CSCI 5454 Final Project: AVL Tree

Robert Werthman

Contents

Introduction **3**
What is an AVL Tree? 3
What problems does it solve? 3
Where is it used? 3

Mathemetical Analysis of Correctness, Runtime, and Space **3**
Correctness 3
Basecase 4
Inductive Proof of the First Invariant 5
Inductive Proof of the Second Invariant 5
Runtime 6
Space 8

Numerical Characterization of Runtime and Space **8**
Description of the code invloved in the Numerical Characterization 9
Characterization of Runtime 9
Runtime of Search 10
Runtime of Insert 10
Runtime of Delete 10
Characterization of Space Usage 11

Extensions, Improvements, and Recent Work **11**

References **13**

Introduction

What is an AVL Tree?

An AVL tree is a binary search tree that is “self-balancing”. This means after each operation, like an insertion or deletion, on the tree the heights of each node’s children differ by at most 1. The height of a node is the number of nodes in the longest path from it to a leaf node. A leaf node always has a height of 0. Its parent node, if the leaf node was the parent’s only child, would have a height of 1. An AVL tree is “self-balancing” because after each operation on the tree the heights and balance of the nodes are readjusted by the tree, itself [1].

What problems does it solve?

AVL trees, like binary trees, are used for storing and retrieving information. Their advantage is that they can perform these operations faster than if the information was stored in an array. As will be shown later in this paper, storing and retrieving takes $\log n$ time for an AVL tree while performing the same operations on an array could take up to n time.

Where is it used?

Red-black trees, another kind of self-balancing binary search tree, are typically used instead of AVL trees in real world applications [2]. Red-black trees can be found in the C++ Standard Library (std) as the underlying data structure for the std::map and std::set containers and it is reasonable to think that AVL trees could be used instead.

Mathemetical Analysis of Correctness, Runtime, and Space

Correctness

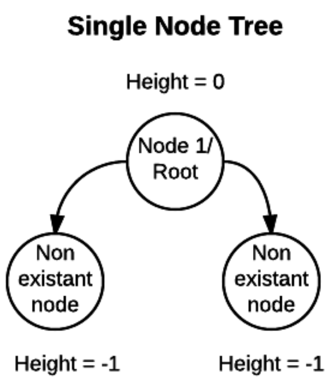
The correctness of the AVL tree relies on maintaining two invariants

1. The height of a node is the number of nodes in the longest path from it to a leaf node.

2. The heights of each node's children differ by at most 1. This is demonstrated by the equation $|x - y| \leq 1$ where x is the height of the left child node and y is the height of the right child node.

Basecase

Figure 1: An AVL tree with a single node



We will use the insertion operation to show these invariants are maintained. We first start with a single, leaf node in a tree as the base case as shown in Figure 1. First, we let the height of a non existent node be equal to -1. This means a single node in a tree has two non existent children nodes each with a height of -1. To find the height of the single node in the tree we take the max of the heights of the children and then add 1 to it to always maintain the first invariant. In this case max(-1, -1) = -1 and adding 1 to this result gives a height of 0 for the single node in the tree. To make sure the second invariant is maintained we use the given equation

$$|x - y| \leq 1$$

In the case of a single, leaf node the height of the left child is -1 and the height of the right child is -1. Substituting these values in for x and y produces the equation

$$\begin{aligned} |(-1) - (-1)| &= 0 \\ &\leq 1 \end{aligned}$$

Since $0 \leq 1$ the second invariant is maintained.

