Programming Assignment 02

1DV516

Name: Tadj Cazaubon

Student: tc222gf

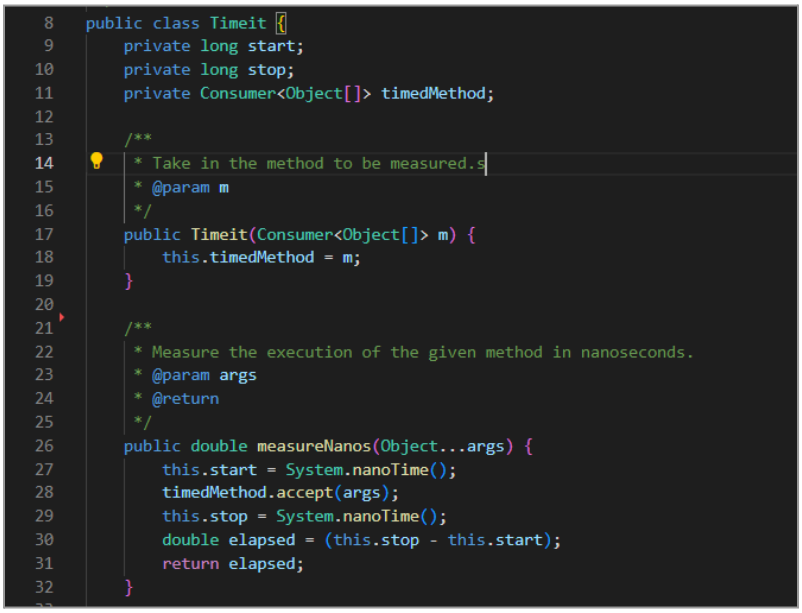
Email: tc222gf@gmail.lnu.se

Prerequisites

I ran all readings on the same Windows desktop system. I fixed the CPU clock speed to 4.4GHz and closed all background processes besides those needed by the OS.

The Timeit Class

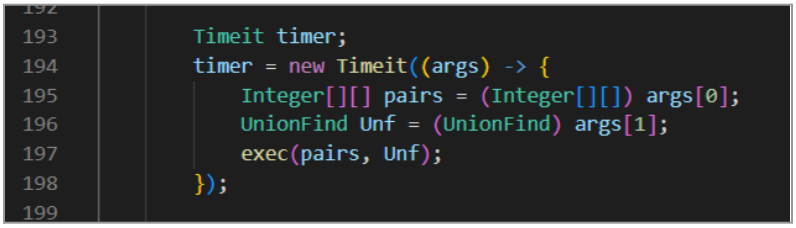
To time the execution of methods as specified in Assignment 01 - Problem 3, I've created the Timeit class. This works similarly to the python package of the same name, where a code snippet / method can be run, and a timer of its execution is given. Within my Timeit implementation a Consumer object can be passed in along with any number of args during instantiation. The method can then be run with the passed args. The only downside of this is the need for arguments passed to be an object, meaning no primitive data types can be passed without being wrapped within some kind of object.



```java
public class Timeit {
    private long start;
    private long stop;
    private Consumer<Object[]> timedMethod;

    /**
     * Take in the method to be measured.s
     * @param m
     */
    public Timeit(Consumer<Object[]> m) {
        this.timedMethod = m;
    }

    /**
     * Measure the execution of the given method in nanoseconds.
     * @param args
     * @return
     */
    public double measureNanos(Object...args) {
        this.start = System.nanoTime();
        timedMethod.accept(args);
        this.stop = System.nanoTime();
        double elapsed = (this.stop - this.start);
        return elapsed;
    }
```

In practice, args must be specified unless specifically programmed for, meaning I specify a simple anonymous function to separate the arguments and pass them to the method being timed correctly. I figure that the impact of this extra step on the final times is small, and when looking at the growth of an algorithm on a millisecond scale when the initial separation step takes nanoseconds, the results should remain consistent.

An example of one of these is the one used for the UnionFind analysis in Assignment 01 - Problem 4:



```java
        Timeit timer;
        timer = new Timeit((args) -> {
            Integer[][] pairs = (Integer[][]) args[0];
            UnionFind Unf = (UnionFind) args[1];
            exec(pairs, Unf);
        });
```

Here, the exec method is the thing we are looking for. This method, as we will see later, takes multiple milliseconds in some cases.

