Bradley Manzo
May 6th, 2022
Prof. Porquet
ECS 36C A03 SQ

ECS 36C Programming Assignment 3 Report

In order to implement the desired functionality of the binary search tree, I adapted the structure of Professor's bst.h code provided in lecture. The ADT is constructed from a tree of nodes, each containing a key to be ordered with, a value to hold, and pointers on left and right children. When a tree is instantiated, a unique pointer root node is created and when new nodes are inserted, the node's key is compared with the root. If it is greater than the root, it is placed on the right, and vice versa for the left. Upon insertion, the location of the node is maintained by the parent node using a pointer. Because the only node with a specific name and instance is our root node, we must traverse through the tree beginning at the root in order to find the desired node. Due to the rules placed on our insertion, traversing the tree to find a desired key is of time complexity O(logn). This is because as we go deeper into the tree, the input size is halved for each operation.

Functions such as Insert(), Remove(), MaxKey(), and MinKey() are implemented recursively, meaning that the desired node is located by calling the function within itself the necessary amount of times, and recursing back through once found. Functions such as Get(), ContainsKey(), CeilKey(), and FloorKey() are implemented iteratively, meaning that the desired node is located through a series of conditional statements as long as the node isn't a leaf. GetCeil() returns the least key greater than or equal to the provided key using iteration. When moving left, the current node is remembered as the most accurate ceiling, and when moving right the new node is checked to see if it is closer to the provided key than the remembered ceiling, and vice versa for FloorKey(). ContainsValue() is the main exception as finding the unsorted value necessitates a complete traversal of complexity O(N) which I implemented using level-order.

For the AnitaB.org program, the treemap.h header is included to implement the desired functionality using the BST implementation from part 1. Rich(), Cheap(), and Who_Amount() wrap MaxKey(), MinKey() and Get() respectively; while Who_Plus_Amount() and Who_Minus_Amount() use CeilKey() and FloorKey(). Since Ceil and Floor return the closest or equal donors, we can instead plug in key +1 or key -1 to find the next closest donor. All() works in the same way, iterating over the entire tree of donors using Ceil(key+1) and having key remember the most recent donor. The program begins asking the user for a file name and a command. The file is opened, parsed, and inserted into the tree similarly to programming assignment 1, and the program counts the amount of arguments to detect which command the user wants to call on the resulting tree.