AVL Tree Implementation

Abstract

This document details the implementation of an AVL tree data structure and provides an analysis of its performance relative to theoretical values. In particular, methods for building and sorting an AVL along with methods for inserting and removing elements have their performance measured experimentally and are then evaluated against the expected time complexity using Big-O notation. The space complexity of this data structure is also discussed.

Background

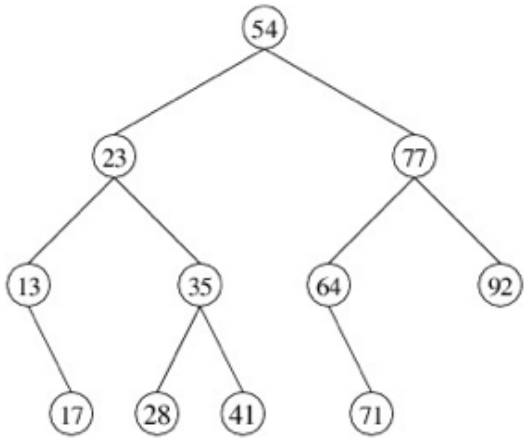
An AVL tree is a binary search tree which obeys the AVL property after the completion of each operation. It allows for search, insertion, and deletion in O(log(n)) time. It allows for sorting in O(n*log(n)) time.

Binary Search Tree- A tree data structure that each node has no more than two children. For any given node, its right child and all of its descendants are either equal to or greater than the node's value. The nodes right child and all of its descendants are less than the node's value.

AVL Property- A requirement such that the height of a tree rooted at any node's right child differs from the height of the tree rooted at the same node's left child by no more than 1

Balance Factor- The height difference between the tree rooted at a node's left child and the tree rooted at the node's right child.

The diagram below is an example of a binary search tree:



For this project, we will define leaves to have height 1. The height of any ancestor nodes is defined to be the number of nodes along the path of maximal length to a leaf node. For example:
-The height of root node with value 54 is 4.
-The height of the node with value 77 is 3.
-The balance factor of node with value 77 is negative 1.
-The balance factor of node with value 23 is 0.

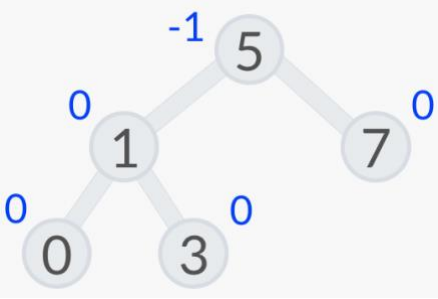
Rotations

Note that the properties of a Binary Search tree as well as the AVL property are both satisfied by this example. A true AVL tree, however, must maintain the AVL property after every insertion, deletion, or other operation.

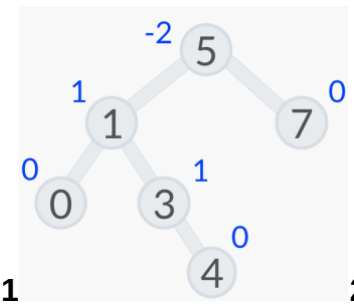
In order maintain the AVL property after various operations, a correction must be made to nodes whose balance factor magnitude exceeds 1 after an operation has occurred. Below is an example of a scenario in which an insertion has led to a violation of the AVL property which is corrected.

The method used to correct forbidden balance factors is called a rotation. There can be left or right rotations. Depending on the situation, double rotations may be needed. There may also be a need to re-assign inner nodes to new parents. Below is an example where both of these are required. However, further documentation should be consulted if the reader is unfamiliar with rotations in AVL trees.

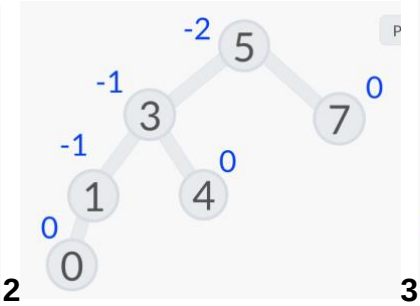
First we have an AVL tree in which all requirements are satisfied. Here, the blue colored numbers represent balance factors.



