Programming 2 Assignment

Gerald Hu, Aaron Skouby

CSCE 221-200

March 11, 2016

Introduction

The purpose of this assignment was to further understanding of data structures and implementations that were discussed in class. Specifically, the project explored implementing a map ADT using a binary tree. The performance of the operations of the map were analayzed for both a normal binary tree implementation and an AVL binary tree implementation.

Implementation Details

The map was implemented with a binary search tree, using the AVL method of self-balancing. The tree had node data members, a size_t "sz" which kept track of the size of the tree, special "leaf" nodes at the end of each branch (null pointers), and a special "superroot" node whose left subtree was the main tree (this superroot's parent was null). Each node of the tree stored pointers to its parent, left child, and right child, as well as storing a pair called "value". This pair was the main data element of the node, storing the key/index in its first position, and the data value in its second position; pair used the standard template library.

The map had two erase functions, one which took a key, one which took an iterator for position. These would take a key or iterator and find the desired node, and would get a pointer to it. Then, it would pass the node pointer to a helper function eraser(node* n), which would find the node (or otherwise do nothing), remove it, place the child in the proper position, and return the next in-order node. Insert functions were implemented similarly–two insert functions, one taking a key, one taking an iterator, and both being based on a helper function inserter. The inserter function would take a key and value, and would either construct a new node and add it to the map; or, in the case where the key already existed in a map, replace that key's value with the new value.

The map's size was automatically adjusted by the helper functions eraser() and inserter() when needed. Both eraser() and inserter() called rebalance() to fix tree imbalances after doing their respective operation. Rebalance() would scan up the tree to find imbalances, and call the restructure() function on them, making appropriate left or right rotations.

Timing.cpp was essentially unchanged from the code provided by the instructors. The timing function was already implemented, and for the most part our tests simply changed the parameters of the function. This timing function relies on high_resolution_clock; given a map and a number, the function would repeatedly push random elements into the map the specified number of times.

Theoretical Analysis

Experimental Setup

Timing tests were conducted using the provided timing.cpp, compiled with the provided makefile's commands. Compilation was done on the "linux.cse.tamu.edu" server, with G++ version 4.7.3 (SUSE Linux) (found via g++ –version). Compilation was set to the C++11 standard, with the -G flag enabled and O2 optimization level, warnings set to -Wall -Werror (all warnings treated as compilation errors), and dependencies flagged with -MMD (auto-generate dependencies).

Tests were run on the "compute.cse.tamu.edu" server, which runs Arch Linux x86_64 version 8.12 (found via arch –version and lsb_release -a). This server has 99026668 total kilobytes of RAM (found via free). It uses Intel Core i7-3970X CPUs (2 sockets, 8 cores per socket, 2 threads per core), with a clock speed of 2000 mHz (found via lscpu). Each core has a Xeon E5-2650 processor (found via lshw –short).

Timing functions output timings for input sizes that were powers of 2, starting from 2 itself, and ending at a maximum size specified by the user. Each step of the timing was repeated 10 times, and the average of each result taken. Linear height n inserts went up to a maximum input size of 32768; logarithmic height n inserts went up to a maximum input size of 4194304, and random n inserts went up to a maximum input size of 1048576.

We ran timing.o twice, first on a tree with AVL rebalancing, and then on a simpler binary tree with no rebalancing method, and compared the performance of the two trees.

Results and Discussion

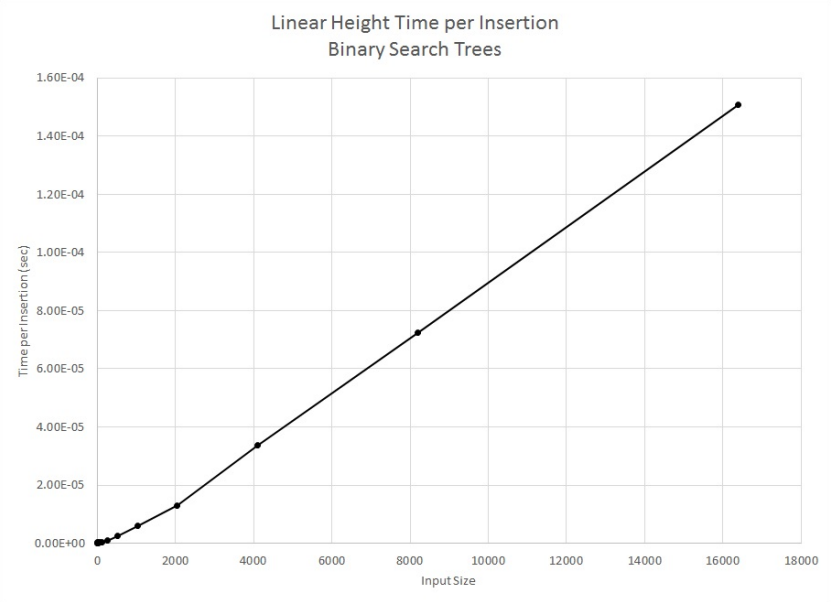

Figure 1: Graph of the Time Taken per Addition for Different Input Sizes for a Linear Order Added Binary Search Tree

