Self Balancing Binary Search Tree (AVL)

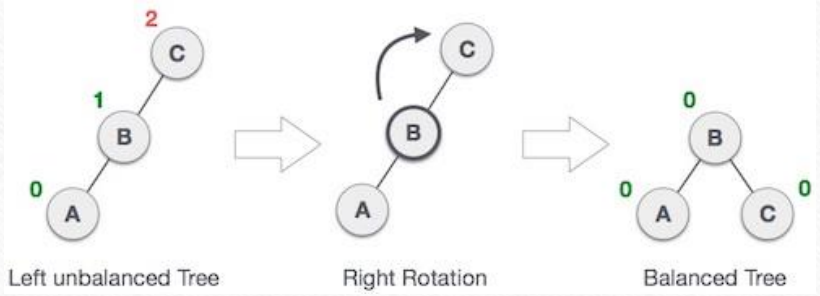
By Akarsh Kumar

Balancing

- In order for a binary search tree to function with proper low times, it must be balanced in order to reduce the time to add and remove nodes.
- Balancing on my tree was done using four different types of rotations.
- Rotations were to rotate nodes around in different places, eventually evening them out around the tree.

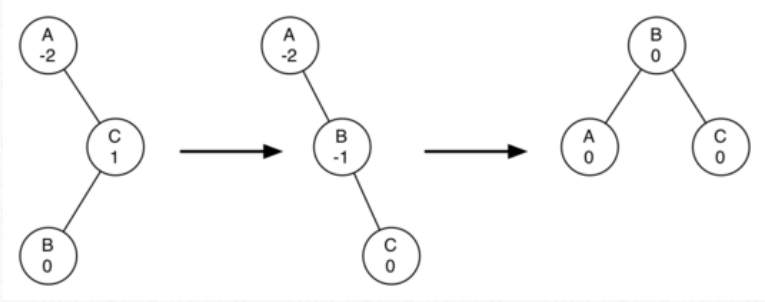
Simple Rotations

- The simple left and right rotations are used to change the arrangements of nodes when one child of a node is much "heavier" than the other, as in it has more children.



Complicated Rotations

- The simple rotations won't work in some situations, as they can run into a permanent loop not ever balancing.
- The solution to this is having another kind of rotation that temporarily unbalances the tree in order to balance it using simple rotations.



Adding

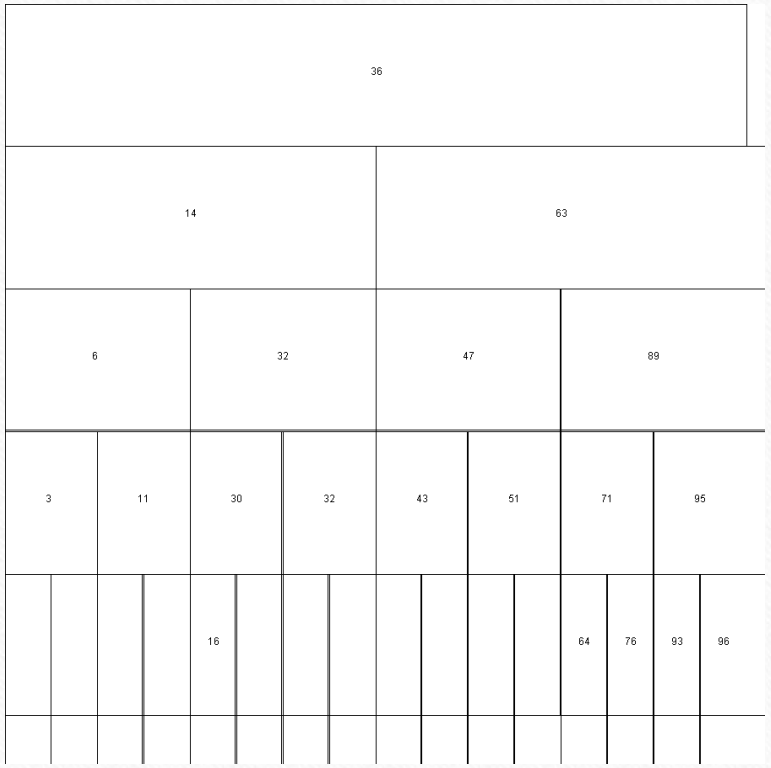
- The way I implemented the add method in my tree was to have a simple recursive call that goes down a line of the tree comparing the values of the nodes.
- The reason my first add method attempt failed on time is because I was balancing from the root all the way down the tree after every addition.
- The new updated code balances from the branch that was added onto up the tree, reducing the number of nodes check for balancing tremendously.