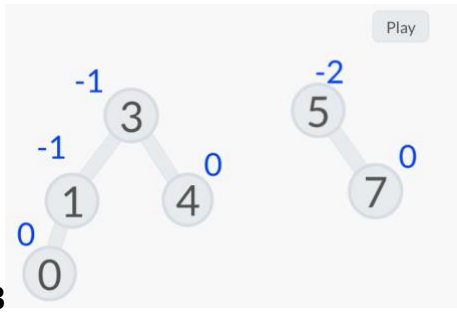
We then perform an insertion of a node with value 4 and use the rotate operation to recover the AVL property.



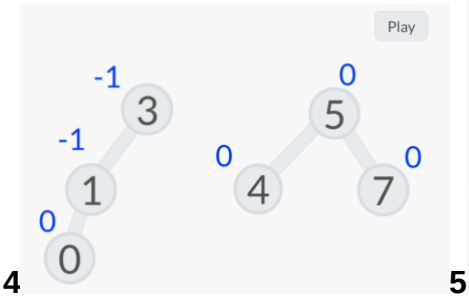
1



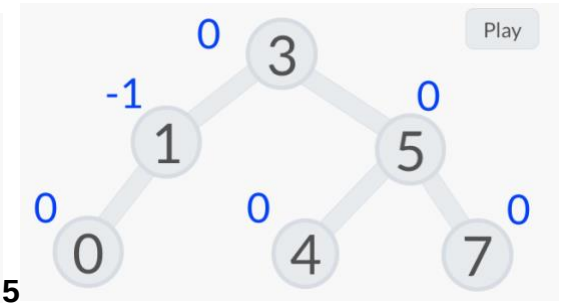
2



3



4



5

Diagram 1- Node with value 4 is inserted. We see that this leads to a forbidden balance factor of negative 2 at the root.

Diagram 2- We want to perform a rotation with the root node as our pivot node to restore the AVL property. In particular, we want to perform a *right rotation* since the balance factor is negative (when the forbidden balance factor is positive, we want to perform a left rotation).

However, in order to perform a rotation, the pivot node and its child on the heavier side of tree rooted at the sub-node must have balance factors of the same sign. If the balance factor of the pivot node and its heavy-side child are opposites, then an *opposite* rotation must first be performed with the heavy-side child as the pivot. Therefore, the transition from diagram 1 to diagram 2 include a *left rotation* with the node of value 1 as the pivot node. Notice that this transition resembles a physical rotation in the counter clockwise direction at the node of value 1 at the axis.

Diagram 3 & 4- We now have the main pivot node and its heavy-side child with balance factors of the same sign. However, before we perform a right rotation about the node with value 5, we must reassign the parent of inner node 4. Any Inner nodes must be re-assigned parents when performing rotations in order to maintain the AVL tree requirements.

Diagram 5- The main pivot node (5) is re-assigned as the right child of its former heavy-side child (3) while node 3 is promoted to take the spot of node 5.

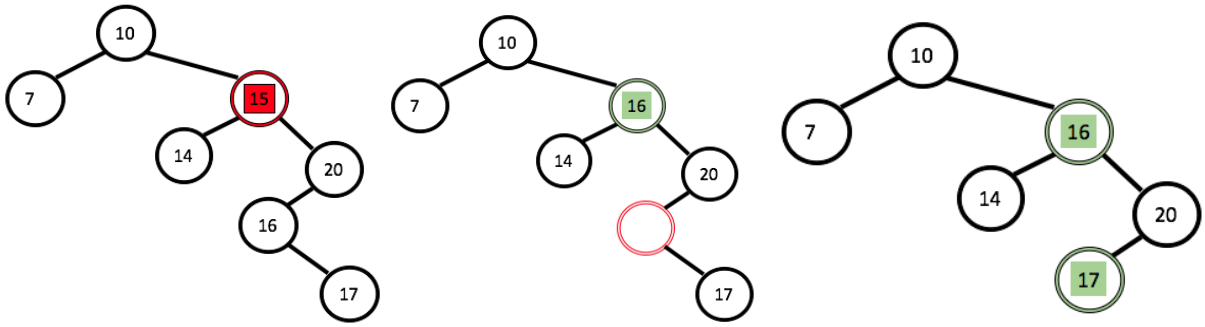
Deletions

Depending on the situation, different actions are required when deleting a node. The three possible cases are described below along with the required actions.

