AVL Tree In the early 60's G.M. Adelson-Velsky and E.M. Landis invented the first selfbalancing binary search tree data structure, calling it AVL Tree. An AVL tree is a binary search tree (BST, defined in §3) with a self-balancing condition stating that the difference between the height of the left and right subtrees cannot be no more than one, see Figure 7.1. This condition, restored after each tree modification, forces the general shape of an AVL tree. Before continuing, let us focus on why balance is so important. Consider a binary search tree obtained by starting with an empty tree and inserting some values in the following order 1,2,3,4,5. The BST in Figure 7.2 represents the worst case scenario in which the running time of all common operations such as search, insertion and deletion are O(n). By applying a balance condition we ensure that the worst case running time of each common operation is O(log n). The height of an AVL tree with n nodes is O(log n) regardless of the order in which values are inserted. The AVL balance condition, known also as the node balance factor represents an additional piece of information stored for each node. This is combined with a technique that efficiently restores the balance condition for the tree. In an AVL tree the inventors make use of a well-known technique called tree rotation. h h+1 Figure 7.1: The left and right subtrees of an AVL tree differ in height by at most 1 54 CHAPTER 7. AVL TREE 55 1 2 3 4 5 Figure 7.2: Unbalanced binary search tree 2 4 5 1 3 4 5 3 2 1 a) b) Figure 7.3: Avl trees, insertion order: -a)1,2,3,4,5 -b)1,5,4,3,2 CHAPTER 7. AVL TREE 56 7.1 Tree Rotations A tree rotation is a constant time operation on a binary search tree that changes the shape of a tree while preserving standard BST properties. There are left and right rotations both of them decrease the height of a BST by moving smaller subtrees down and larger subtrees up. 14 24 11 8 2 8 14 24 2 11 Right Rotation Left Rotation Figure 7.4: Tree left and right rotations CHAPTER 7. AVL TREE 57 1) algorithm LeftRotation(node) 2) Pre: node.Right ! = ∅ 3) Post: node.Right is the new root of the subtree, 4) node has become node.Right's left child and, 5) BST properties are preserved 6) RightNode ← node.Right 7) node.Right ← RightNode.Left 8) RightNode.Left ← node 9) end LeftRotation 1) algorithm RightRotation(node) 2) Pre: node.Left ! = ∅ 3) Post: node.Left is the new root of the subtree, 4) node has become node.Left's right child and, 5) BST properties are preserved 6) Lef tNode ← node.Left 7) node.Left ← Lef tNode.Right 8) Lef tNode.Right ← node 9) end RightRotation The right and left rotation algorithms are symmetric. Only pointers are changed by a rotation resulting in an O(1) runtime complexity; the other fields present in the nodes are not changed. 7.2 Tree Rebalancing The algorithm that we present in this section verifies that the left and right subtrees differ at most in height by 1. If this property is not present then we perform the correct rotation. Notice that we use two new algorithms that represent double rotations. These algorithms are named LeftAndRightRotation, and RightAndLeftRotation. The algorithms are self documenting in their names, e.g. LeftAndRightRotation first performs a left rotation and then subsequently a right rotation. CHAPTER 7. AVL TREE 58 1) algorithm CheckBalance(current) 2) Pre: current is the node to start from balancing 3) Post: current height has been updated while tree balance is if needed 4) restored through rotations 5) if current.Left = ∅ and current.Right = ∅ 6) current.Height = -1; 7) else 8) current.Height = Max(Height(current.Left),Height(current.Right)) + 1 9) end if 10) if Height(current.Left) - Height(current.Right) > 1 11) if Height(current.Left.Left) - Height(current.Left.Right) > 0 12) RightRotation(current) 13) else 14) LeftAndRightRotation(current) 15) end if 16) else if Height(current.Left) - Height(current.Right) < −1 17) if Height(current.Right.Left) - Height(current.Right.Right) < 0 18) LeftRotation(current) 19) else 20)