Conclusion

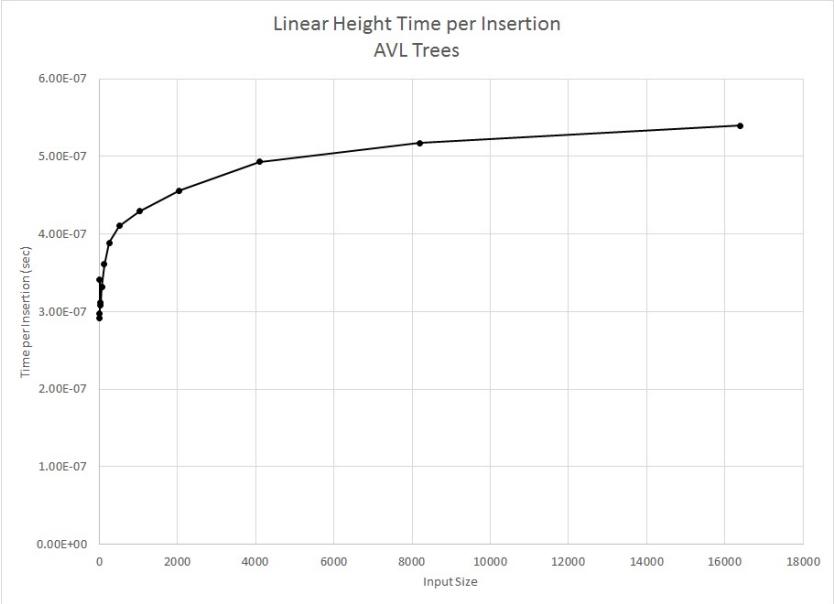

Figure 2: Graph of the Time Taken for Different Input Sizes for a Linear Order Added AVL Tree

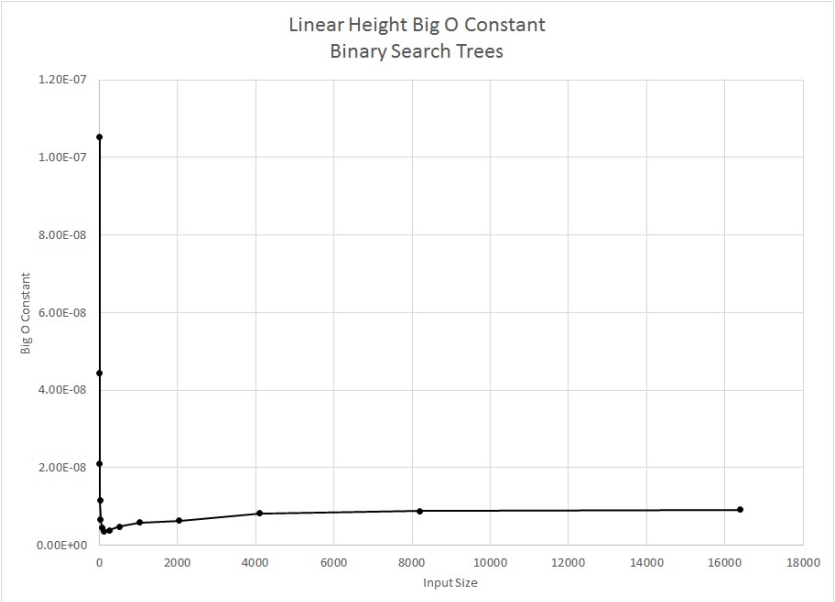

Figure 3: Graph of the Big O Constants for Different Input Sizes for a Linear Order Added Binary Search Tree