.

1.0. Problem 05

While I had already implemented a binary Tree in Problem 04, I took the advice of using a “limited implementation of a binary tree, e.g., not containing general methods to add, delete, etc. In this approach, Trees are represented via their root Node pointers in most methods, and manipulated by static class methods within the TreeIsomorphism class.

1.1. Procedure:

In finding if two Trees are isomorphic, I have created the method isIsomorphic(), which takes in two root Nodes of trees and recursively searches through them. At each Node, it’s checked whether the current Node of both trees are equal. If any combination of two of a node’s children exists for the corresponding node of the other tree, it’s considered to still be isomorphic. This continues until either a non-corresponding node combination is reached, or the entire tree is iterated through.

To test the growth of the trees for large numbers of nodes while maintaining the idea of an initially average tree, I employ a few static methods.

The Randomize() method allows me to randomize BSTs as I attempt to get them to reach a certain number of nodes.To enable sufficient randomness while ensuring that the Tree does eventually get filled, I took an idea from the previous Percolation Problem in assignment 01. I set it to add an element randomly 58% of the time, whilst removing an element randomly 42% of the time.

```java
/**
 * Given the root Nodes of two trees, check them for "strict" isomorphism.
 * @param root1
 * @param root2
 * @return
 */
public static <T extends Comparable<T>> boolean isIsomorphic(BSTNode<T> root1, BSTNode<T> root2) {
    /**
     * If both roots are null, return true.
     */
    if (root1 == null && root2 == null) {
        return true;
    }
    /**
     * If only one is null, return false.
     */
    if (root1 == null || root2 == null) {
        return false;
    }

    /**
     * If the values are not the same, return false.
     */
    if (!root1.value.equals(root2.value)) {
        return false;
    }

    /**
     * Check all combinations of left and right nodes for the wo input nodes.
     * We don't need to explicitly return True because that will happen when a leaf is reached.
     * Either the nodes are exactly the same, or they are swapped.
     */
    return (isIsomorphic(root1.left, root2.left) && isIsomorphic(root1.right, root2.right) ||
            isIsomorphic(root1.left, root2.right) && isIsomorphic(root1.right, root2.left));
}
```

As with the previous assignment, I define a Timer which separates the passed arguments, in this case the Two root nodes, and runs the method. When creating trees of an initial size, a static size didn’t seem appropriate, so a dynamic size of one tenth (1/10) of the total size would be the base size of the tree.

```java
private static <T extends Comparable<T>> void Randomize(BSTTree<T> a, int n) {
    int x = 0;

    Random rand = new Random();

    while (x != n) {
        T key = getValue(rand, n);
        if ((rand.nextInt(10) + 1) > 0.42) {
            a.add(key);
            x+=1;

        } else {
            a.remove(key);
            x-=1;
        }
    }

    if(x != n) System.out.println("Couldn't randomize fully!");
}
```

To retrieve the readings, I've defined the runSame() and runDiff() methods. These are to get the timings of isomorphic checks for identical and randomly different trees of same height respectively. These are essentially the same, save for the fact that the latter gives the height of both trees (which are usually nearly identical).

The sampling scheme is simple. The trees are created and randomized, then an initial measurement to eliminate a strange bug I found is made. The tree is padded with nodes by a certain interval, then the measurement is made for checking isomorphism. This means that each run is using the same tree across its measurements, we can see measurements for large numbers of nodes without taking that many measurements, and we can arbitrarily specify intervals to have more or less measurements for a given range of nodes