- Case I: Node had no children- Here we just delete the node since it's a leaf.
- Case 2: Node has 1 child- Here we delete the node and promote its one child to take its place

- Case 3: 2 children- Here we must take the following steps.
    1. Delete the node
    2. Replace the node with its next node in non-descending order (call this the promoted node)
    3. Move the promoted node's former right child to take the spot the promoted node has left open.

The following example illustrates the 3rd case. Here, we are deleting the node with value 15.



*Note that the above example is a valid binary tree, but not an AVL tree since the AVL property is not observed.

In practice, we must update node heights and balance factors whenever an insertion or deletion is done and perform rotations whenever a node's balance factor magnitude is found to exceed 1.

Implementation Details

An AVL tree has been implemented in the file *avl_imp.py*. The AVL tree has been implemented as a class with the following methods and attributes:

ATTRIBUTES

tree- A nested array data structure containing the AVL tree
format for a given node element, i:
tree[i][0]= node value
tree[i][1]= left child index
tree[i][2]= right child index
tree[i][3]= parent index
tree[i][4]= node height
tree[i][5]= node balance factor

root_index- The array index of the root element

free_spots- an array of indices of AVL elements which have been deleted and can be re-assigned

METHODS

find- Search for node of provided value
update_heights- Update node heights of provided node and all of its ancestors
rotate- Rotate sub-tree at provided index to rebalance tree and recover AVL property
insert- insert new node of given value (duplicates are allowed)
next- return the index of the next node in ascending order
sort_tree- return a sorted array of all element values (ascending)
delete_by_index- delete the node at the provided index
delete_by_value- delete the first node found with provided value

Performance Measurements and Analysis

The appeal of an AVL tree is its ability to quickly search, insert, and delete elements while maintaining an ordering. We will find that AVL trees take a significant amount of time to build initially, but offer great performance for these 3 methods once constructed.

Building AVL Tree

The theoretical running time for constructing an AVL tree is bounded by O(nlog(n)) where n is the number of elements used to build the tree. This is accomplished by calling an "insert" method on each input value. The time required to search for the site to insert a new element is bounded by O(log(n)), as with any balanced binary search tree.

Performing a rotation (if required) is a constant time process and updating the heights of all ancestor nodes contributes an additional log(n) term since the height updates follow a direct path to the root. Therefore, the total runtime for a single insertion is:

O(a*log(n) + b*log(n) + C) = O(log(n))

Where
a= some constant associated with find method
b= some constant associated with height update method
C= some constant associated with rotations
n= the size of the AVL tree at the moment of insertion

This process is done "n" number of times with the size of the AVL tree growing by 1 with each insertion. Therefore, the runtime of building an AVL tree is bounded by

$$O\left(\sum_{1}^{n}\log(n)\right) = O(\log(n!)) = O(n\log(n))$$

Where the above equality follows from the properties of logarithms.

Sorting AVL Trees

The theoretical running time for returning a sorted array from an AVL Tree (already constructed) is bounded by O(n). In order to sort an AVL tree, we must first find the lowest valued node by recursively following the left child of the root node. This process takes O(log(n)). We must then call a “next” method repeatedly until we reach the largest value of the tree. Each next operation is bounded by O(log(n)) since we can (at most) climb the entire height of a tree when finding the next node.

From this, it follows that returning a sorted array from a pre-built AVL tree is bounded by O(nlog(n)). However, the majority of “next” operations will have us traverse just 1 position on the tree. In fact, each node is visited no more than twice during this sorting process, implying that the amortized runtime is bounded by:

