An Asymptotic Analysis of Sort, Build and Merge Functions on AVL Binary Search Trees

Joseph Scheidt

June 18, 2018

Abstract

This project involves a study of AVL trees, along with presenting algorithms for three different functions involving AVL trees: creating a tree from a sorted array of values, transforming an existing tree into a sorted array of values, and efficiently merging two different AVL trees.

AVL Tree Description

AVL Trees are a special type of binary search tree such that, for each node of the tree, the height of its children differ by at most one. As with all binary search trees, the values of the left children of a node are less than its value, and the values of the right children of that node are greater than its value.

Problem Statement

The given task is to design an efficient algorithm for merging two AVL trees, which should run in $O(\min\{m + n, m \lg(m + n), n \lg(m + n)\})$ time. As an intermediate step in this design, algorithms for transforming an AVL tree into a sorted array and for transforming a sorted array into an AVL tree, both in $O(n)$ time, will also be presented.

Build Algorithm

To build an AVL tree from a sorted array, we first observe that the root of the tree must come from near the midpoint of the array. In fact, by choosing the midpoint element of the array as the root of the tree, and by recursively building the left subtree with the elements left of the midpoint, and the right subtree with the elements right of the midpoint, we can build a balanced tree, regardless of the number of elements in the array. The pseudocode for this algorithm is presented below:

```
FUNCTION BUILDTREE (ARRAY, BEGINNING INDEX, END INDEX) }

    //Find middle element of array subset
    INDEX middle = BEGINNING INDEX + (BEGINNING INDEX + LAST INDEX)/2

    //build this node with the middle element as its key value
    NODE = new NODE(ARRAY[middle])

    //if there are more elements in sub-array to the left or right of
    //the middle elements, call build function for subtrees of this
    //node
    IF (BEGINNING INDEX < middle) {
        NODE.leftchild = BUILDTREE(ARRAY, BEGINNING INDEX, middle - 1)
    }
```

```
    IF (middle < END INDEX) {
        NODE.rightchild = BUILDTREE(ARRAY, middle + 1, END INDEX)
    }

    //return complete node with constructed subtrees
    RETURN NODE

}
```

Each node will take constant time to construct. As n nodes are constructed for an array of size n, our hypothesis therefore is that this algorithm will run in $O(n)$ time.

Sort Algorithm

In an AVL Tree, each node's left child is comprised entirely of elements smaller than it, and each node's right child is comprised entirely of elements larger than it. Therefore, we can call a recursive function to sort the AVL tree into an array as follows: sort the left subtree into the array, then add this node, then add the right subtree into the array. The pseudocode is below:

```
//create empty array of the same size as tree, with an index tracker
SORTED ARRAY = new ARRAY[TREE SIZE]
INDEX = 0

//call recursive function on the root of the tree
SORT TREE(TREE ROOT)

FUNCTION SORT TREE (NODE) {

    //sort the left subtree into array
    IF(NODE has left child) {
        SORT TREE (left child)
    }

    //add this node's key value and increase index by 1
    ARRAY[INDEX] = NODE.VALUE
    INDEX = INDEX + 1

    //sort the right subtree into array
    IF(NODE has right child) {
        SORT TREE (right child)
    }
}

//return sorted array
return SORTED ARRAY
```

Each node will take constant time to add to the array, and increase the index value. The time complexity of this algorithm will therefore be O(left subtree) + $O(1)$ + O(right subtree). Recursively we find that each node accounts for $O(1)$ time. Therefore, for a tree with n nodes, we hypothesize that this algorithm will return a sorted array in $O(n)$ time.

Merge Algorithm

Before we present the algorithm for merging two AVL trees in $O(\min\{m + n, m \lg(m + n), n \lg(m + n)\})$, we first note that the minimum of the three above complexities is $m + n$. We would prefer, whenever possible, to use an algorithm that executes in $O(n + m)$ time. Between the other two complexities, which one is smaller depends on the size of m and n. If m is smaller, $m\lg(m+n)$ is preferred, and if n is smaller, $n\lg(m + n)$ is preferred.