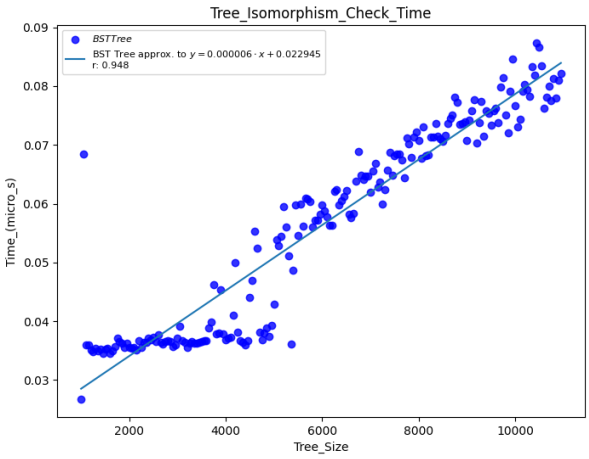
```java
private static <T extends Comparable<T>> Double[][] runSame(Integer amt, int spacing) {
    int amount = (int)(amt/spacing);
    double[][] runs = new double[SAMPLES][amount];
    double[][] heights = new double[SAMPLES][amount];
    T[] padders;
    double __;


    BSTTree<T> a = new BSTTree<>();
    BSTTree<T> b = new BSTTree<>();

    for (int i = 0; i < SAMPLES; i++) {
        a = new BSTTree<>();
        b = new BSTTree<>();
        BSTNode<T> aN = a.getRoot();
        BSTNode<T> bN = b.getRoot();
        Randomize(a, b, amt);
        __ = Timer().measureMicros(aN, bN);

        for(int j = 0; j<amount; j++) {
            padders = getGenericArray(spacing);
            execAdd(padders, a, b);

            runs[i][j] = Timer().measureMicros(aN, bN);
            heights[i][j] = a.height();
            System.out.println("SAMPLE: "+  i + "\t|\t" + "Point: " + j + "\t|\t" + "Heights:" + heights[i][j]);
        }
    }

    Double[] times = Util.sampleMean(runs);
    runs = null;
    Double[] h = Util.sampleMean(heights);
    heights = null;

    return new Double[][] {times, h};
```

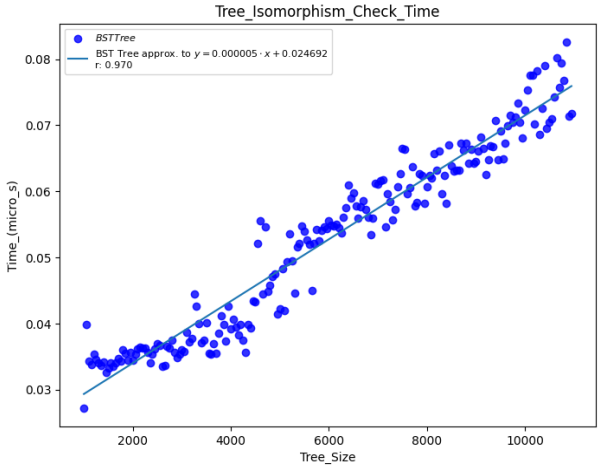
Each reading is stored in a sub-array of samples, along with the height of the trees. The samples are averaged together with outliers removed via the IQR before being returned.

1.2. Overview

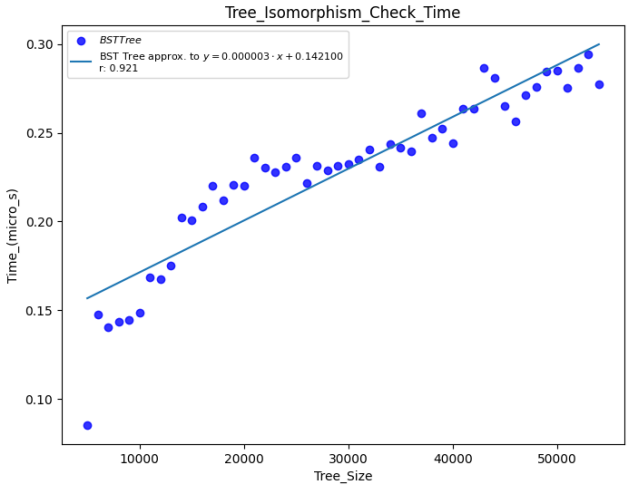
It is my assumption that the complexity of this algorithm is O(min(m,n)), where m and n are the number of nodes in the two trees. This is because in the worst case, the algorithm needs to traverse all nodes in both trees. However, the traversal stops as soon as a non-isomorphic structure is found, so if one tree is significantly smaller than the other, the algorithm will not need to traverse all nodes in the larger tree.
Looking at graphs made from the readings though, we see a slightly different, more confusing story. Runs were done for trees of varying sizes, including 10,000, 50,000 and 100,000 nodes.

