Andy Fleischer

Prof. Karla Fant

CS202

14 August 2021

Program 3 Efficiency Writeup

For program 3, we had to create a program to keep track of athletes and Olympic sports to practice implementing operator overloading. We needed to provide support for three different events (some individual and some team sports) as well as information on athletes, such as past olympics and medals won. I chose baseball, swimming, and running for my three sports, each being their own class inherited from the base class sport. Each sport implemented the insertion and extraction operators for display and read, as well as the equality operators for LLL removal. I had to make an extra protected read function for the sport class so the children could read in the data for the sport, as well. Next, I created an athlete class that had the data for an athlete, insertion and extraction operators, and comparison operators for the tree. The Athlete class had an LLL of sports, where each node of the LLL had a next pointer and a void pointer to point to a sport at runtime. Finally, BNode derived from Athlete and was a binary search tree node for the BST class.

Overall, I believe this approach was both object-oriented and effective. There were a lot of classes for managing the data structure but it did help to separate tasks out by class and not try to have one giant “BST of LLLs” class. As always, the Node classes had to have getters and setters, but I still believe that it was object-oriented since Node classes are really just packaging classes anyway. Plus, it saved much pain from having to create a million wrapper functions or fit all of the data structures in one class. The rest of the program was definitely object-oriented in how the sports all derived from the base Sport class. I could have maybe implemented dynamic binding here, but there was not too much of a common interface among the classes, so I decided to just use void pointers in the nodes to point to a certain sport at runtime.

Going through the process, I did not ever need to make drastic changes since I did a lot of planning before implementing (measure twice cut once). In program 1, I needed to make big changes to the main method to implement the data structure, but using void pointers made

that much easier. In program 2, I ran out of time trying to fix my main class to implement the data structure, so this time I planned early on how my actual main would look. In the end, my main was much cleaner since I did not need three different data structures, each with their own menus with different wrapper functions to call the functions of their respective classes. I just had one overall BST to add, remove, display, remove all, and search from.

Time to talk about efficiency in this project. In terms of speed efficiency, this data structure (a binary search tree of linear linked lists) was definitely faster for traversing than the previous projects' data structures (array of linear linked list, circular linked list, and doubly linked list). The binary search tree is a great, simple way to cut down runtime performance from O(n) time to at worst O(n) time and at best O(lg(n)) time. Of course, since the tree was not always balanced and each node was a linear linked list (which will have some performance issues), it was not always O(lg(n)) time. However, it should on average be faster than the previous data structures. For memory efficiency, this is where a BST lacks. Since each node has a left and right pointer (except the LLL nodes), it does take a bit more memory than just a CLL or LLL. However, it does take as much as a DLL, but it is a bit more speed efficient, so I think that was a good tradeoff to make.