For case 1, we note that, using our previous algorithms, we can transform a tree of size n into a sorted array in n time, and a tree of size m into a sorted array in m time. If these two arrays can be simply merged into one sorted array, a new tree can be reconstructed in $m + n$ time. In order for the new array to be correctly sorted after a merge, one tree's maximum element must be less than the other tree's minimum element. Finding the maximum or minimum element of a tree depends on the height of the tree, therefore it takes lg n time. Finding the maximum and minimum of each tree will take 2 lg n + 2 lg m time. So our hypothesis is that in total, for two trees of size m and n whose ranges do not overlap, said trees can be merged in 2 lg$n$ + 2 lg$m$ + m + n + (m + n) time, which is equal to $O(m + n)$ time.

For cases 2 and 3, we select the smaller of the two trees (we'll call it the one of size n here). This tree, we will sort into an integer array, which, as already posited, will take O(n) time.

We will then insert each element from the array one at a time into the other tree. Insertion with binary search trees is determined by the height of the tree, as the maximum number of comparisons needed to place the new element equals the maximum height of the tree, which is lg(n). For AVL trees, after insertion we must also retrace our path up the tree, rebalancing any parts which have become unbalanced by the insertion. Again, this step takes lg(n) time.

In this case, the maximum height of the tree for the last insertion will be lg(m + n).

Therefore, to sort the smaller tree of size n into an array, and to then insert each of n elements into the larger tree of size m, it will take n + n * 2 lg(m + n) time, or O($n$lg(m+n)).

Our hypothesis, then, is that for two AVL trees with non-overlapping ranges, this algorithm will merge them in O(m + n) time, whereas for other AVL trees, it will merge them in O(m lg(m + n)) time if m is smaller, or O(n lg(m + n)) time, if n is smaller.

Hypotheses

In summary, we posit the following:

- Using our build algorithm, a tree can built from a sorted array in O(n) time.
- Using our sort algorithm, a tree can be sorted into an array in O(n) time.
- Using our merge algorithm, two trees whose ranges of values do not overlap can be sorted in O(m + n) time. Other trees can be sorted in O($n$lg(m+n)) time, where n is the smaller of the two trees.

Expermental Design

To test our hypotheses, we will count the number of comparison statements executed in an implementation of our algorithms on AVL trees in the Java programming language. We will use three types of data sets to merge, each in five different sizes, in order to track the asymptotic growth in the number of comparison statements. We will then compare the results with our theoretical analysis.

The five sizes of data sets will be: k = 100000, k = 200000, k = 300000, k = 400000, and k = 500000.

We will first build a tree of size k from the data set and count comparisons made. We will then take that tree and sort it back into an array, again of size k.

The three types of merge data sets will be as follows:

Type 1 Merge

Tree 1 contains {1 to (k / 2)}; tree 2 contains {(k / 2) to k} (non-overlapping sets)

Type 2 Merge

Tree 1 contains {1 to 0.2k, k}; tree 2 contains {0.2k to (k-1)} (unbalanced set sizes)

Type 3 Merge

Tree 1 contains {1, 3, 5, 7, . . . , k - 1}; tree 2 contains {0, 2, 4, 6, . . . , k - 2} (balanced set sizes)

Results

Implementing the algorithms in Java and tracking the number of comparisons executed yielded the following results:

Number of Comparisons Made

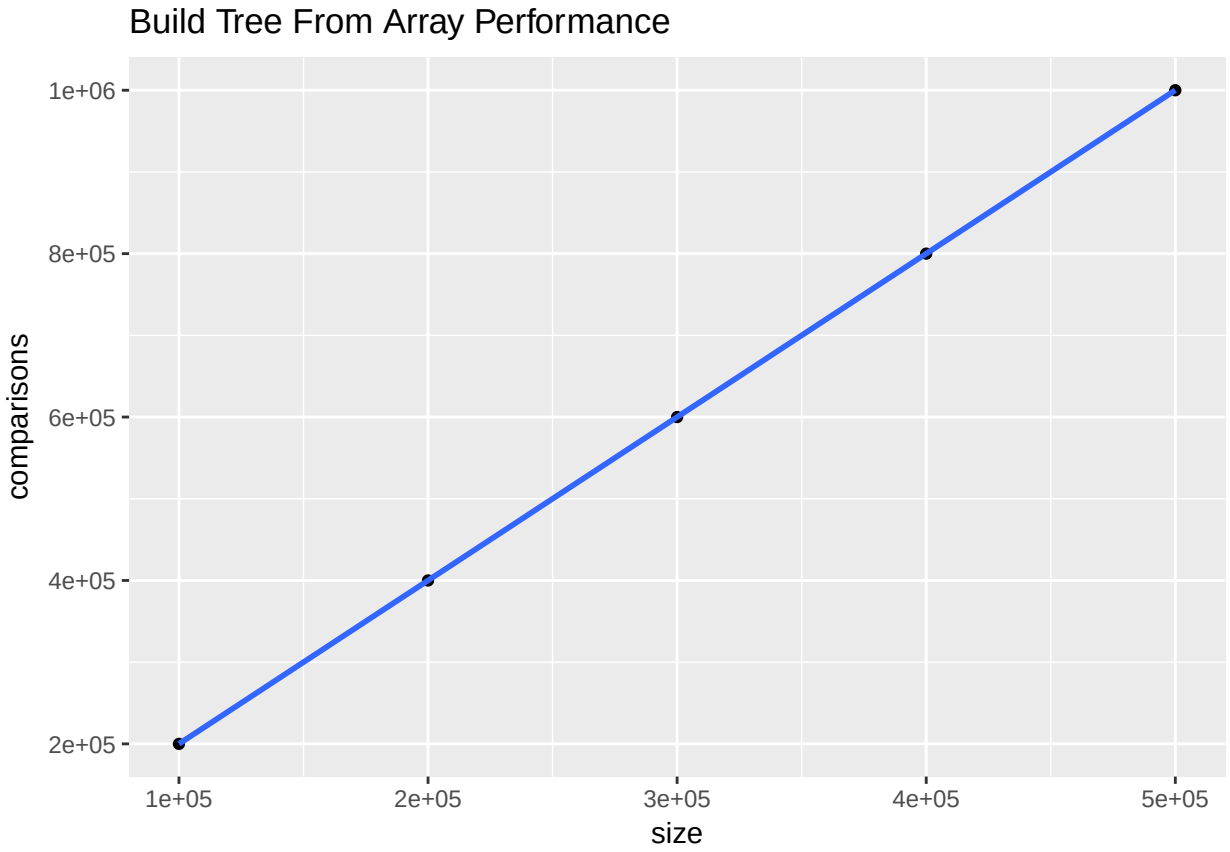
$SetSize$	$Build$	$Sort$	$Type1Merge$	$Type2Merge$	$Type3Merge$
100,000	200,000	200,000	500,030	780,087	1,718,990
200,000	400,000	400,000	1,000,032	1,640,093	3,637,922
300,000	600,000	600,000	1,500,034	2,460,097	5,625,782
400,000	800,000	800,000	2,000,034	3,440,099	7,675,782
500,000	1,000,000	1,000,000	2,500,034	4,300,103	9,725,782

Type 1 merges, as predicted, were the quickest. Type 2 merges, with one tree four times larger than the other tree, were next quickest. Type 3 merges, with two trees the same size, were slowest.