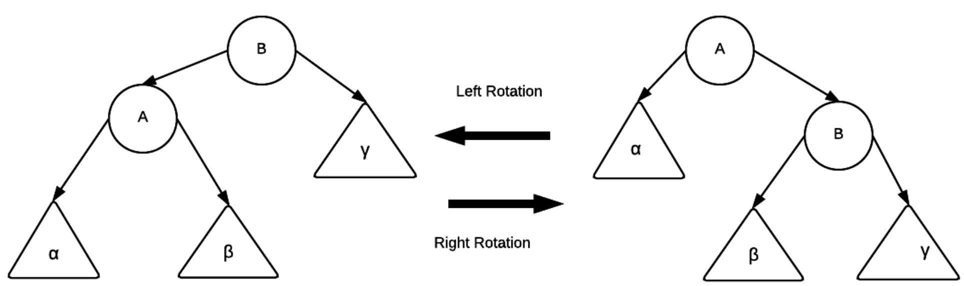
Inductive Proof of the First Invariant

Next it must be inductively shown that the two invariants hold for every AVL tree with size greater than 1 node. Concerning the first invariant, every time a new node is inserted into the AVL tree, it will be a leaf node with a height of 0 as was demonstrated in the base case. Once an insertion takes place, the AVL tree updates the the balances and heights of the new node and all of the nodes above it. This is done by recursing through the parent nodes starting with the inserted node and moving upwards to the root. The height of each parent node is equal to the max of the childrens' heights plus 1. Since this update of the heights takes place for every node inserted into the tree, the first invariant holds for any tree of size greater than 1 nodes.

Inductive Proof of the Second Invariant

To show the correctness of the second invariant we have to look at the possible cases of a tree being out of balance and how those are corrected. In

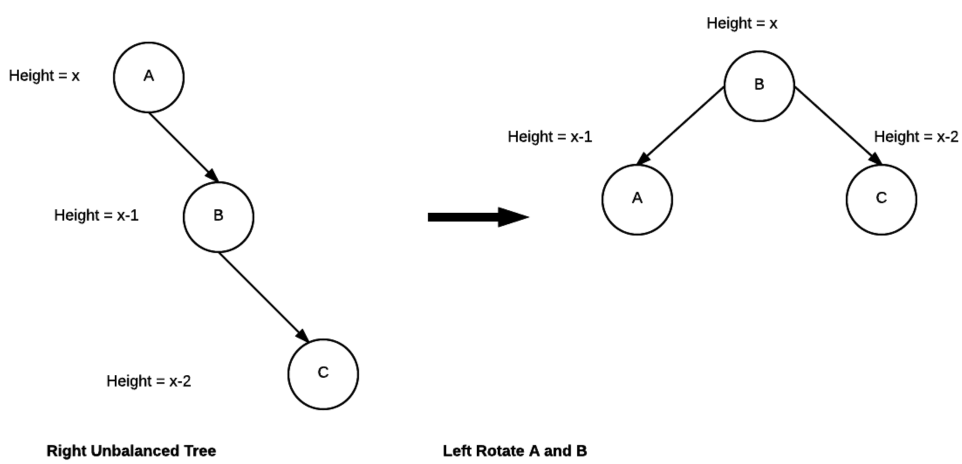
Figure 2: Left and Right Tree Rotations



order for an AVL tree to remain in balance, the tree has to perform what are known as tree rotations after nodes are inserted. [1]. The basic two rotations can be seen in Figure 2. During tree rotations, the shape of a tree is changed in order to change the heights of subtrees at a particular node. For example, doing a right rotation of the tree in Figure 2 "increases" the height of node A while "decreasing" the height of node B [3]. These can be judiciously used to rebalance the tree and maintain the second invariant of an AVL tree.

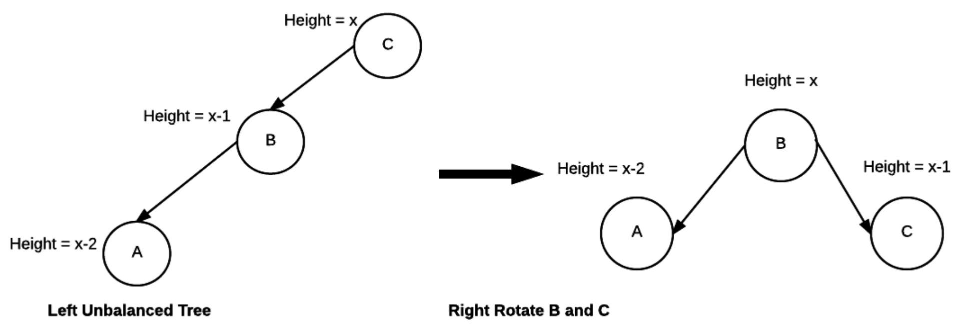
When a node is inserted, the first thing that the tree does is check the balance at the inserted node and all of the nodes above to ensure that the equation for the second invariant is maintained. If the balances of any of the nodes do not satsify the second invariant equation, tree rotations must take place in order to balance the tree. Four cases of an unbalanced tree may occur after a node is inserted into a tree, as shown in Figures 3-6, that require one or more rotations to rebalance a tree [4]. Once one or more rotations of the sub-

Figure 3: Right Unbalanced Tree



trees of a node take place and that node is balanced according to the second invariant equation, the height of that node and any of the other nodes that participated in the rotation are updated. The tree then recursively checks the node's parents, grandparents, etc. until the root of the tree is reached for correct balances, performing any rotations if necessary, and updating the heights as it goes. This method of updating the balances and heights ensures that given any number of inserted nodes, the tree with always be rebalanced

Figure 4: Left Unbalanced Tree



and each node will have the correct height.

