanced Binary Search Tree below. Searching for "M" means that all nodes except 1 must be compared. But searching for "M" in the AVL Tree below only requires us to visit 4 nodes. So in worst case, algorithms like search, insert, and delete must run through the whole height of the tree. This means that keeping the height (\(h \)) of the tree low, like we do using AVL Trees, gives us a lower runtime. B G E K F P I M Binary Search Tree(unbalanced) G E K B F I P M AVL Tree(self-balancing) See the comparison of the time complexities between Binary Search Trees and AVL Trees below, and how the time complexities relate to the height (\(h\)) of the tree, and the number of nodes (\(n\)) in the tree. The BST is not self-balancing. This means that a BST can be very unbalanced, almost like a long chain, where the height is nearly the same as the number of nodes. This makes operations like searching, deleting and inserting nodes slow, with time complexity \(O(h) = O(n)\). The AVL Tree however is self-balancing. That means that the height of the tree is kept to a minimum so that operations like searching, deleting and inserting nodes are much faster, with time complexity \(O(h) = O(\log n)\). \(O(\log n)\) Explained The fact that the time complexity is \(O(h) = O(\log n)\) for search, insert, and delete on an AVL Tree with height \(h\) and nodes \(n\) can be explained like this: Imagine a perfect Binary Tree where all nodes have two child nodes except on the lowest level, like the AVL Tree below. H D B F E G A C L J N M O I K The number of nodes on each level in such an AVL Tree are: \[1, 2, 4, 8, 16, 32, ..\] Which is the same as: \[2^0, 2^1, 2^2, 2^3, 2^4, 2^5, ..\] To get the number of nodes \(n\) in a perfect Binary Tree with height \(h=3\), we can add the number of nodes on each level together: \[n_3=2^0 + 2^1 + 2^2 + 2^3 = 15\] Which is actually the same as: \[n_3=2^4 - 1 = 15\] And this is actually the case for larger trees as well! If we want to get the number of nodes \(n \) in a tree with height \(h=5 \) for example, we find the number of nodes like this: \[n_5=2^6 - 1 = 63\] So in general, the relationship between the height \(h \) of a perfect Binary Tree and the number of nodes in it \(n \), can be expressed like this: \[n_h = 2^{h+1} - 1\] Note: The formula above can also be found by calculating the sum of the geometric series \(2^0 + 2^1 + 2^2+ 2^3 + ... + 2^n \) We know that the time complexity for searching, deleting, or inserting a node in an AVL tree is \(O(h) \), but we want to argue that the time complexity is actually \(O(\log(n)) \), so we need to find the height \(h\) described by the number of nodes \(n\): \[\begin{equation} \begin{aligned} n & = 2^{h+1}-1 \\ n+1 & = 2^{h+1} \\ \log_2(n+1) & = \log_2(2^{h+1}) \\ h & = \log_2(n+1) - 1 \\ \\ O(h) & = O(\log{n}) \end{aligned} \end{equation} \] How the last line above is derived might not be obvious, but for a Binary Tree with a lot of nodes (big \(n\)), the "+1" and "-1" terms are not important when we consider time complexity. For more details on how to calculate the time complexity using Big O notation, see this page. The math above shows that the time complexity for search, delete, and insert operations on an AVL Tree \(O(h) \), can actually be expressed as \(O(\log{n}) \), which is fast, a lot faster than the time complexity for BSTs which is \(O(n) \). DSA Exercises AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.In rotation operation, the positions of the nodes of a subtree are interchanged.In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.In left-right rotation, the arrangements are first shifted to the left and then to the right.In right-left rotation, the arrangements are first shifted to the right and then to the left.A newNode is always inserted as a leaf node with balance factor equal to 0.A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed. # AVL tree implementation in Python import sys # Create a tree node class TreeNode(object): def __init__(self, key): self.key = key self.left = None self.right = None self.height = 1 class AVLTree(object): # Function to insert a node def insert_node(self, root, key): # Find the correct location and insert the node if not root: return TreeNode(key) elif key < root.key: root.left = self.insert_node(root.left, key) else: root.right = self.insert_node(root.right, key) root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right)) # Update the balance factor and balance the tree balanceFactor = self.getBalance(root) if balanceFactor > 1: if key < root.left.key: return self.rightRotate(root) else: root.left = self.leftRotate(root.left) return self.rightRotate(root) if balanceFactor < -1: if key > root.right.key: return self.leftRotate(root) else: root.right = self.rightRotate(root.right) return