Analysis

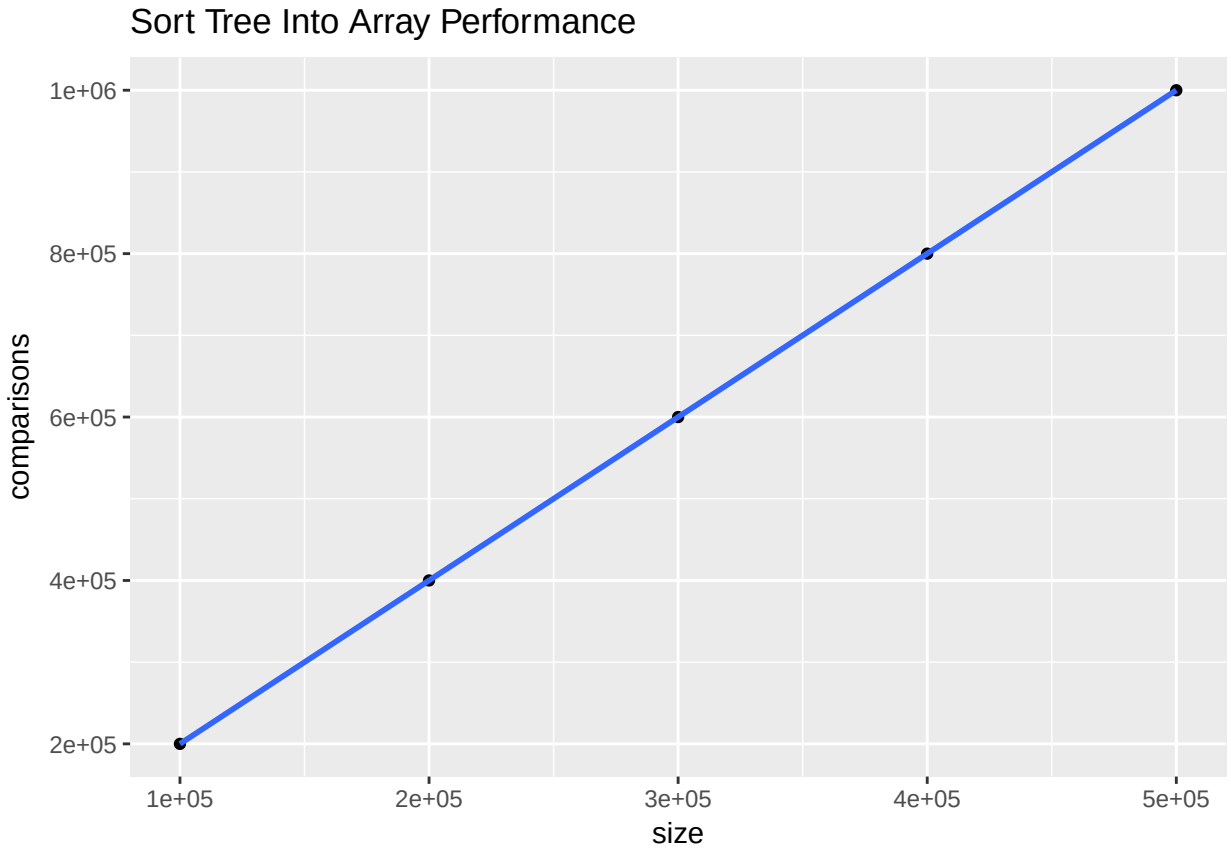
Graphing the results should help reveal whether the algorithms performed on the predicted order of time.

Build Algorithm



Our implementation of the algorithm building an AVL tree from a sorted array performed in $O(n)$ time.