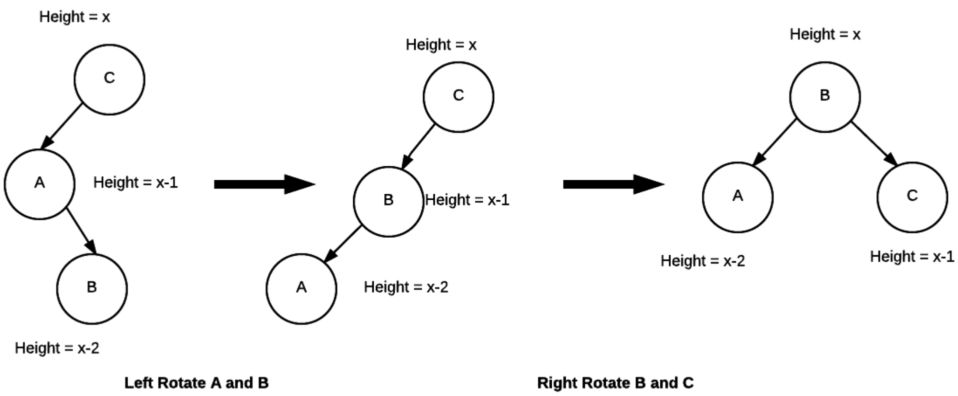
Runtime

Since an AVL tree is a binary search tree any AVL tree operations' runtime depends on the trees height, which is the longest path from the root node to a leaf – also known as its longest branch [5]. In order to prove the runtime any AVL tree operations is O(log n) we need to show that the height of an AVL tree can be bounded by O(log n).

Let the root of the AVL tree have a height of h. One child of the root will have a height of $h-1$ and the other, if we follow the property of an AVL tree that says the heights of the children differ by at most 1, will have a minimum height of $h-2$. Let n_h be the number of nodes in the tree including the root. Let n_{h-1} be the number of nodes underneath and including the child of the root with height $h-1$ and n_{h-2} be the number of nodes underneath and including the child with height $h-2$. We can construct the total size of the tree n_h by using the equation

$$n_h = n_{h-1} + n_{h-2} + 1$$

Figure 5: Right Left Unbalanced Tree



We can bound the height of the AVL tree by reducing the equation with the following setps:

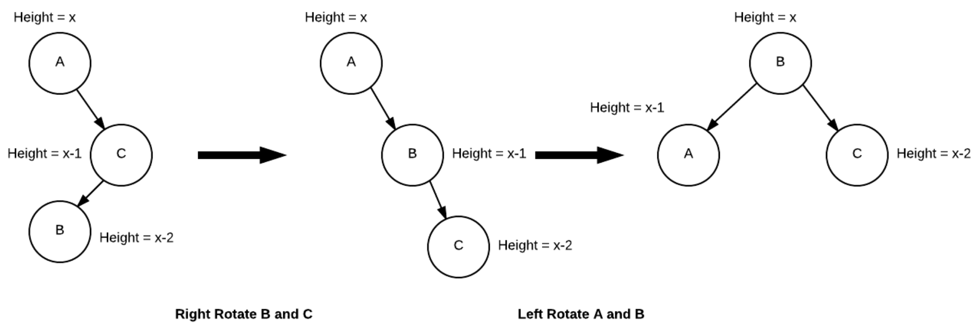
$$
\begin{aligned}
n_h &= n_{h-1} + n_{h-2} + 1 && (1)\\
n_{h-1} &= n_{h-2} + n_{h-3} + 1 && (2)\\
n_h &= (n_{h-2} + n_{h-3} + 1) + n_{h-2} + 1 && (3)\\
n_h &> 2n_{h-2} > 2(2n_{h-4}) > 2(2(2n_{h-6})) > \ldots > 2^{\frac{h}{2}} && (4)\\
\log\, n_h &> \log\, 2^{\frac{h}{2}} && (5)\\
\log\, n_h &> \frac{h}{2} && (6)\\
2\log\, n_h &> h && (7)
\end{aligned}
$$

This shows that the height h of an AVL tree will always be less than O(log n). This means the runtime for any AVL tree operation will take O(log n).