self.leftRotate(root) return root # Function to delete a node def delete_node(self, root, key): # Find the node to be deleted and remove it if not root: return root elif key < root.key: root.left = self.delete_node(root.left, key) elif key > root.key: root.right = self.delete_node(root.right, key) else: if root.left is None: temp = root.right root = None return temp elif root.right is None: temp = root.left root = None return temp temp = self.getMinValueNode(root.right) root.key = temp.key root.right = self.delete_node(root.right, temp.key) if root is None: return root # Update the balance factor of nodes root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right)) balanceFactor = self.getBalance(root) # Balance the tree if balanceFactor > 1: if self.getBalance(root.left) >= 0: return self.rightRotate(root) else: root.left = self.leftRotate(root.left) return self.rightRotate(root) if balanceFactor < -1: if self.getBalance(root.right) 0) { Node x = y.left; Node T2 = x.right; x.right = y; y.left = T2; y.height = max(height(y.left), height(y.right)) + 1; x.height = max(height(x.left), height(x.right)) + 1; return x; } Node leftRotate(Node x) { Node y = x.right; Node T2 = y.left; y.left = x; x.right = T2; x.height = max(height(x.left), height(x.right)) + 1; y.height = max(height(y.left), height(y.right)) + 1; return y; } // Get balance factor of a node int getBalanceFactor(Node N) { if (N == null) return 0; return height(N.left) - height(N.right); } // Insert a node Node insertNode(Node node, int item) { // Find the position and insert the node if (node == null) return (new Node(item)); if (item < node.item) node.left = insertNode(node.left, item); else if (item > node.item) node.right = insertNode(node.right, item); else return node; // Update the balance factor of each node // And, balance the tree node.height = 1 + max(height(node.left), height(node.right)); int balanceFactor = getBalanceFactor(node); if (balanceFactor > 1) { if (item < node.left.item) { return rightRotate(node); } else if (item > node.left.item) { node.left = leftRotate(node.left); return rightRotate(node); } } if (balanceFactor < -1) { if (item > node.right.item) { return leftRotate(node); } else if (item < node.right.item) { node.right = rightRotate(node.right); return leftRotate(node); } } return node; } Node nodeWithMimumValue(Node node) { Node current = node; while (current.left != null) current = current.left; return current; } // Delete a node Node deleteNode(Node root, int item) { // Find the node to be deleted and remove it if (root == null) return root; if (item < root.item) root.left = deleteNode(root.left, item); else if (item > root.item) root.right = deleteNode(root.right, item); else { if ((root.left == null) || (root.right == null)) { Node temp = null; if (temp == root.left) temp = root.right; else temp = root.left; if (temp == null) { temp = root; root = null; } else root = temp; } else { Node temp = nodeWithMimumValue(root.right); root.item = temp.item; root.right = deleteNode(root.right, temp.item); } } if (root == null) return root; // Update the balance factor of each node and balance the tree root.height = max(height(root.left), height(root.right)) + 1; int balanceFactor = getBalanceFactor(root); if (balanceFactor > 1) { if (getBalanceFactor(root.left) >= 0) { return rightRotate(root); } else { root.left = leftRotate(root.left); return rightRotate(root); } } if (balanceFactor < -1) { if (getBalanceFactor(root.right) height; } int max(int a, int b) { return (a > b) ? a : b; } // Create a node 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; return (node); } // Right rotate struct Node *rightRotate(struct Node *y) { struct Node *x = y->left; struct Node *T2 = x->right; x->right = y; y->left = T2; y->height = max(height(y->left), height(y->right)) + 1; x->height = max(height(x->left), height(x->right)) + 1; return x; } // Left rotate struct Node *leftRotate(struct Node *x) { struct Node *y = x->right; struct Node *T2 = y->left; y->left = x; x->right = T2; x->height = max(height(x->left), height(x->right)) + 1; y->height = max(height(y->left), height(y->right)) + 1; return y; } // Get the balance factor int getBalance(struct Node *N) { if (N == NULL) return 0; return height(N->left) - height(N->right); } // Insert node struct Node *insertNode(struct Node *node, int key) { // Find the correct position to insertNode the node and insertNode it if (node == NULL) return (newNode(key)); if (key < node->key) node->left = insertNode(node->left, key); else if (key > node->key) node->right = insertNode(node->right, key); else return node; // Update the balance factor of each node and // Balance the tree 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; } // Delete a node struct Node *deleteNode(struct Node *root, int key) { // Find the node and