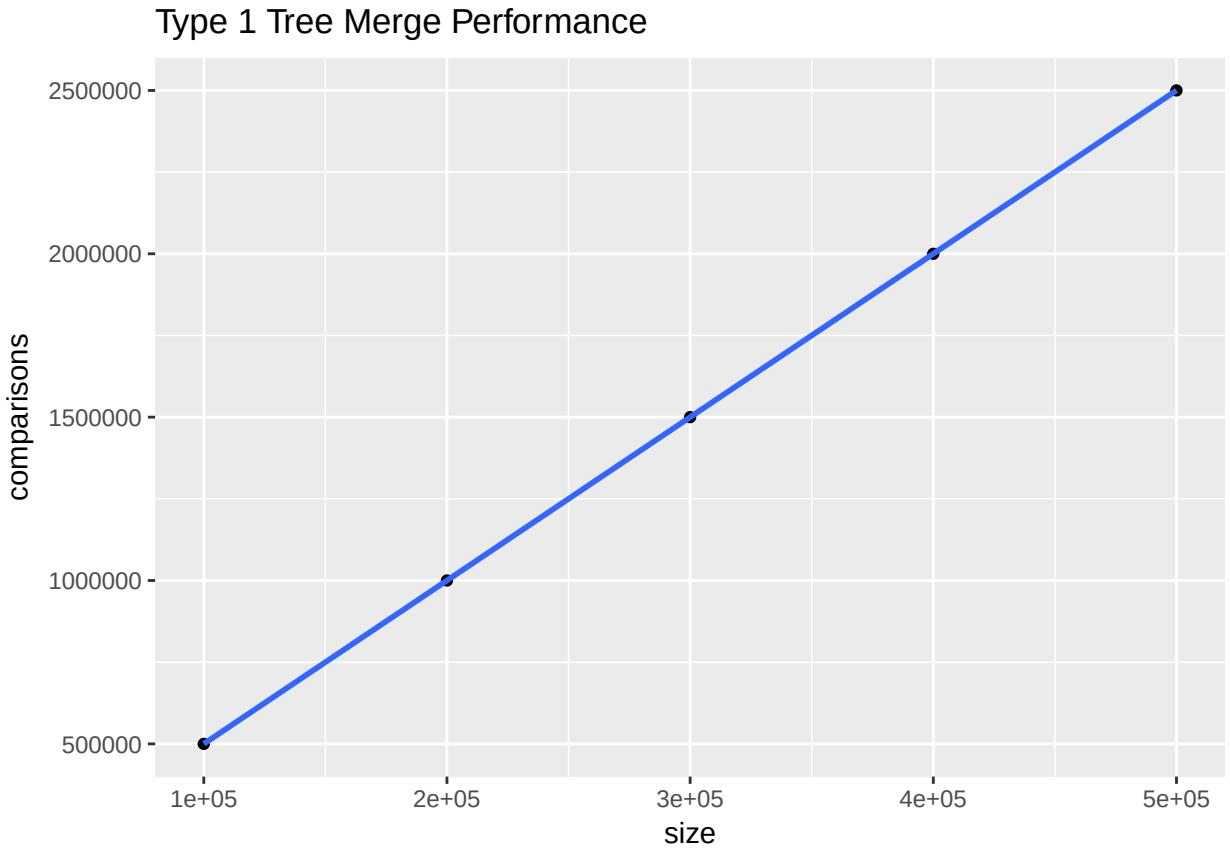
Sort Algorithm



Our implementation of the algorithm sorting an AVL tree into a sorted array also performed in O(n) time.

