CSE222 / BİL505
Data Structures and Algorithms
Homework #7 – Report

Cemal BOLAT

My Techniques to Handle the Problem and My Solutions

I divided the given task into two parts:

1. Implementing the random input file generator.
2. Implementing the AVL Tree and performing performance analysis.

Part 1: Implementing the Random Input File Generator

For the random input file generator, I used a Set data structure to ensure that each random symbol was unique and not repeated.
The steps I followed were:

1. Define the range of possible symbols.
2. Generate random symbols and add them to the Set until the desired number of unique symbols is reached.
3. Write the unique symbols to an output file.

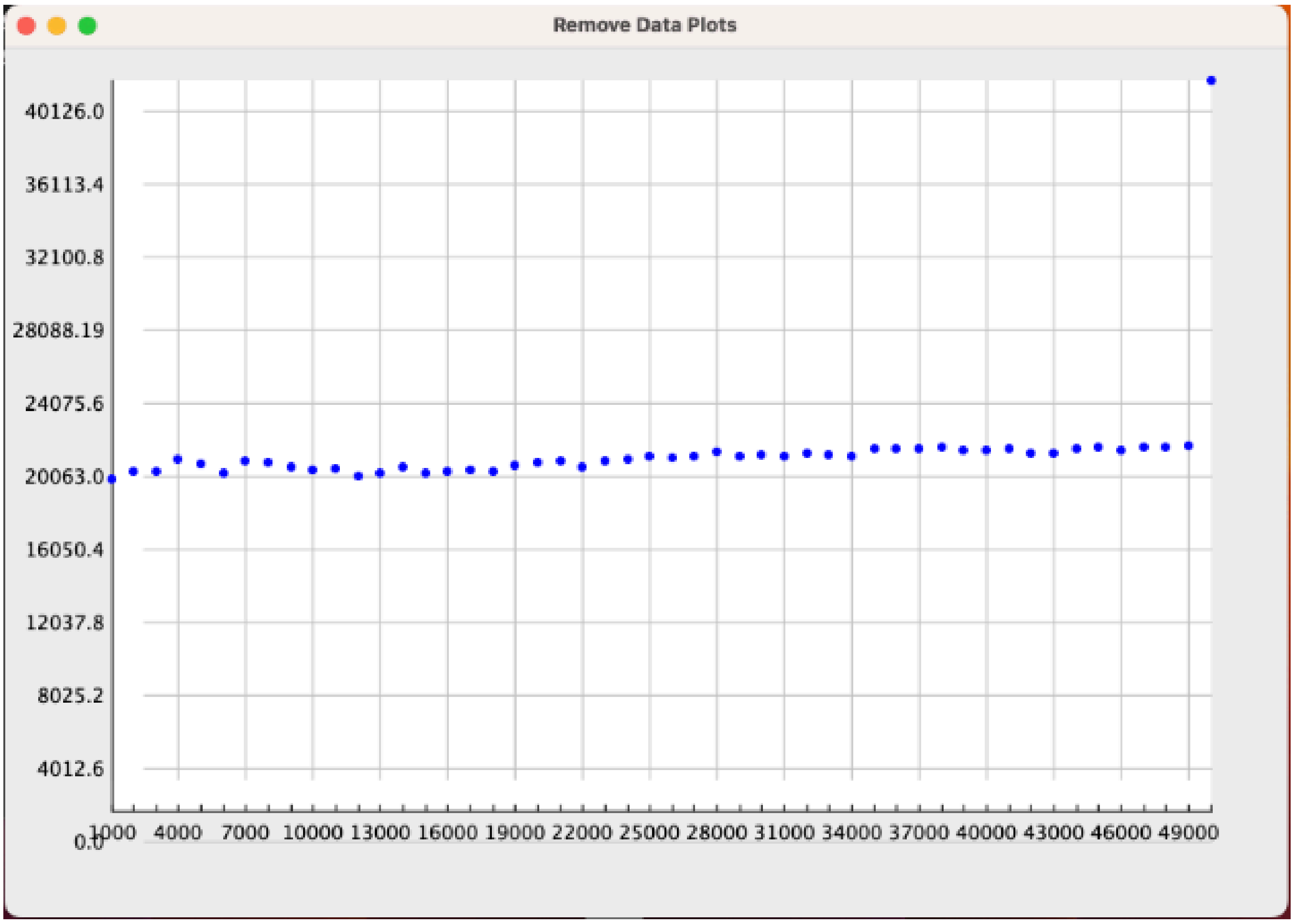
Part 2: AVL Tree Implementation and Performance Analysis