RightAndLeftRotation(current) 21) end if 22) end if 23) end CheckBalance 7.3 Insertion AVL insertion operates first by inserting the given value the same way as BST insertion and then by applying rebalancing techniques if necessary. The latter is only performed if the AVL property no longer holds, that is the left and right subtrees height differ by more than 1. Each time we insert a node into an AVL tree: 1. We go down the tree to find the correct point at which to insert the node, in the same manner as for BST insertion; then 2. we travel up the tree from the inserted node and check that the node balancing property has not been violated; if the property hasn't been violated then we need not rebalance the tree, the opposite is true if the balancing property has been violated. CHAPTER 7. AVL TREE 59 1) algorithm Insert(value) 2) Pre: value has passed custom type checks for type T 3) Post: value has been placed in the correct location in the tree 4) if root = ∅ 5) root ← node(value) 6) else 7) InsertNode(root, value) 8) end if 9) end Insert 1) algorithm InsertNode(current, value) 2) Pre: current is the node to start from 3) Post: value has been placed in the correct location in the tree while 4) preserving tree balance 5) if value < current.Value 6) if current.Left = ∅ 7) current.Left ← node(value) 8) else 9) InsertNode(current.Left, value) 10) end if 11) else 12) if current.Right = ∅ 13) current.Right ← node(value) 14) else 15) InsertNode(current.Right, value) 16) end if 17) end if 18) CheckBalance(current) 19) end InsertNode 7.4 Deletion Our balancing algorithm is like the one presented for our BST (defined in §3.3). The major difference is that we have to ensure that the tree still adheres to the AVL balance property after the removal of the node. If the tree doesn't need to be rebalanced and the value we are removing is contained within the tree then no further step are required. However, when the value is in the tree and its removal upsets the AVL balance property then we must perform the correct rotation(s). CHAPTER 7. AVL TREE 60 1) algorithm Remove(value) 2) Pre: value is the value of the node to remove, root is the root node 3) of the Avl 4) Post: node with value is removed and tree rebalanced if found in which 5) case yields true, otherwise false 6) nodeT oRemove ← root 7) parent ← ∅ 8) Stackpath ← root 9) while nodeT oRemove 6= ∅ and nodeT oRemove.V alue = V alue 10) parent = nodeT oRemove 11) if value < nodeT oRemove.Value 12) nodeT oRemove ← nodeToRemove.Left 13) else 14) nodeT oRemove ← nodeToRemove.Right 15) end if 16) path.Push(nodeToRemove) 17) end while 18) if nodeT oRemove = ∅ 19) return false // value not in Avl 20) end if 21) parent ← FindParent(value) 22) if count = 1 // count keeps track of the # of nodes in the Avl 23) root ← ∅ // we are removing the only node in the Avl 24) else if nodeT oRemove.Left = ∅ and nodeT oRemove.Right = null 25) // case #1 26) if nodeT oRemove.Value < parent.Value 27) parent.Left ← ∅ 28) else 29) parent.Right ← ∅ 30) end if 31) else if nodeT oRemove.Left = ∅ and nodeT oRemove.Right 6= ∅ 32) // case # 2 33) if nodeT oRemove.Value < parent.Value 34) parent.Left ← nodeT oRemove.Right 35) else 36) parent.Right ← nodeT oRemove.Right 37) end if 38) else if nodeT oRemove.Left 6= ∅ and nodeT oRemove.Right = ∅ 39) // case #3 40) if nodeT oRemove.Value < parent.Value 41) parent.Left ← nodeT oRemove.Left 42) else 43) parent.Right ← nodeT oRemove.Left 44) end if 45) else 46) // case #4 47) largestV alue ← nodeT oRemove.Left 48) while largestV alue.Right 6= ∅ 49) // find the largest value in the left subtree of nodeT oRemove 50) largestV alue ← largestV alue.Right CHAPTER 7. AVL TREE 61 51) end while 52) // set the parents' Right pointer of largestV alue to ∅ 53) FindParent(largestV alue.Value).Right ← ∅ 54) nodeT oRemove.Value ← largestV alue.Value 55) end if 56) while path.Count > 0 57) CheckBalance(path.Pop()) // we trackback to the root node check balance 58) end while 59) count ← count − 1 60) return true 61) end Remove