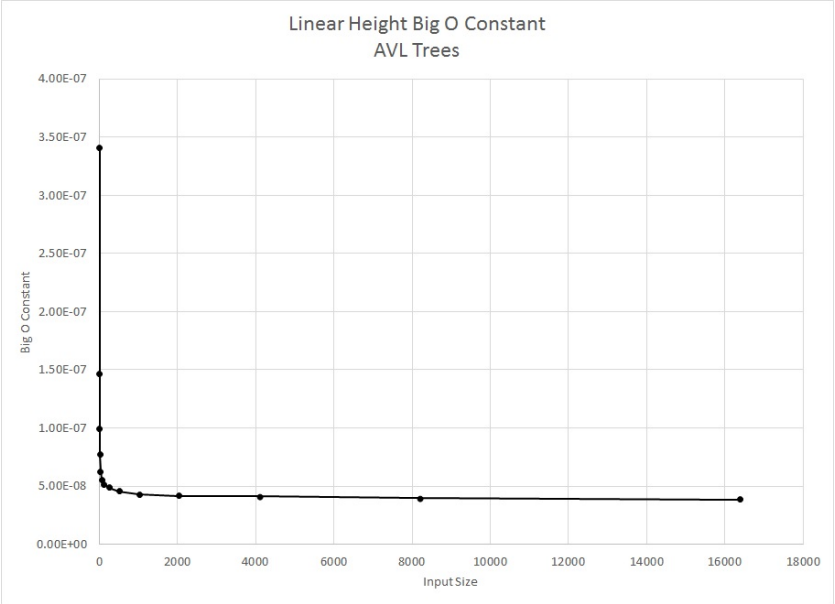

Figure 4: Graph of the Big O Constants for Different Input Sizes for a Linear Order Added AVL Tree

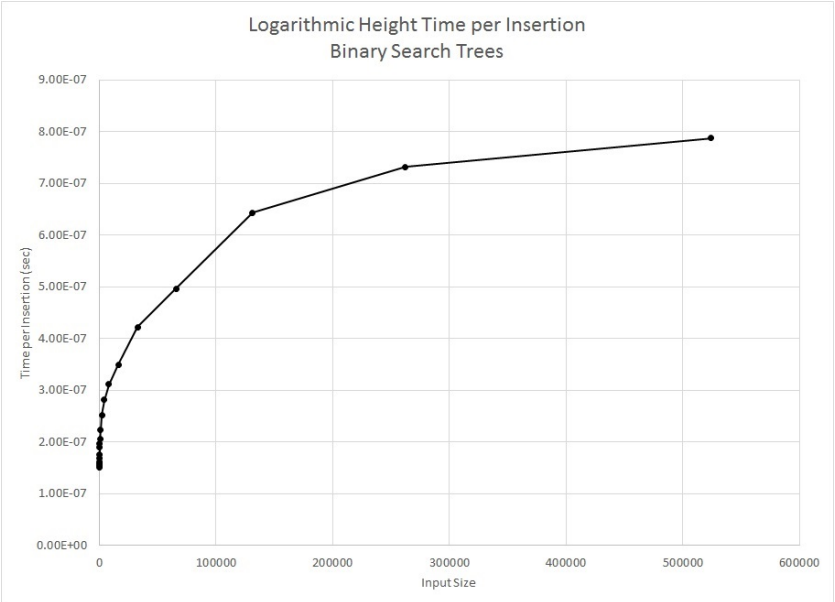

Figure 5: Graph of the Time Taken per Addition for Different Input Sizes for a Logarithmic Order Added Binary Search Tree

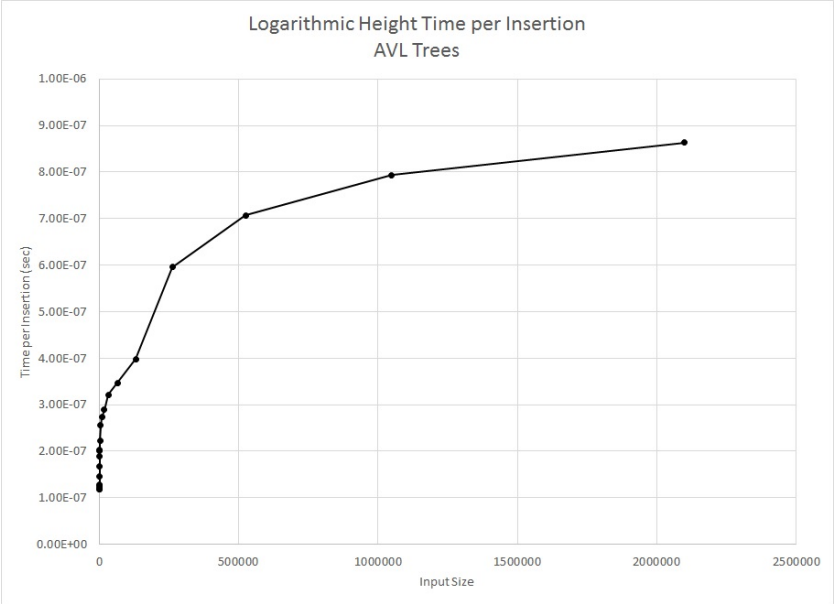

Figure 6: Graph of the Time Taken for Different Input Sizes for a Logarithmic Order Added AVL Tree