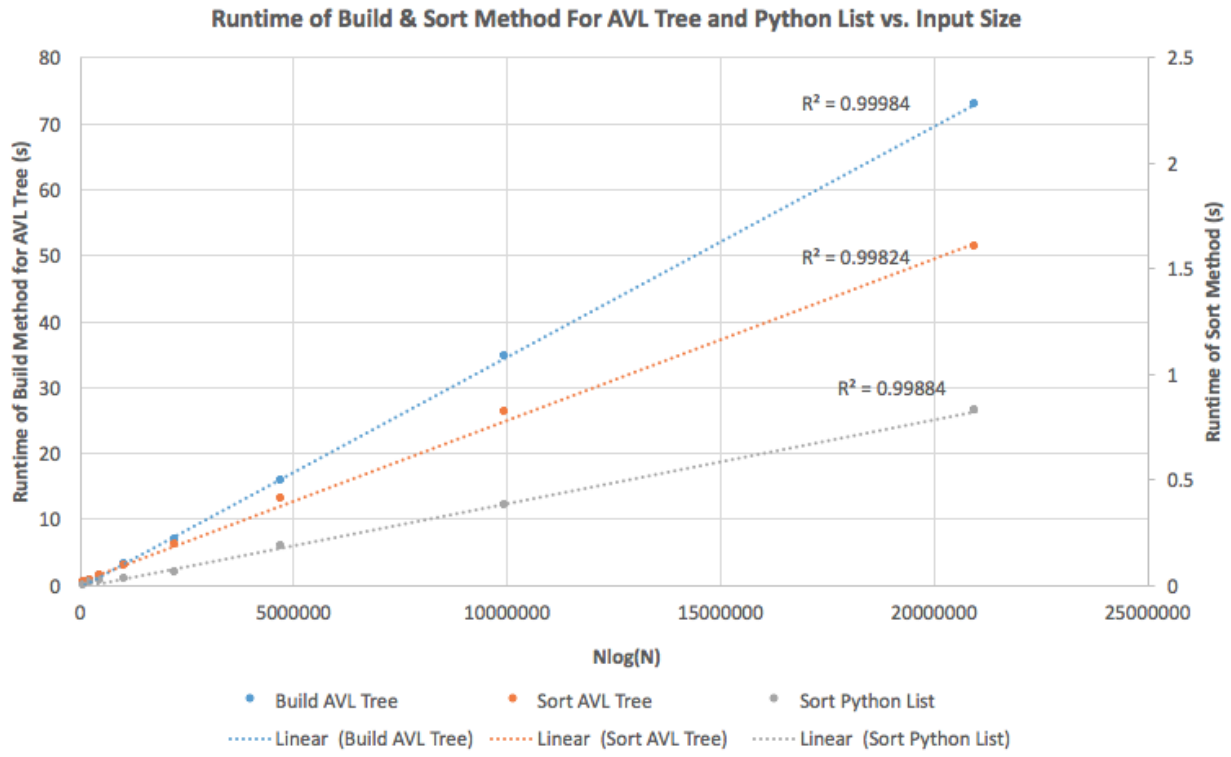
$$O(2n) = O(n)$$

Measurements

The runtime of building and sorting an AVL tree has been measured experimentally using the implementation being described in this document. For comparison, a python list was created and sorted using the same randomized input and had its runtime measured. The python sort method uses the “timesort” algorithm which is bounded by O(nlog(n)).

Several different input sizes were used to initialize an AVL tree. In particular, randomized input sizes of increasing integer exponents of two were used. These included inputs of size 2^{13} (8192), 2^{14} (16384), … 2^{20} (1048576). The input included both positive and negative integers with duplicates allowed. The allowed range was (-n, +n) where n is the input size. The results were plotted against nlog(n) (where n=input size) as well as against n in order to find the better fit and compare with theoretical predictions.