Type 1 Merge

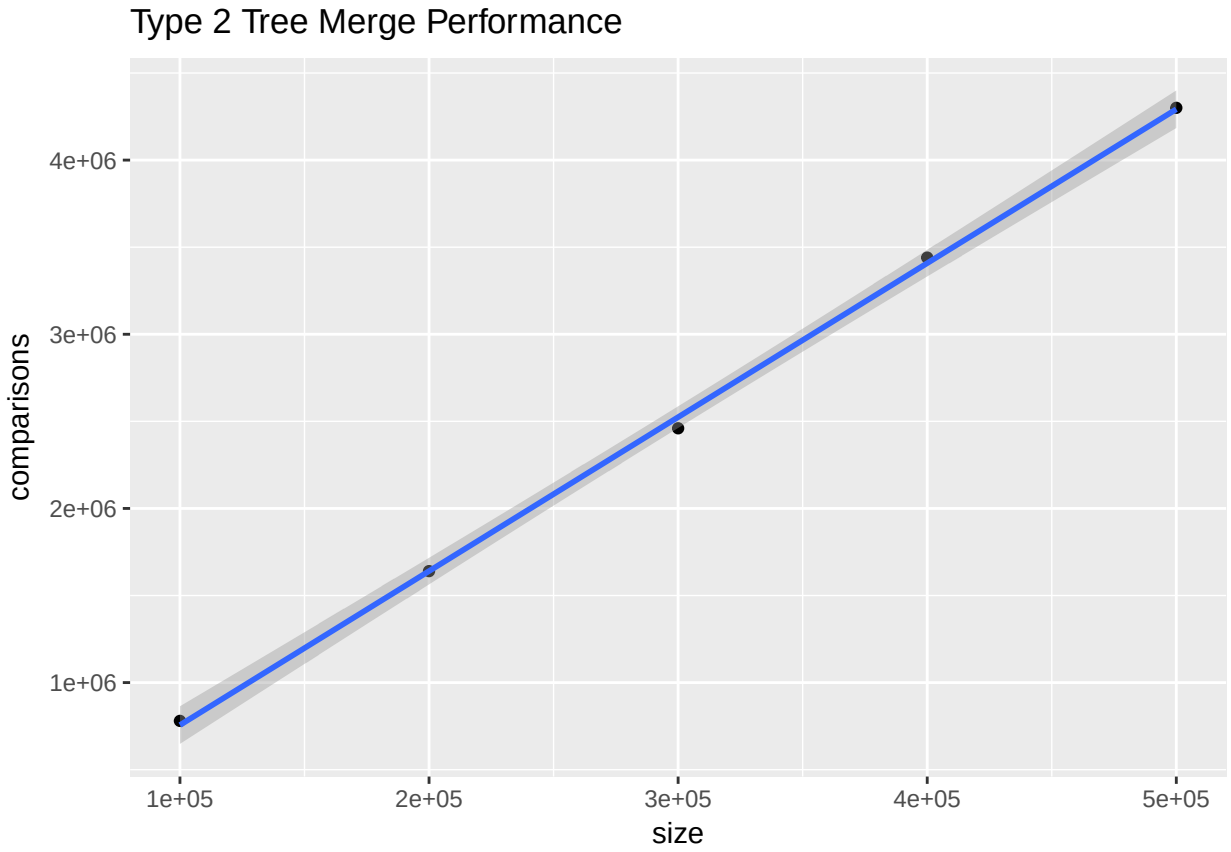
As a reminder, our Type 1 merges were merges of trees with nonoverlapping ranges of values, so the program sorted each tree into an array, merged the two arrays, and rebuilt the resulting array into a tree.



This algorithm also performed in O(n) time. Here, however, the constant coefficient for n is 5, instead of 2 as in the two algorithms above.