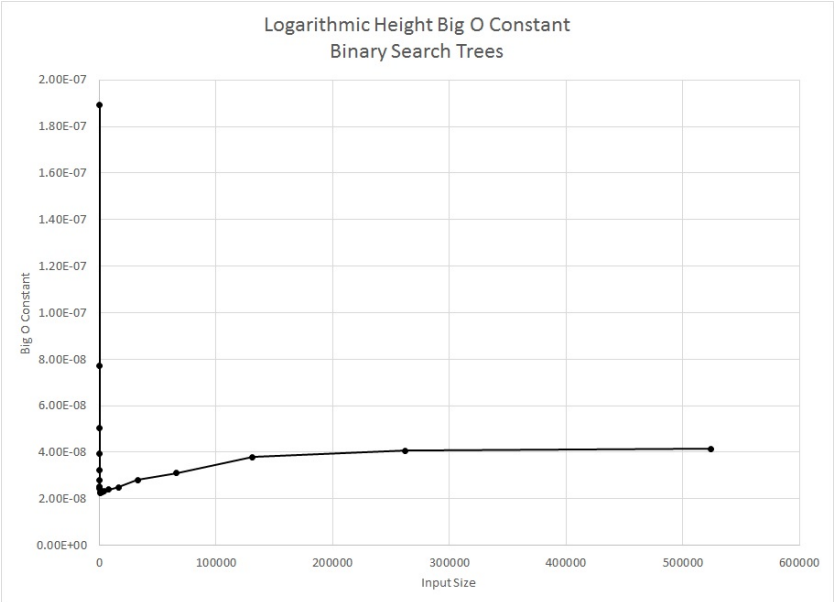

Figure 7: Graph of the Big O Constants for Different Input Sizes for a Logarithmic Order Added Binary Search Tree

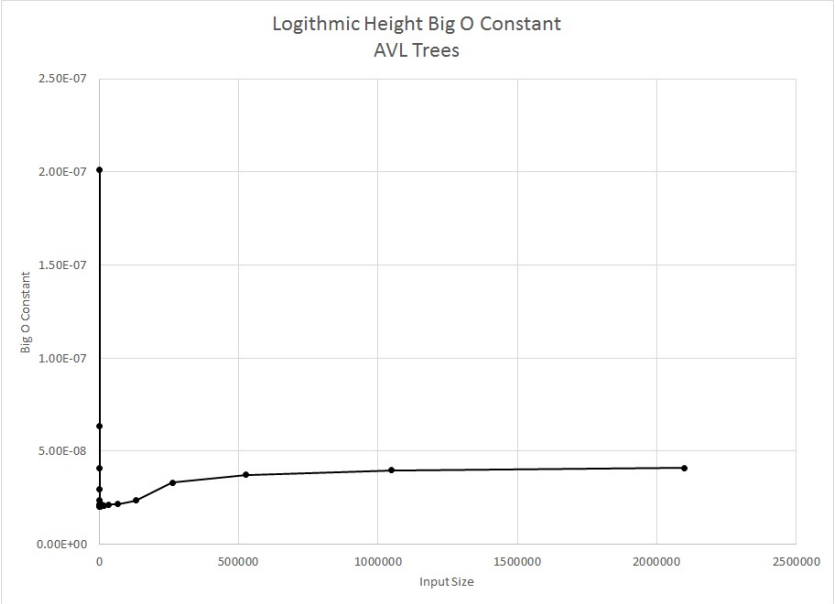

Figure 8: Graph of the Big O Constants for Different Input Sizes for a Logarithmic Order Added AVL Tree

