Andy Fleischer

Prof. Karla Fant

CS163

28 May 2021

Program 4 Efficiency Writeup - BST

This program was required to store information on various modern C++ libraries, like vectors, templates, and strings as a table abstract data type implementing a binary search tree algorithm. In particular, the tree stored the name of the concept, a description, some example code, and an additional data field of our choice, so I added a link to more information on the syntax. In my implementation, nodes had a left and right pointer to other nodes, as well as a data of type item struct, which is where all the char pointers to the stored data were held. Overall, the binary search tree data structure was usually a good balance between memory and performance, and often made searching through the tree fairly fast.

If I were to start the project over, I probably would not use another data structure. If anything, I would switch to a balanced tree, like a 2-3 tree, but we have not learned much on how to fully implement balanced trees yet. A balanced tree will also use more memory, especially a 2-3-4 tree (7 pointers per node!), but it would have much faster performance (always a logarithmic performance). Although the data sets used for this project were usually pretty small, they did use up a lot of memory storing multiple large strings per node, and more pointers will not help the memory issue. Also, since the trees were not often large, logarithmic performance would barely help performance over even the worst binary trees.

I would likely not switch away from a tree for many reasons. For one, the recursive code used to maintain trees is not much more complex than a linked list, and you are usually saving performance at the expense of a small amount of memory. I believe that the reduction in speed scales much more importantly than the reduction in memory. Also, the data in this structure is not inherently stack- or queue-shaped and has multiple functions to search through all the data, which stacks and queues are not made for. Hash tables are not a bad substitute, since the table ADT means most functions work with the key. However, a hash table would do very poorly for

display range, which a tree can slightly optimize, but a hash table must search through all the data.

Generally, the tree was pretty balanced in terms of memory and speed efficiency. It is not as memory efficient as a hash table, linked list, stack, queue, etc., but it is much more speed efficient than them. Also, it is not as speed efficient as a balanced tree like a 2-3-4 tree, but it is more memory efficient than them.

If I had more time to solve this problem, I would make a few changes. First, I would consider switching to a balanced tree if I had enough time to figure out how to implement the special algorithms. Second, I would find a better way to input, store, and print out the example code. Also, I would add a few more functionalities, like searching for a given name, a copy tree function, and anything else that may be important. Further, I would like to improve my efficiency function since I am not sure if my current calculations properly explain the efficiency of the tree. Additionally, testing with more data (such as from an external data file) would be nice to see how the performance and memory usage are affected, and to see what larger trees look like. As always, many functions could do with better error handling, especially display range, which has many different ways that the given input results in an error. Finally, outside of the ADT, I could always make a better UI that has stronger input checking and looks nicer.