The graph below depicts the results of this experiment

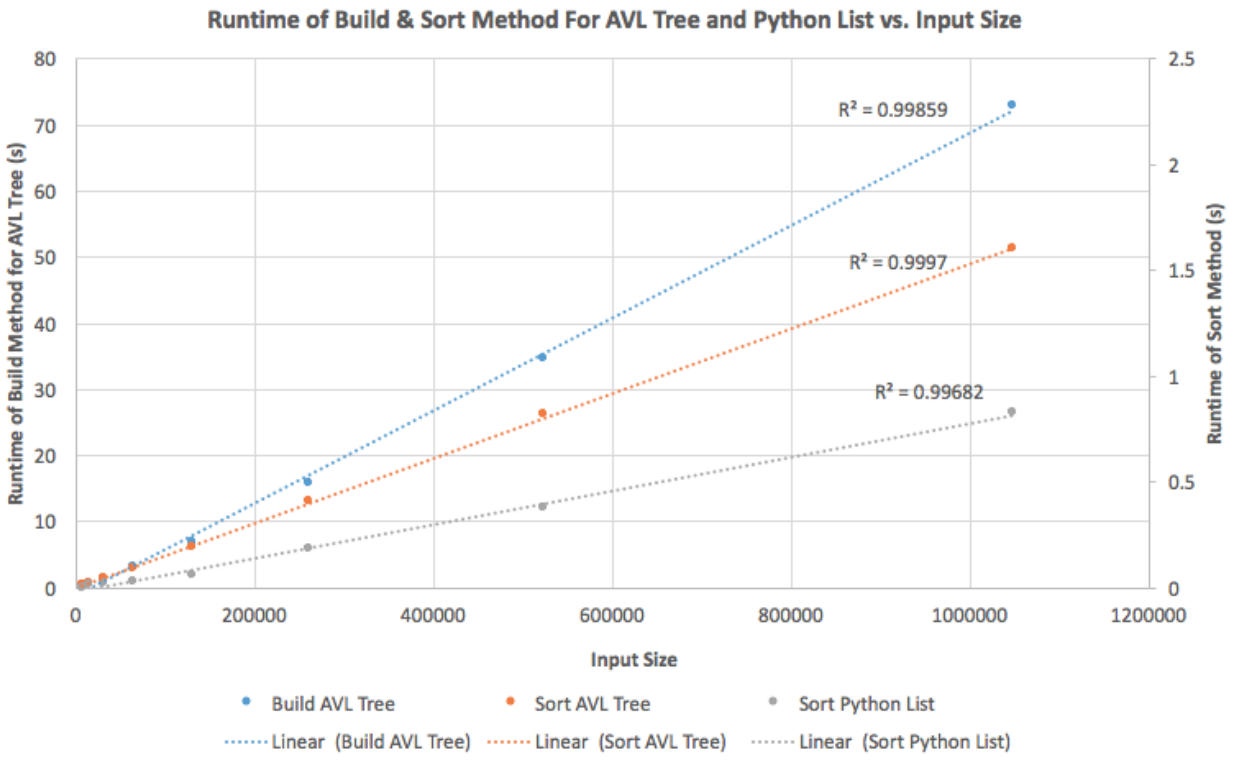


The left vertical axis includes the runtime for building the AVL tree whereas the right vertical axis shows runtime for the sort method for both the AVL Tree and the Python List. Notice that the horizontal axis graphs Nlog(N) where N is the input size.

According to the preceding discussion, the runtime of building the AVL and sorting the python list should be bounded by $\theta(n\log(n))$. The runtime of sorting the pre-built AVL tree should be bounded by $\theta(n)$. Therefore, we should expect a linear trend for building the AVL tree and for sorting the python list when runtime is plotted against nlog(n).

It appears that the runtime of all 3 methods are fairly well described by an nlog(n) trend, as all R^2 values exceed 0.998.

For comparison, the same data has been plotted against n instead of nlog(n) as seen below:



Here we see that the AVL sort method has a better fit with a linear trendline as compared to an nlog(n) trendline (R^2=0.9997 vs. R^2=0.99824). The Build AVL tree method and the python list sorting method both have a better fit when plotted against an nlog(n) trendline. All 3 of these outcomes are in agreement with theoretical predictions.

As mentioned earlier, the AVL Tree suffers from a long runtime for the initial build. With an input size of 2^{20} (~1 million) elements, the runtime was found to be 73 seconds.

The python sort method appears to be roughly twice as fast as the sort AVL tree method with input sizes tested. A considerably larger input size would be needed in order for the linear AVL sort method to outperform the python sort method which grows like nlog(n).

Find, Insert, & Delete Method

The theoretical runtime for searching for an element within an AVL tree is bounded by O(log(n)). This follows from the fact that searching for an element requires visiting “h” number of nodes where “h” is the height of an AVL tree. Since AVL trees are self-balancing, this height is guaranteed to be bounded by O(log(n)).

The insert and delete methods each use the find method. After the find method is carried out, the remaining work includes constant time processes (re-arranging/rotating a small number of nodes) as well as the O(log(n)) process of updating node heights. Therefore, the overall process is bounded by O(log(n))

$$\theta(insert/delete) = \theta(find) + \theta(rearrange/rotate\ nodes) + \theta(update\ heights)$$