Deleting

- Deleting was achieved by first checking what kind of node it is based on the number of children it has.
- If it has no children, it can simply be untied from the tree and balanced from the removed node up to the root.
- If it has one children, the parent and child can be connected directly and balanced.
- If it has two children, the lowest node of the right child must be found to be the replacement, and then the balance must be run.

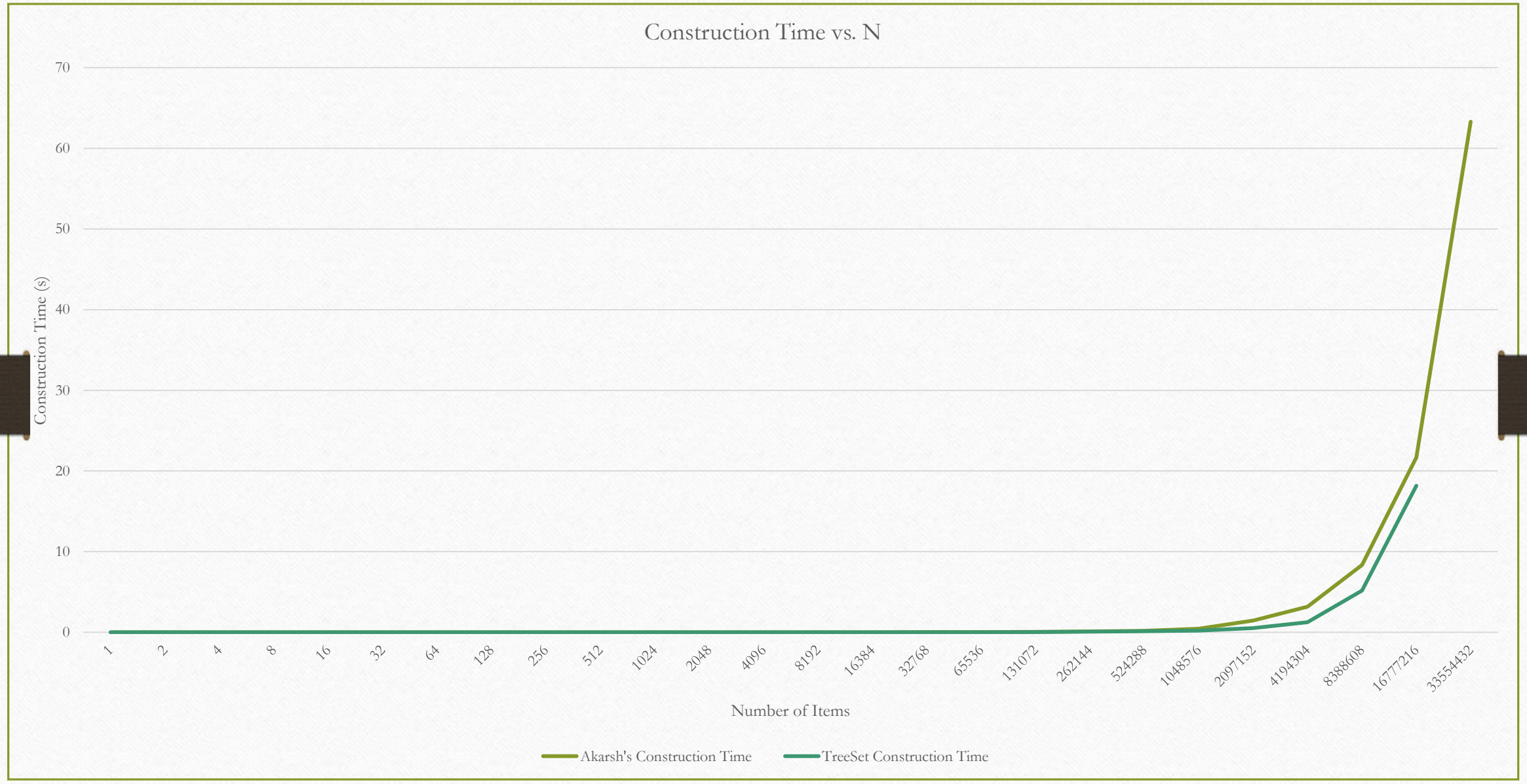
Displaying the Tree

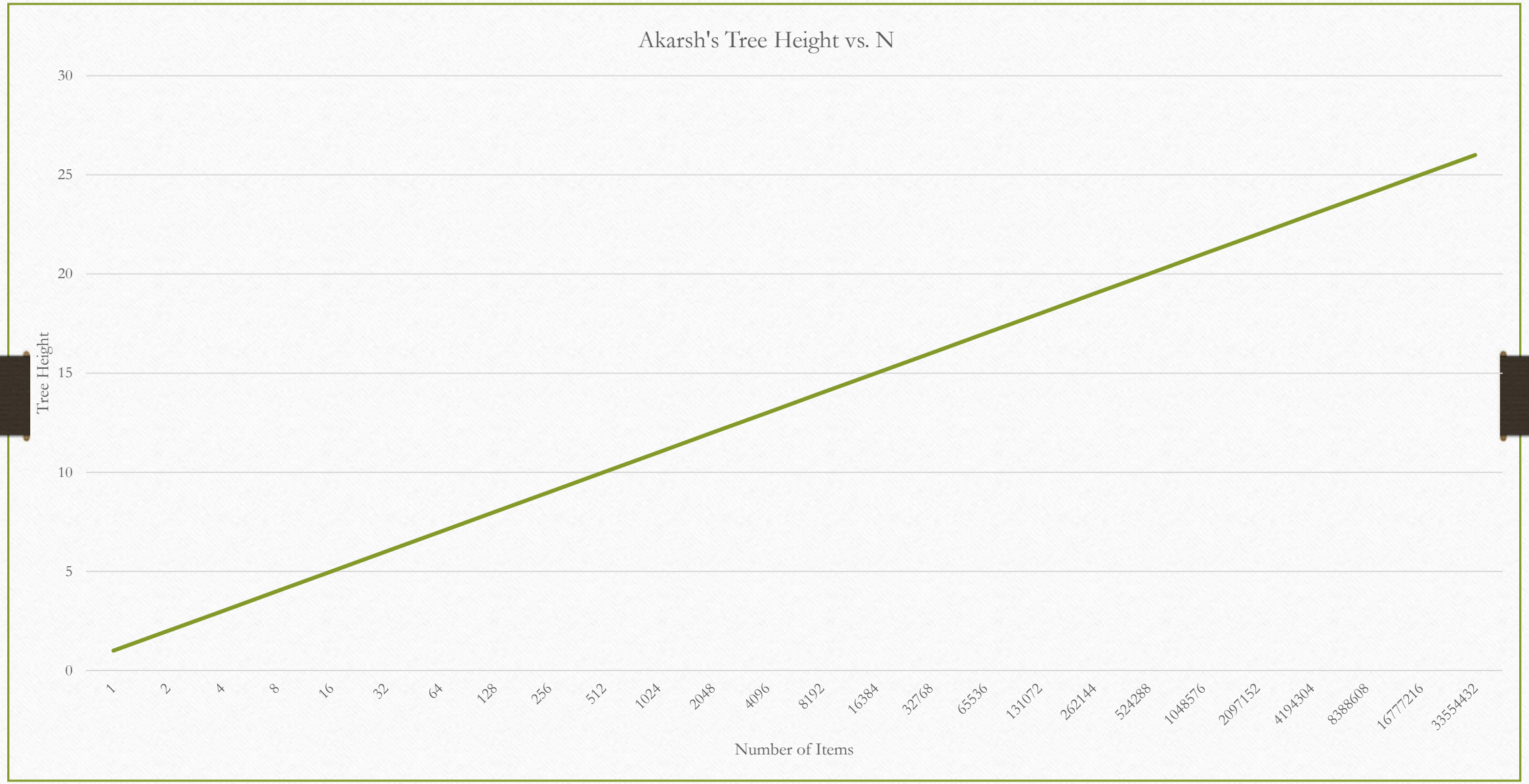
- I made a TreeDisplayer class that extended JFrame to display the tree.
- The following is a picture of the graphics painted onto a BufferedImage.