delete it if (root == NULL) return root; if (key < root->key) root->left = deleteNode(root->left, key); else if (key > root->key) root->right = deleteNode(root->right, key); else { if ((root->left == NULL) || (root->right == NULL)) { struct Node *temp = root->left ? root->left : root->right; if (temp == NULL) { temp = root; root = NULL; } else *root = *temp; free(temp); } else { struct Node *temp = minValueNode(root->right); root->key = temp->key; root->right = deleteNode(root->right, temp->key); } } if (root == NULL) return root; // Update the balance factor of each node and // balance the tree root->height = 1 + max(height(root->left), height(root->right)); int balance = getBalance(root); if (balance > 1 && getBalance(root->left) >= 0) return rightRotate(root); if (balance > 1 && getBalance(root->left) < 0) { root->left = leftRotate(root->left); return rightRotate(root); } if (balance < -1 && getBalance(root->right) right) > 0) { root->right = rightRotate(root->right); return leftRotate(root); } return root; } // Print the tree void printPreOrder(struct Node *root) { if (root != NULL) { printf("%d ", root->key); printPreOrder(root->left); printPreOrder(root->right); } } int main() { struct Node *root = NULL; root = insertNode(root, 2); root = insertNode(root, 1); root = insertNode(root, 7); root = insertNode(root, 4); root = insertNode(root, 5); root = insertNode(root, 3); root = insertNode(root, 8); printPreOrder(root); root = deleteNode(root, 3); printf("After deletion: "); printPreOrder(root); return 0; } // AVL tree implementation in C++ #include using namespace std; class Node { public: int key; Node *left; Node *right; int height; }; int max(int a, int b); // Calculate height int height(Node *N) { if (N == NULL) return 0; return N->height; } int max(int a, int b) { return (a > b) ? a : b; } // New node creation Node *newNode(int key) { Node *node = new Node(); node->key = key; node->left = NULL; node->right = NULL; node->height = 1; return (node); } // Rotate right Node *rightRotate(Node *y) { Node *x = y->left; Node *T2 = x->right; x->right = y; y->left = T2; y->height = max(height(y->left), height(y->right)) + 1; x->height = max(height(x->left), height(x->right)) + 1; return x; } // Rotate left Node *leftRotate(Node *x) { Node *y = x->right; Node *T2 = y->left; y->left = x; x->right = T2; x->height = max(height(x->left), height(x->right)) + 1; y->height = max(height(y->left), height(y->right)) + 1; return y; } // Get the balance factor of each node int getBalanceFactor(Node *N) { if (N == NULL) return 0; return height(N->left) - height(N->right); } // Insert a node Node *insertNode(Node *node, int key) { // Find the correct postion and insert the node if (node == NULL) return (newNode(key)); if (key < node->key) node->left = insertNode(node->left, key); else if (key > node->key) node->right = insertNode(node->right, key); else return node; // Update the balance factor of each node and // balance the tree node->height = 1 + max(height(node->left), height(node->right)); int balanceFactor = getBalanceFactor(node); if (balanceFactor > 1) { if (key < node->left->key) { return rightRotate(node); } else if (key > node->left->key) { node->left = leftRotate(node->left); return rightRotate(node); } } if (balanceFactor < -1) { if (key > node->right->key) { return leftRotate(node); } else if (key < node->right->key) { node->right = rightRotate(node->right); return leftRotate(node); } } return node; } // Node with minimum value Node *nodeWithMimumValue(Node *node) { Node *current = node; while (current->left != NULL) current = current->left; return current; } // Delete a node Node *deleteNode(Node *root, int key) { // Find the node and delete it if (root == NULL) return root; if (key < root->key) root->left = deleteNode(root->left, key); else if (key > root->key) root->right = deleteNode(root->right, key); else { if ((root->left == NULL) || (root->right == NULL)) { Node *temp = root->left ? root->left : root->right; if (temp == NULL) { temp = root; root = NULL; } else *root = *temp; free(temp); } else { Node *temp = nodeWithMimumValue(root->right); root->key = temp->key; root->right = deleteNode(root->right, temp->key); } } if (root == NULL) return root; // Update the balance factor of each node and // balance the tree root->height = 1 + max(height(root->left), height(root->right)); int balanceFactor = getBalanceFactor(root); if (balanceFactor > 1) { if (getBalanceFactor(root->left) >= 0) { return rightRotate(root); } else { root->left = leftRotate(root->left); return rightRotate(root); } } if (balanceFactor < -1) { if (getBalanceFactor(root->right) right = rightRotate(root->right); return leftRotate(root); } } return root; } // Print the tree void printTree(Node *root, string indent, bool last) { if (root != nullptr) { cout