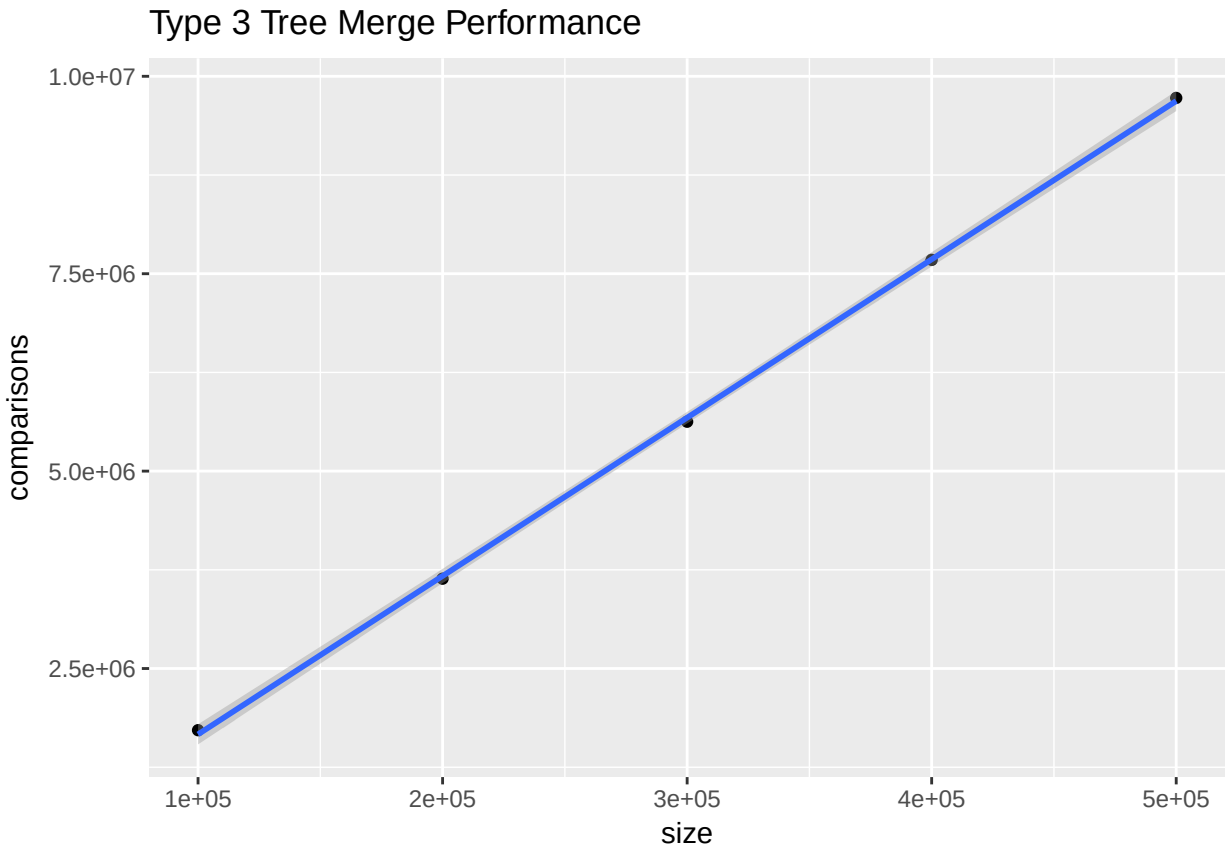
Type 2 Merge

The algorithm for type 2 and 3 merges were to take the smaller of the two trees, sort it into an array, and insert it element by element into the larger tree.



The performance is still nearly, but not quite, linear when one tree is significantly smaller than the other. Still, $O(n\lg(m+n))$ is likely to be the correct bound here.

Type 3 Merge



Again, the performance appears to be nearly linear, however, in this graph you can see the data points linked together are very slightly concave upward, meaning $O(n\lg(m+n))$ is likely the correct order for this algorithm.

Conclusions

By implementing the posited algorithms and testing for asymptotic growth, we have given support to our hypotheses regarding those algorithms. The test results did not contradict our predicted orders of growth. While more rigorous testing would be necessary to draw firm conclusions, we can at least say that are algorithms are more likely than not to be ones that fulfill the requirements of the problem statement.

Citation

AVL Trees, named after their creators, were first presented in the following paper.

Georgy Adelson-Velsky, G.; Evgenii Landis (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences* (in Russian). 146: 263–266. English translation by Myron J. Ricci in *Soviet Mathematics - Doklady*, 3:1259–1263, 1962. Retrieved from http://professor.ufabc.edu.br/~jesus.mena/courses/mc3305-2q-2015/AED2-10-avl-paper.pdf