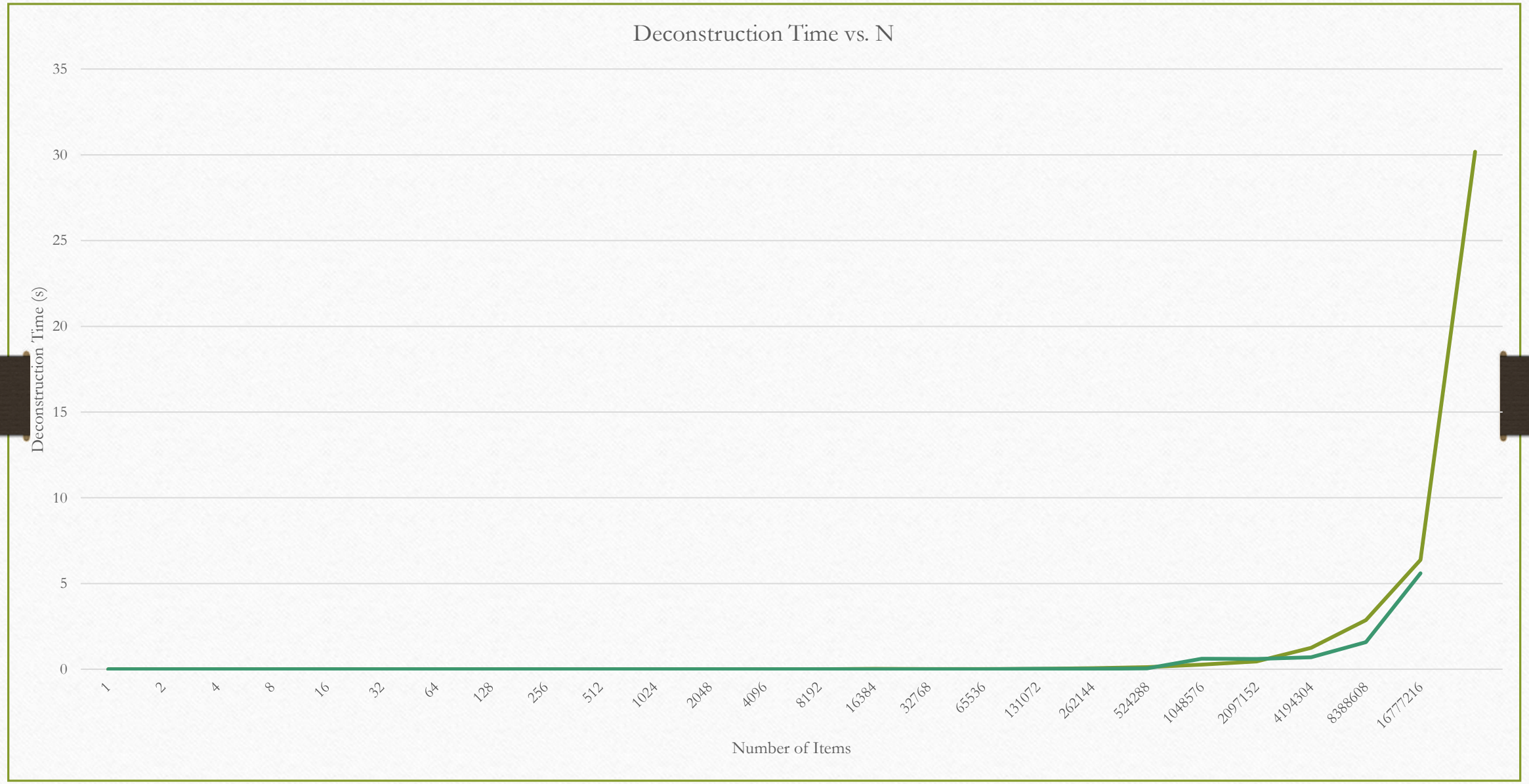


Performance of the Tree

- The performance of the tree was compared to the Java reference implementation – TreeSet.
- The performance was calculated and the graphs made were:
- Construction time vs. N
- Tree Height vs. N
- Deconstruction time vs N
- Where N is the number of items being testing, which powers of 2 (1, 2, 4, 8, 16…)


Construction Time vs. N
70
60
50
40
30
20
10
0
Construction Time (s)
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
Number of Items
Akarsh's Construction Time
TreeSet Construction Time


Akarsh's Tree Height vs. N
Tree Height
30
25
20
15
10
5
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
Number of Items


Deconstruction Time vs. N
Deconstruction Time (s)
35
30
25
20
15
10
5
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
Number of Items

Performance Details

- All of the x-axis on the graphs are exponential growth of doubling the number of items, so the graph is not presented linearly.
- To make it into a linear graph, the log needs to be taken of both axis.
- After this operation is made, the construction and deconstruction time will have a linear relationship to N.
- The height of the tree will have a logarithmic relationship to N.