Program #3 Review

Written by Alexander DuPree

Data Structure Performance

The primary data structure of Program #3 was the binary search tree. The BST worked great in allowing the user to add different food item objects to the menu dynamically. However, the largest drawback of the BST is that it can become unbalanced. If the user attempts to add multiple food items objects with similar caloric counts the tree can become unbalanced towards one side. This imbalance will reduce the runtime performance of the BST to something like that of a linear linked list. With more time and experimentation an implementation of a balanced search tree like the 2-3-4 tree or red-black tree would have been more appropiate for this application. Furthermore, because of the recursive nature of the tree, it was difficult to traverse the tree at the clients leisure and examine/modify each food item object individually. To remedy this problem a traversal queue of food item pointers was built to hold the indexes of each item in the tree. The iterators would then pop the pointers from the traversal queue, allowing the user full forward iterator support.

Object Oriented Design

The design of program #3 emphasized the use of operator overloads to create a clear and accessible experience when working with class objects in the system. As such each class had relevant operators overloaded in an intuitive way. For example, comparison operators for the Food Item hierarchy would compare caloric values, iterators contained proper incrementation operators and etc. Furthermore the BST used in this program utilized dynamic binding and inheritence to create a heterogenous collection of food item objects. To instantiate the different types of food item objects factory objects were used to gather attribute data and add them to the BST.

GNU Debugger Discussion

The GDB was an invaluable tool during the testing and production of program #3's code. I also utilized the catch2 unit testing framework to establish a suite of tests for my program. This test suite coupled with the GDB allowed me to catch and pinpoint errors in my code almost immediately. If a certain test failed after a lengthy refactor I could fire up GDB, set a break point at the test and step through each line of could to identify the error. The debugger, coupled with unit tests, was able to exponentially increase my productivity as a programmer as I spent more time writing code and less time debugging. The GDB was also important in affirming that my program was utilizing dynamic binding correctly, as I could step through each line of code through the debugger and ensure the correct function was being called from the hieriarchy.