Space

The space an AVL uses is equal the size of the nodes inserted. If there are $n + 1$ nodes inserted in the tree, the space used will be $n + 1$. If there are $n - 1$ nodes removed from the tree, the space used will be $n - 1$.

Figure 6: Left Right Unbalanced Tree



Numerical Characterization of Runtime and Space

Description of the code invloved in the Numerical Characterization

To show the numerical characterization of the space and time performance of an AVL tree, I created a randomized input generator to show the runtime and space of the three operations that can be performed on the AVL tree: insertion, deletion, and search. I ran 12 iterations for each of these tree operations, varying the number of nodes n in the trees by a factor of 2. I ran each of these iterations 50 times each to find the average number of operations given a value of n. n takes on all of the values in the set $\{2^4,\, 2^5, \ldots, 2^{16}\}$ for each tree operation. The keys for each of the nodes in the n node tree were randomly chosen from the set $\{0, \,, 1, \ldots, 2^n$ and then inserted into the tree to create a tree of size n.

Once the tree was generated for an iteration, I then chose a random key from the set of keys that were already in the tree, and I either searched for it, inserted it, or deleted it. To keep track of the runtime when performing these operations on the tree, I kept a global variable as a counter and incremented it every time an atomic operation occurred. To keep track of the space used when performing these operations on the tree, I kept a global variable as a counter and incremented it when a node was inserted and decremented it when a node was removed from the tree. These global variables were reset after each iteration.

Once all the iterations were complete for a specific operation, I took the n for each iteration and the values of the global variables for each iteration and graphed them as a function $f(x)$. The values of n were placed on the x-axis and the values of the globabl variables were placed on the y-axis. The graphs use the logarithmic scale instead of the linear scale because the logarthmic scale more clearly shows what n and the values of the global variables are doing. I then graphed one other line representing the function $g(x)$, which is the function that bounds $f(x)$. $g(x)$ is then multiplied by a constant, c_2, producing a separate line.

Characterization of Runtime

Runtime of Search

Figure 7: Graph of the runtime for searching for a key in different-sized AVL trees

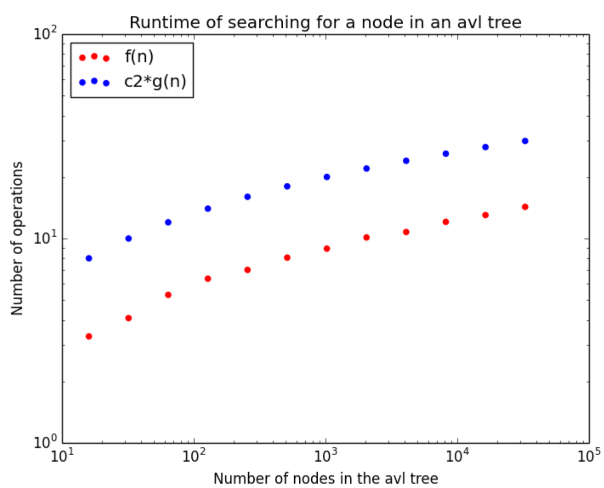


Figure 7 shows the runtime for searching for a key in different-sized avl trees. The function that bounds the runtime of search is $g(n) = \log n$ multiplied by the constant $c_2 = 2$. As can be seen in the graph the runtime of search is O(log n), with the function log n bounding the search runtime from above.

This means given an AVL tree of size n, it will take O(log n) operations to search for key in the tree.

Runtime of Insert

Figure 8: Graph of the runtime for inserting a key into different-sized AVL trees

