Matt Stevenson CS202
5/29/2020
Program #4 Efficiency Write Up

Program 4 has been an enjoyable challenge for a number of reasons, most of all because of the use of a new language. Fortunately, there are enough similarities between Java and C++ that the transition was mostly painless. Additionally, I found that without some much syntactic rigidity I was able to focus more on the problem solving and less on making sure I had exactly the right syntax. As a result, the programming process was in many ways easier and I was able to produce a decent end result. That said, given more time there are a number of changes I would like to implement.

My data structure is sound, and I was happy with my AVL tree. I was initially intimidated by this new tree type, but ultimately I found it easier to work with than other structures I have used in the past. I chose the AVL because I read it was better for searching if a bit slower when adding/removing. Since I knew I would not be doing any removal, I thought faster search would be better; in real life, I would want to give speed preference to users searching for services rather than businesses adding new services to the database.

I also like how the AVL works just like a normal BST, and the only difference is a cool rotation mechanism to rebalance the tree as needed. I found this operation much more intuitive than the splitting and shifting found in other balanced trees. Beyond my data structure, I am mostly happy with the design of the rest of the program. I managed to keep everything mostly segregated in terms of object responsibility, though there are a few awkward crossovers that I would amend given more time. For instance, I have my read form file functionality in my Client class, while the read out is in my Tree class. The Tree class also handles a couple methods that might be better suited in the node derived classes.

Regarding the program requirements, I was able to implement the required overloading via a set_cost function; this was implemented in the base class, and again in the two derived classes with different signatures. I also made use of dynamic binding throughout the program, using base class references for my Service derived objects as well and others like my Link List object (which is derived from Node). For example, use of dynamic binding is evident in the display functions in my Service hierarchy; they have the same name, return, and signature, but output different data depending on the class (seen in Link Link::display()). Lastly, I made use of the super keyword in my constructors for my Service hierarchy.