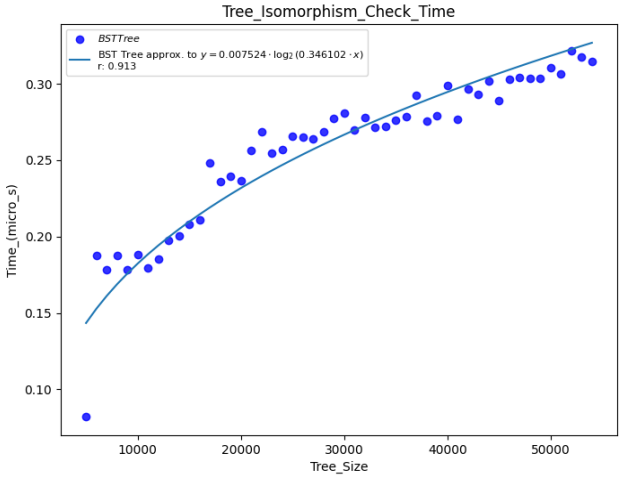
At the smallest size of 10,000, my assumption would seem somewhat validated. Here, the graph of time taken for the isomorphism check seems linear, though this would prove wrong later. These show the readings taken for different trees versus the same tree from left to right respectively.



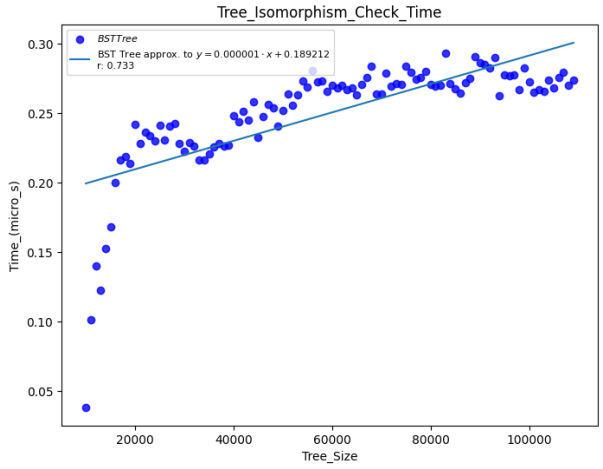


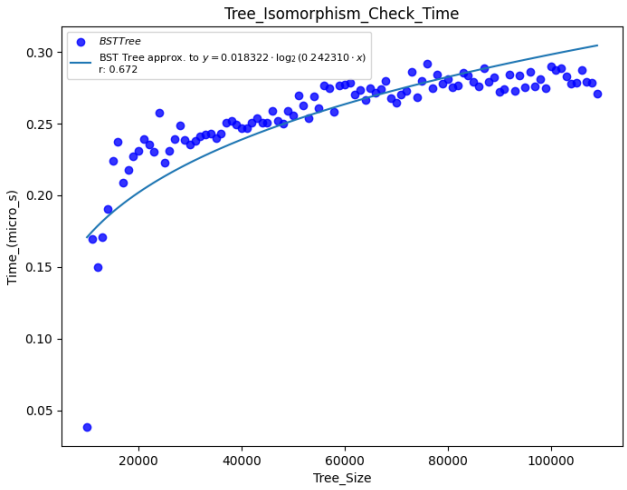
This trend remains somewhat at the 50,000 node mark for the differing trees, however, the identical trees have shifted to a pattern better described by a logarithmic curve than a line. It was my assumption that this is due a reflection of the height-dependent nature of the traversal and how many times the recursive function is called regardless of the actual number of nodes. This began to make less sense however as I looked at later readings.





Here now are the respective readings for the 100,000 node runs of the isomorphic check algorithm. As we can see, the trend towards what appears to be a logarithmic curve has become more prominent.



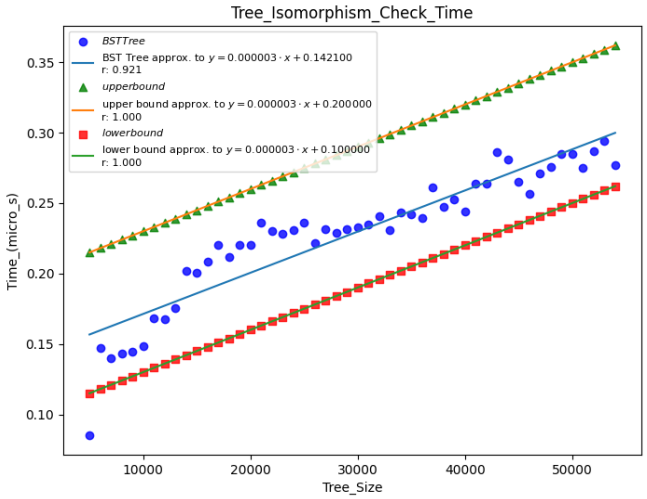


I realize now that this may be either an error in my implementation, or I lack the understanding of the complexity of the algorithm. However, the test cases I have implemented have passed, as well as the case presented within the assignment.