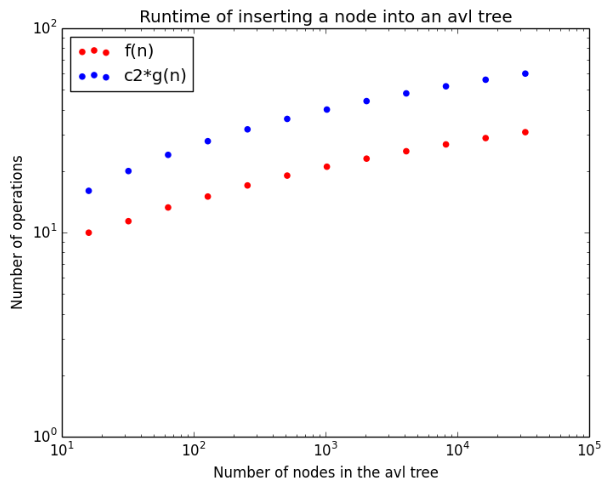
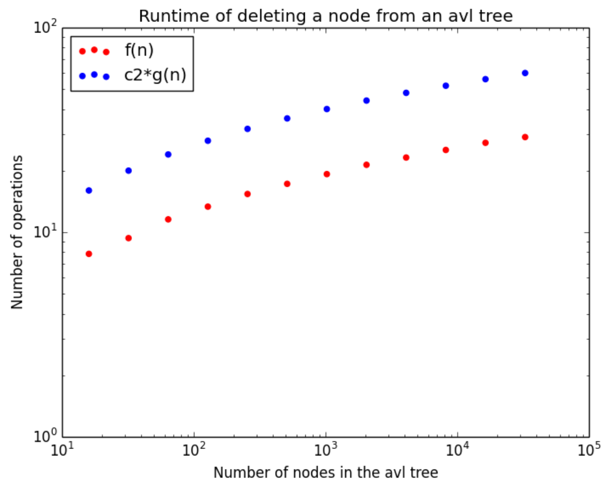


Figure 8 shows the runtime for inserting a key into different-sized AVL trees. The function that bounds the runtime of insert is $g(n) = \log n$ multiplied by the constant $c_2 = 4$. As can be seen in the graph the runtime of insert is O(log n), with the function log n bounding the insert runtime from above. This means given an AVL tree of size n, it will take O(log n) operations to insert a key into the tree.

Runtime of Delete

Figure 9 shows the runtime of deleting a key from different-sized AVL trees. The function that bounds the runtime of delete is $g(n) = \log n$ multiplied by the constant $c_2 = 4$. As can be seen in the graph the runtime of delete is O(log n), with the function log n bounding the delete runtime from above. This means given an AVL tree of size n, it will take O(log n) operations to delete a key from the tree.

Figure 9: Graph of the runtime for deleting a key from different-sized AVL trees

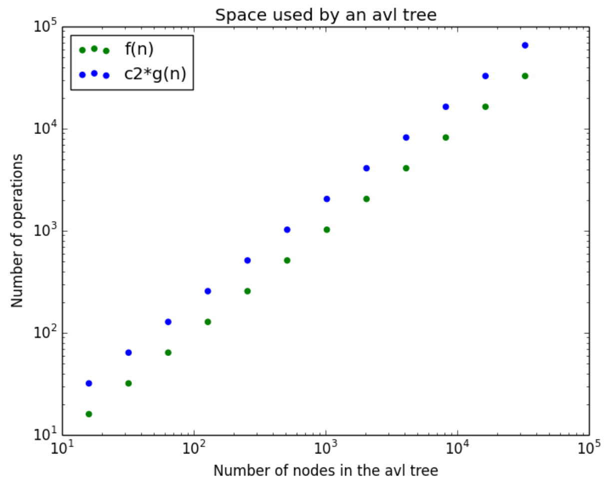


Characterization of Space Usage

Figure 10 shows the space used by different-size AVL trees. The functions that bounds the space used is $g(n) = n$ multiplied by the constant $c_2 = 2$. As can be seen in the graph the space that is used by an AVL tree is $\mathrm{O}(n)$, with the function n bounding the space used from above. This means that given an AVL tree of size n, there will be exaclty n nodes in the tree. With a tree of size n, if a node is inserted there will be $n + 1$ nodes in the tree. With a tree of size n, if a node is deleted there will be $n - 1$ nodes in the tree.

Extensions, Improvements, and Recent Work

Figure 10: Graph of the space used by different-size AVL trees



References

[1] Wikipedia. Avl tree — wikipedia, the free encyclopedia, 2016. [Online; accessed 11-April-2016].

[2] Wikipedia. Redblack tree — wikipedia, the free encyclopedia, 2016. [Online; accessed 18-April-2016].

[3] Wikipedia. Tree rotation — wikipedia, the free encyclopedia, 2015. [Online; accessed 25-April-2016].

[4] TutorialsPoint. Data Structure - Avl Trees, 2016. [Online; accessed 24-April-2016].

[5] MIT. 6.006 Intro to Algorithms, Recitation 04, 2011. [Online; accessed 26-April-2016].