$$= \theta(\log(n)) + \theta(c) + \theta(\log(n))$$

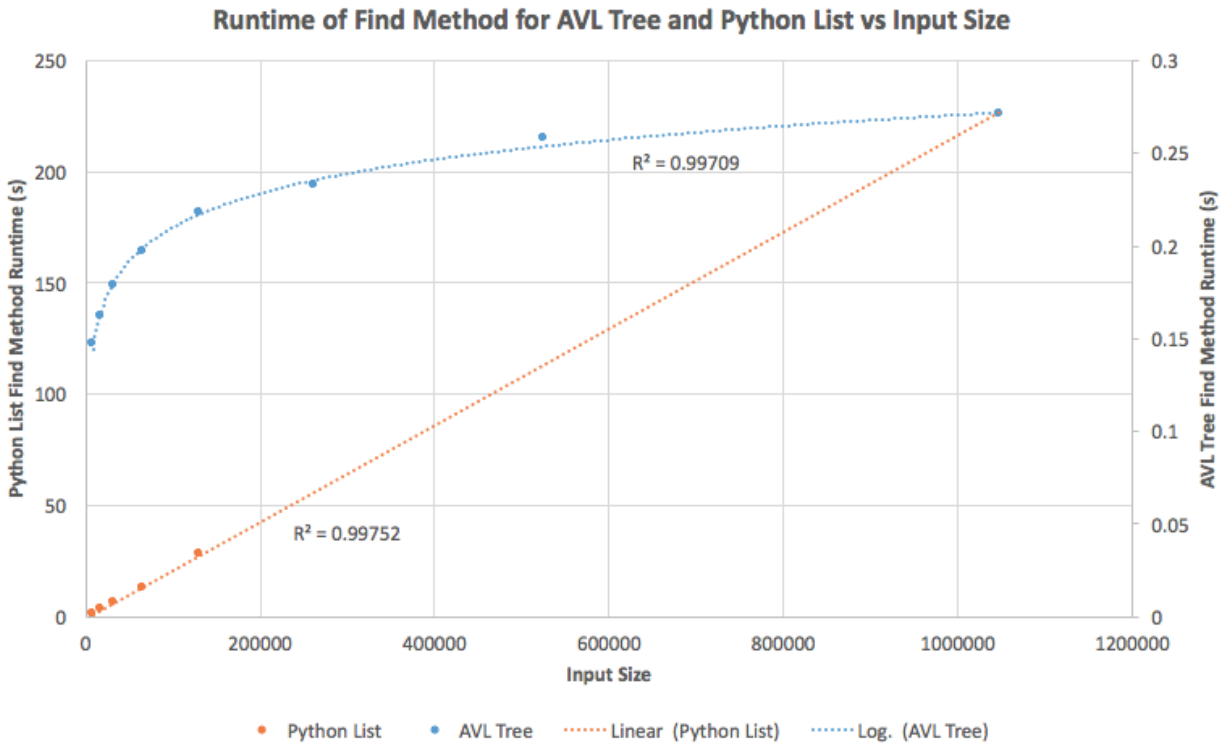
$$= \theta(\log(n))$$

An array, on the other hand requires potentially requires visiting every element of the array when searching for an element which results in an θ(n) runtime growth. Deleting an element from an array also requires adjusting the index of every element in the array which again grown like θ(n).

The insert method for a python list has an arbitrarily short runtime which is a constant time process if we append values to the end of the list.