While I understand that the recursive calls are dependent on height, and that efficiency would be lost on a heavily unbalanced tree, I fail to see how the algorithm's average case would not be O(min(m,n)). The fact that the same time essentially is taken on average for both identical and dissimilar trees is also strange.

1.3. Analysis

I do not believe that the algorithm in question is logarithmic, and that I have some sort of error in my implementation. I will test the result of my implementation as being Θ(N), as even if they were Θ(LogN), they would also be Θ(N). As the two groups are so similar, I will analyze only the readings taken from dissimilar trees. Unless specified otherwise, I am speaking about the readings capping off at 50,000 nodes, as any larger becomes unwieldy for my current computer.



This therefore means there must be some positive constants c_1 and such that for all points T(N) on our graph, $T(N) \leq c_1 \cdot N, \forall N \geq n_0$, as well as some constant c_2 such that for all points on the graph T(N), $T(N) \geq c_2 \cdot N, N \geq n_0$. Looking at the graph, we can come up with the function line $g(N) = 3.0 \cdot 10^{-6} \cdot N + 0.2$, and $f(N) = 3.0 \cdot 10^{-6} \cdot N + 0.1$ as the upper and lower bounds respectively. These both use n_0=6000 as only the first value at (5000, 0.8518) is not within our boundary range.

Now having $c_1 = c_2 = 3.0 \cdot 10^{-6}$, we can say that T(N) is both O(N + 0.2) and Ω(N + 0.1) respectively. We can simplify further, dropping the constants from our complexities to end up with O(N) and Ω(N) respectively. Having this, we can now say that our measured values of T(N) for this given range are Θ(N) as it is both O(N) and Ω(N).

2.0. Problem 06

2.1. AVL Tree

To construct the AVL Tree, I've used a base abstract 'BST' class which the AVLTree class shares with the BSTTree class as a parent. This is to standardize the interface between them. I also did the same for the Nodes of the BSTTree and AVLTree, having them inherit a Generic LRNode class which can hold any data type once specified. The tree was constructed and given the basic methods needed for the assignment specifications, as well as the tests I have run. The AVL does lack iteration, but since the normal BST does, this isn't an issue.

2.1.1. Overview

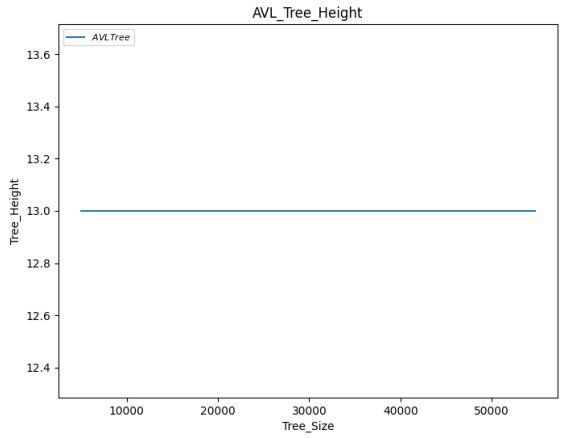
To test the various operations of the two trees , I've created the 'TreeComp' class, which shares much in common with the previous 'BSTIso' class, in that they operate purely using static methods. It is my assumption that due to the add and remove operations of these trees being completely dependent on their height, The AVL Tree outperforms the BST massively. Throughout the tests, the height of the AVL tree remained constant, even when adding ten (10) times the number of nodes already present within the tree. The BST on the other hand, seemed to climb linearly in height, ending up with the performance over time recorded below.
It is my assumption that the AVL tree performs insertion and removal operations in O(Log N) time, and will remain relatively unchanging in comparison to the BST, as the height of the BST will not only change, but balloon in size depending on the initial tree balance.

