Christian Schmid

Dr. Karla Fant

CS163 - Data Structures

**Program 4 – Efficiency Write Up**

For Program 4, we implemented a Binary Search Tree which stored a collection of terms from the Carrano reading. The terms relate to data structures, and hold information about them, as well as the ADT they’re associated with.

For this particular exercise, a BST was a good choice of data structure. Because we wanted to implement the ability to find a range of items, using a tree over a hash table was required. Even with the extra traversal required for a BST, the O(log2n) performance was excellent in comparison to list data structures – especially when considering hundreds, thousands, or even millions of elements. Furthermore, having the ability to retrieve the elements in sorted order is a definite plus.

The one caveat with speed is, of course, the BST must be efficient. With numerous insertions and removals it seems that BSTs can quickly become lopsided. Balanced trees, such as 2-3 trees or Red-Black trees, would seem to fix this issue. In fact, this makes me wonder if there are any benefits to using a generic binary search tree without any balancing method. There wouldn’t be as much of a performance overhead regarding checking vertices like balanced trees must do, but it seems that the cost of having an entirely inefficient tree does not make such a tradeoff worth it.

With this in mind, if I had the know-how I probably would use a balanced tree to store my data in this case – perhaps a 2-3 tree. The “bubble up” performance overhead wouldn’t necessarily be a problem – losing a few milliseconds due to balancing wouldn’t be noticeable to the user.

Regarding my particular code, I believe I am getting more of a hang of abstracting data. My CharString class (a wrapper for a char array) is pretty robust at this point. It allows comparisons between two CharStrings (==, <, >), receiving and storing input directly from *cin*, printing the array with a prefix and suffix, and returning an immutable char array for functions that require one. These are probably all basic features of the *string* class, except with a lack of operator overloading. Nonetheless, it’s quite exciting to be able to create it from scratch and continually build upon it over time.

One area that I wish I could have spent more time on was the traversal of the BST. I was hoping to implement data-parallelism, especially in regards to determining the height of the tree. Unfortunately I ran out of time, so I didn’t implement it. I also wish, as I usually do, that I could create a way to streamline the test-creation process. The intuitive method would be to implement a console user-interface class that could auto-generate a menu. Associating these selections with methods, however, would require treating a method as an object, which I believe falls into the realm of functional programming. Although I’ve (lightly) experienced this in other languages, I have yet to see how it works in C++.

All in all, this programming assignment was an interesting one. The whole concept of the tree data structure is intriguing – especially balanced trees. Being able to achieve the same search speed as a sorted array, and maintaining sorted order, all the while allowing dynamic growth in size, is an excellent combination. All the same, the entire process of insertion, removal, and retrieval is a complex process (again, especially when considering balanced trees). However, with complexity of implementation comes experience and greater understanding of how memory allocation, pointers, and the like work. With this in mind, I felt I learned a good deal from this program.