Andy Franck - CS 163
Program 4 - Efficiency Writeup

Like the programs before, the data structure was assigned, so there was no freedom in choice. The type was given, so we had the freedom to store the collectable data in a class or a struct.

Because the data was the same as the last program, I was able to use the same struct as last time to store my data. Although the information was more complex than the first programs, I could not find any reason to use a class, different to my previous program. The struct was simpler, and I had no issue with reading any of my data. I used a node struct to point to my collectable struct which stored all of the data.

In regards to data structure, I think that if used with different functions, this data structure would be better than the hash table. However, in this case it is no better than the hash table—in fact I believe it is worse—because we are just searching by value and displaying that data. If we focus on lowering collisions in the data with the hash table, I think that would be more efficient than switching the entire structure to a BST.

Because all of our functions directly search for values, we are using all of the benefits of the hash table, and very few of the negatives. We are not looking for “similar” matches, where a BST would be more beneficial because we could search through and divide up the tree based on what is similar instead of just looking for exact matches.

When discussing the memory overhead and usage, I think it is important to understand that the BST uses two pointers in each node. Because we are not taking full advantage of the benefits of the BST, I think that this is not the best strategy for memory usage. I think that because of the low level of collisions in the data when tested in the previous program, we can conclude that the hash table is most likely more efficient in this scenario.

An efficient part of my program is the destructor. Although this is often overlooked, I was pretty proud of mine in this case because I took advantage of deleting after the recursive call. This meant that I was basically deleting in reverse order, so I traversed to the end and then went back and deleted each node as I went. This meant my code was extremely efficient (only 13 lines) because of my recursive call. If I had not done this recursively, it would have been extremely inconvenient because I would either have to find another way to traverse in reverse, or navigate through the entire tree forward and try to delete nodes as I go left and right.

In the same vein of discussion, this was also a way I was able to shorten and simplify my code. Because we were forced to work recursively, I found considerably more efficient ways to display names and types of data. I was able to directly compare the length to Program 3, since the display functions were very similar, and my code was much shorter and easier to read.

I was also able to condense my insert function considerably when I noticed that since I was only adding at the root, I could just add one check and otherwise return the function with the left or right pointer based on how the name to be inserted compared to the name inside of the current node.

The main difficulty I faced (similarly to many other people, I’m sure) was the remove function. I found most of the other functions very straightforward, except this one. Because there

were so many different cases depending on where I was in the tree, I had trouble coding up every single different case properly—as well as testing them. I had to create a custom set of data to properly test each situation more efficiently. I also had trouble with connecting up my nodes because of the how the functions were designed recursively. I still have trouble understanding the specifics of that.

If I had more time to work on the project, I would definitely spend it trying to simplify my remove function. Right now, it is by far my longest and most complex function. While I don't believe I can reduce the complexity of its design, I think it is definitely possible for me to shorten it and design it to be considerably easier to understand, making each case clearer with a more modular design. It was extremely helpful to follow Professor Fant's instructions in the videos, where she describes how she draws out the problems in advance and plans out each condition to code. I think once I refine my planning techniques, I will have a considerably easier time coding up the actual functions, especially ones with many different cases to consider.