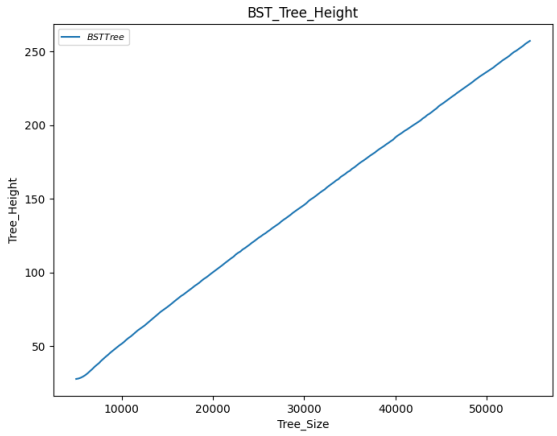
To test the Trees, I utilize two main methods alongside methods similar to those previously discussed, 'runDel' and 'runAdd'. These are not dissimilar to the 'runSame' method I mentioned in the previous Problem section. Each run is once again averaged, graphed and saved for later to be reused.

2.1.2. Analysis

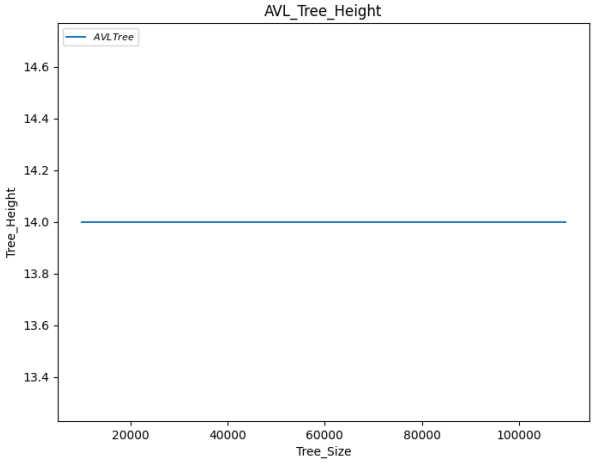
A number of sizes of Tree with various steppings between readings were tested. A number of observations can be made about the difference in the trees:

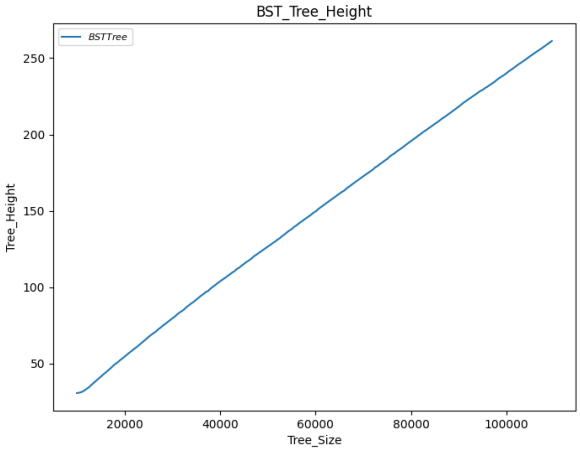
2.1.2.1. Height





The height of trees differ greatly. Not only has the AVL Tree remained at a much lower height overall, but the height stays consistent throughout the testing, even for large changes in the number of nodes, such the ten (10) fold addition mentioned. Above, we see an example of this in action. The BSt started the test already at a height of 50, and ended near 250, while the AVL Tree stayed constant at a height of 13. This holds true for other sizes as well.