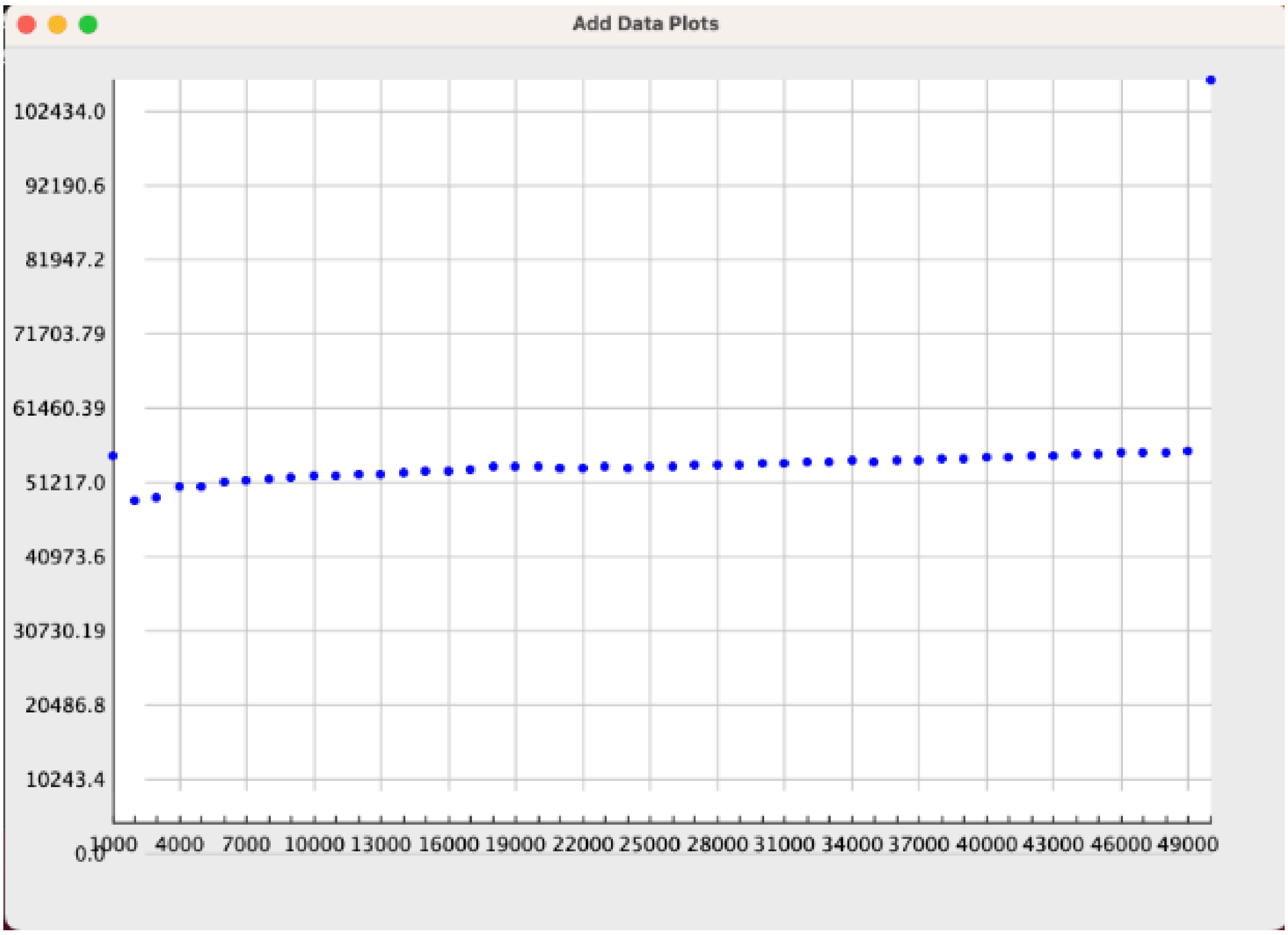
While implementing the AVL Tree, I noted several differences compared to the Binary Search Tree (BST) that we learned in the CSE102 course. The AVL Tree automatically balances itself using the heights of the left and right subtrees.
Here are the key aspects of my implementation:

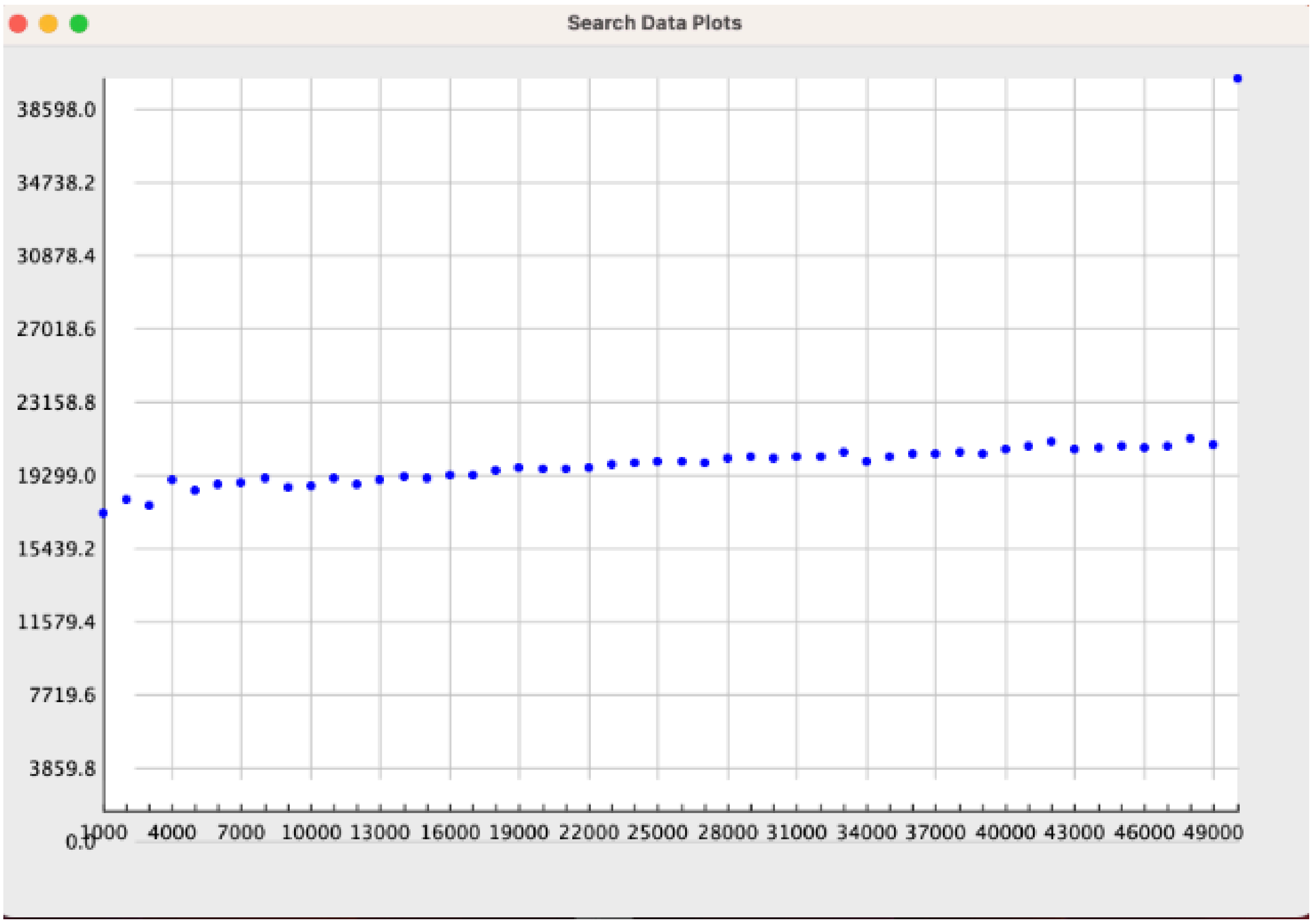
1. **Node Structure:** Each node contains the data, pointers to its left and right children, and an integer to store its height(different from BST).
2. **Insertion:** When inserting a new node, I performed the standard BST insertion and then updated the heights of the nodes. After insertion, I checked the balance factor (the difference between the heights of the left and right subtrees) of each node. If the balance factor was outside the range [-1, 1], I performed the necessary rotations (single or double rotations) to restore balance.
3. **Rotations:**
    a. **Left Rotation: Applied when a right-heavy imbalance occurs.**
    b. **Right Rotation: Applied when a left-heavy imbalance occurs.**
    c. **Left-Right Rotation: Applied when a left-right imbalance occurs.**
    d. **Right-Left Rotation: Applied when a right-left imbalance occurs.**
4. **Deletion:** Similar to insertion, after deleting a node, I updated the heights and checked the balance factor of each node. Performed the necessary rotations to restore balance if needed.
5. **Performance Analysis:** I tested the AVL Tree implementation with different sets of input data. Measured the time taken for insertion, deletion, update, and search operations. The results confirmed that the time complexity for these operations in an AVL Tree is $O(\log n)$