Measurements

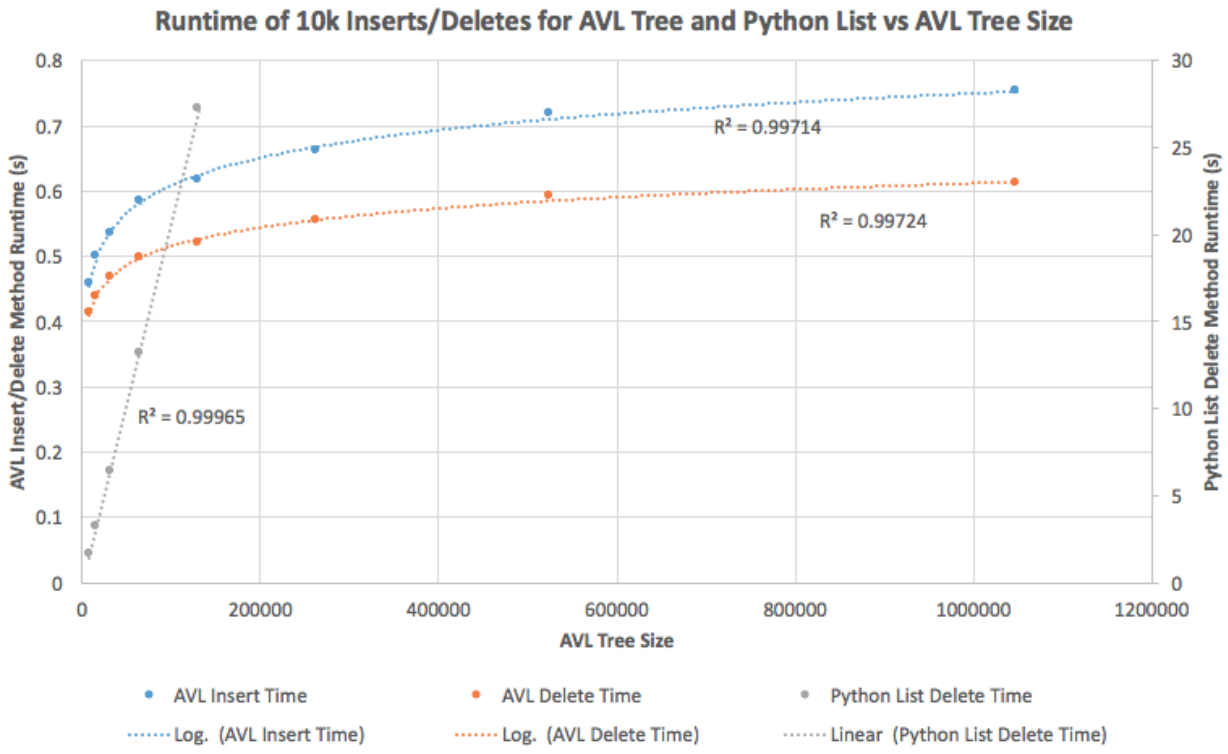
The time taken for 10k "find" calls was measured for AVL trees of sizes of increasing integer powers of 2. The searched values were randomized with a 50% chance of existing in the AVL tree. The same values were searched in a python array of the same size and content. The results are shown below.



The right vertical axis shows the runtime of the AVL "find" method whereas the left vertical axis shows the runtime of searching a value in a python list (result= <val> in <list>). Note that the intersection of the two graphs does not indicate equal runtime.

The AVL tree performed dramatically better than the python list for all input sizes tested (over 100 times faster for input of size 2^{17}, ~100k). Furthermore, the growth of the of the AVL find method grows like $\theta(\log(n))$ as opposed to the $\theta(n)$ growth for searching a python list.

We see a similar result when looking at the insert and delete methods. Again, 10k randomized values were chosen for each trial. Specifically, the 'remove' method was used with the python list.



This time we see the AVL tree performing ~50 times faster than python list's remove method for input of size 2^{17}, ~100k.

Space Complexity

A nested python list was used to store the AVL tree. Each node was stored as an element in this list. Each of these elements was itself a list which contained the node's value, pointers to its parent and children, the node's height, and the node's balance factor.

Whenever a node was deleted, the element at that node's index in the list was set to null. A separate list kept track of these nulled-out positions and would reassign these to newly inserted nodes before the length of the list was ever increased. Therefore, the space complexity of this data structure grows like:

$$\theta(5N_{OCC}) + \theta(2N_{DEL})$$

Where N_{OCC} is the number of occupied positions in the main AVL tree list, and N_{DEL} is the number of vacant positions. The vacant positions are counted twice these there is a single null entry in the main AVL tree list, and a single integer pointing to the vacant node indices in the 2nd list.

Since the maximum possible number of list elements only grows when an element is inserted into a fully occupied list, the space complexity is bounded by $\theta(\max(N))$ where max(N) is the maximum size of the AVL tree throughout the lifecycle of the data structure.

Conclusions

These experiments have demonstrated that the provided AVL tree implementation performs in accordance with theoretical values.

While the initial building of an AVL can be a very costly process, the time required to find, insert, and delete elements is dramatically superior to that of a basic array. The time required to return a sorted array from a pre-built AVL tree was slower, but comparable to the time required to use pythons sort method on an unsorted array. However, the AVL tree's sort method is expected to out-perform the python list sort method for larger input sizes due to the linear growth.

One area in which this implementation can be improved is in its space complexity. By using a linked list instead of a nested array, there will be no need to store any data for deleted nodes. This would reduce the space complexity from θ(max(N)) to θ(N) where N is the instantaneous size of an AVL tree and max(N) is the greatest size that an AVL tree reaches in its lifetime.