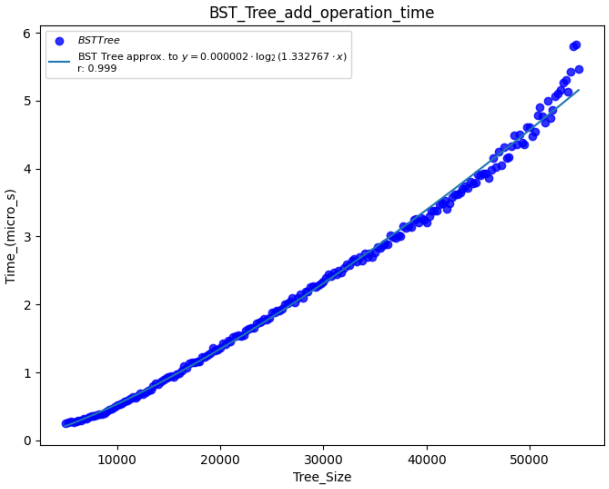


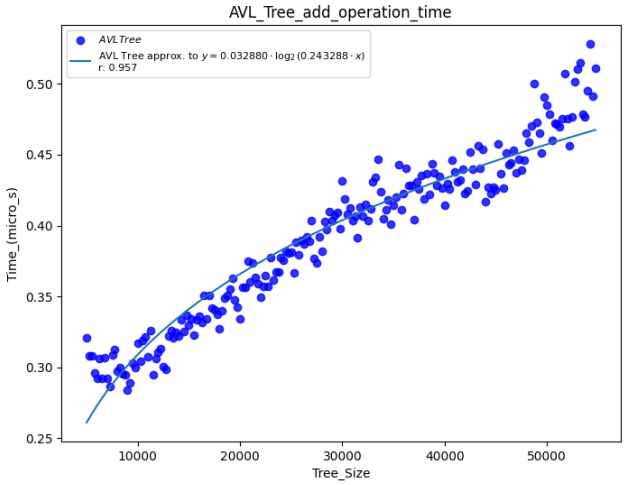


Here we see that the BSt was fairly balanced at the beginning of the test at a height of 50 with 20,000 items. However, it has increased to the same 250 by the end, unlike the AVL Tree which remains at 14.

2.1.2.2. Add operations

The time taken for add operations for Binary Trees are directly dependent on the height, with a time complexity of O(log N) for the average case. However, with degraded balance, degraded performance follows. We can see this clearly when looking at not only the time complexity of the results, but simply the time taken per operation. As expected, once the balance of the BST has degraded, its performance degrades more and more in line with being O(N) for add operations, as it must traverse longer and longer subtrees.

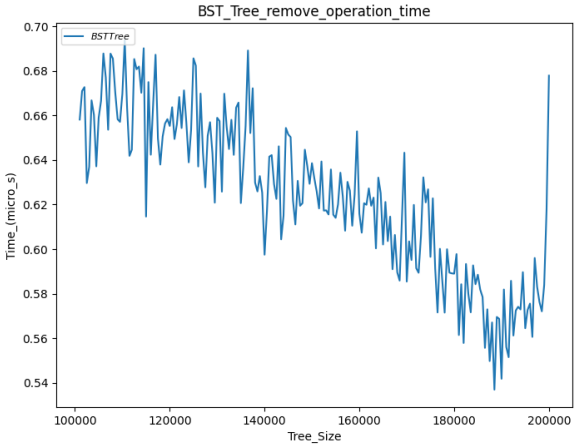


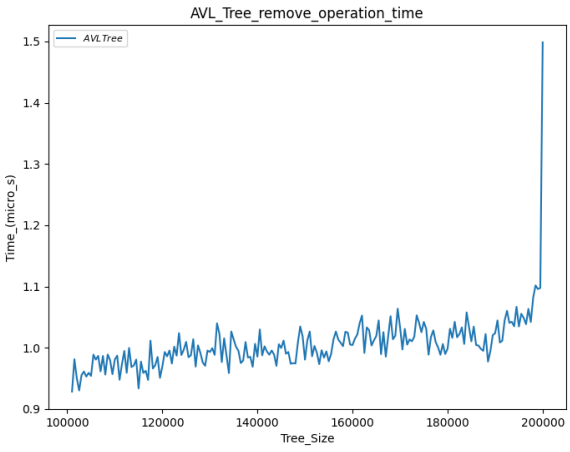


This leads to a situation whereby the AVL tree does increase in time taken per insert operation, but the difference in their increase inserting 50,000 elements is 6 microseconds for the BST, versus approximately 0.15 microseconds for the AVL Tree.

2.1.2.3. Delete Operations

I had found some difficulty in accurately displaying the changes in time for the delta operations between these trees. In the end, I settled on this simple plotting scheme.





The scale shows the Tree size, however the time scale is in the opposite direction. This graph shows the time taken for delete operations as items were removed in succession from the trees. Interestingly, while the BST increased the time taken as expected, it still took less time than the AVL Tree on average for removal operations. The AVL Tree however, actually decreased in the time taken to delete items, near to the time taken by the BST. The AVL Tree still initially does take longer than the BST per operation however, as it must rebalance itself on every delete operation, which is costly, but ensures much faster add and find operations. I did not test the find operations (the ‘contains()’ method), as both the delete and add utilize it in some way during traversal.