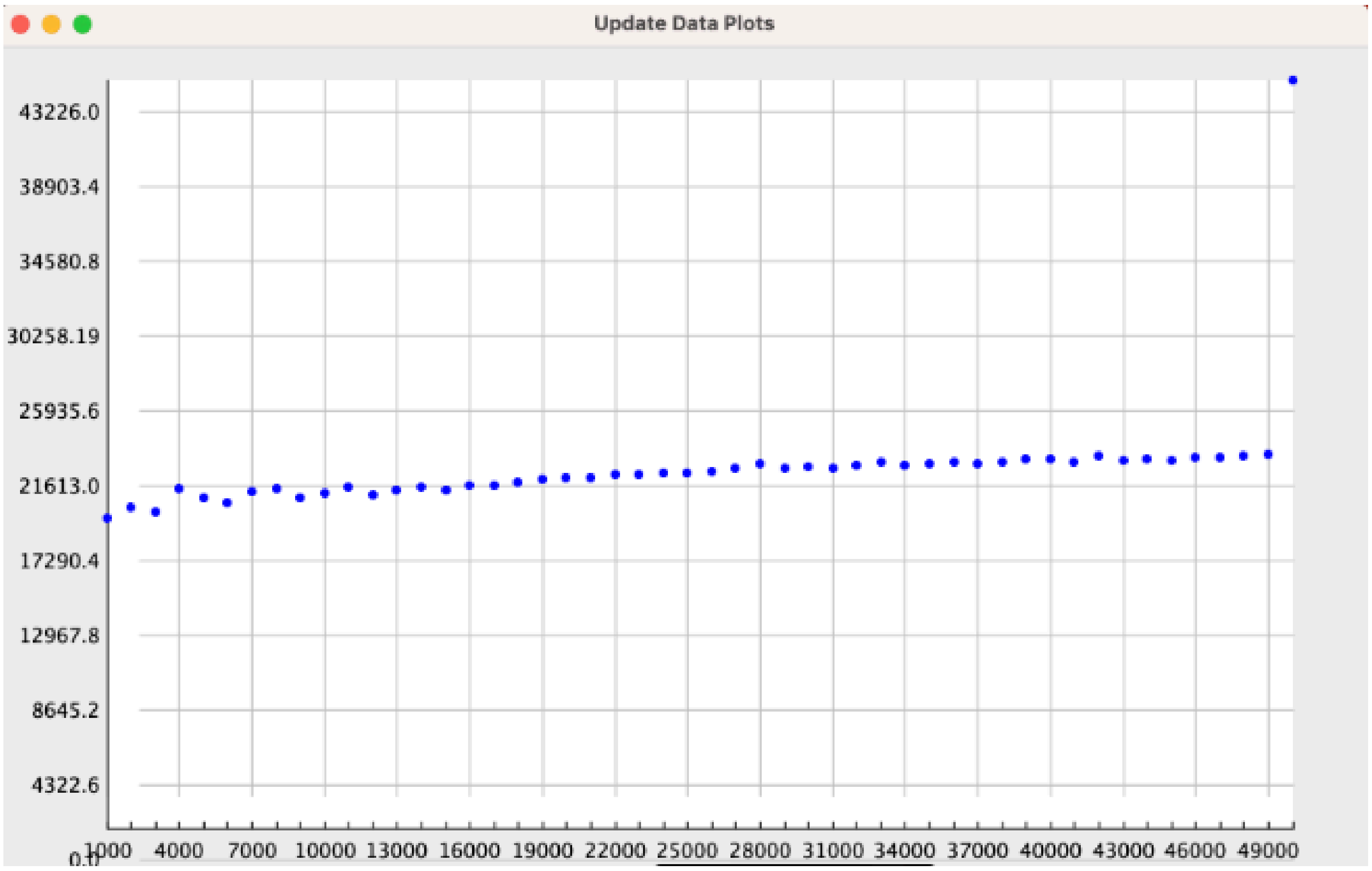
Challenge that I faced during the implementation is plotting the graph because calculating times always suffers on every language time depending on many environmental changes. And the graph is the high part of the window so I add one more element on my graph that is 2 times larger than the last element.

GUI OUTPUT


Remove Data Plots
40126.0
36113.4
32100.8
28088.19
24075.6
20063.0
16050.4
12037.8
8025.2
4012.6
0.0
1000 4000 7000 10000 13000 16000 19000 22000 25000 28000 31000 34000 37000 40000 43000 46000 49000


Add Data Plots
102434.0
92190.6
81947.2
71703.79
61460.39
51217.0
40973.6
30730.19
20486.8
10243.4
0.0
1000 4000 7000 10000 13000 16000 19000 22000 25000 28000 31000 34000 37000 40000 43000 46000 49000


Search Data Plots
38598.0
34738.2
30878.4
27018.6
23158.8
19299.0
15439.2
11579.4
7719.6
3859.8
0.0
1000
4000
7000
10000
13000
16000
19000
22000
25000
28000
31000
34000
37000
40000
43000
46000
49000


Update Data Plots
43226.0
38903.4
34580.8
30258.19
25935.6
21613.0
17290.4
12967.8
8645.2
4322.6
0.0
1000
4000
7000
10000
13000
16000
19000
22000
25000
28000
31000
34000
37000
40000
43000
46000
49000