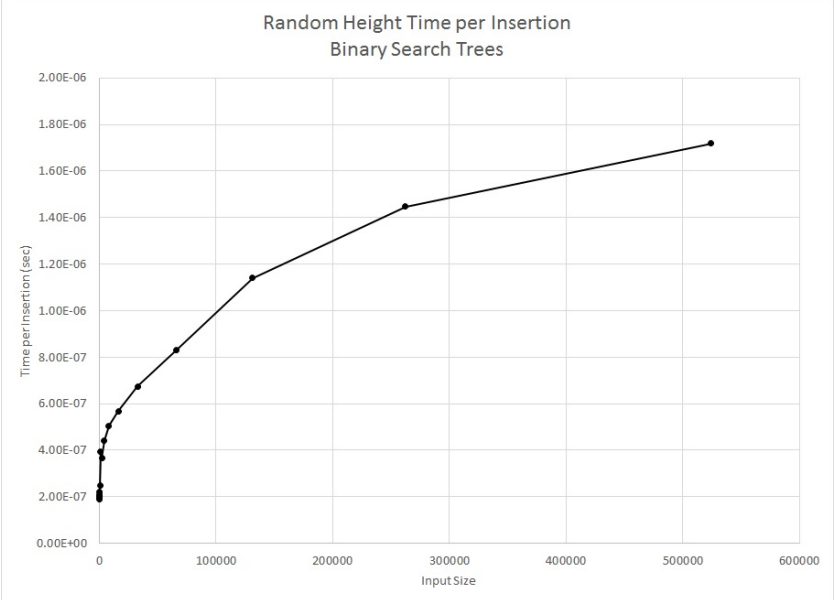

Figure 9: Graph of the Time Taken per Addition for Different Input Sizes for a Random Order Added Binary Search Tree

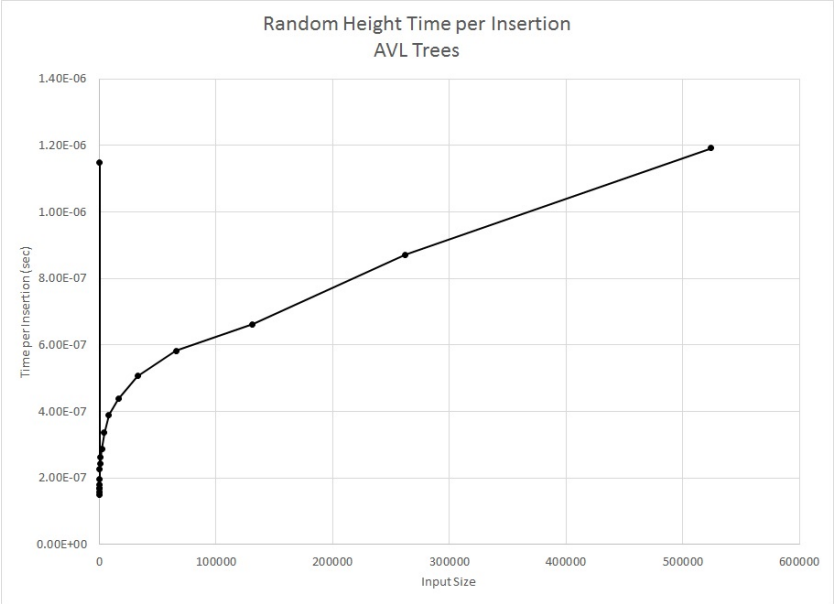

Figure 10: Graph of the Time Taken for Different Input Sizes for a Random Order Added AVL Tree

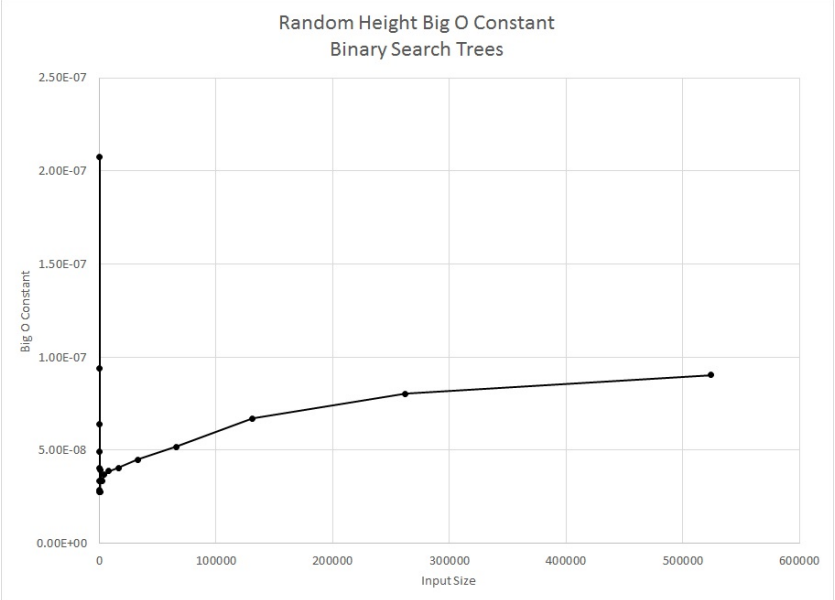

Figure 11: Graph of the Big O Constants for Different Input Sizes for a Random Order Added Binary Search Tree

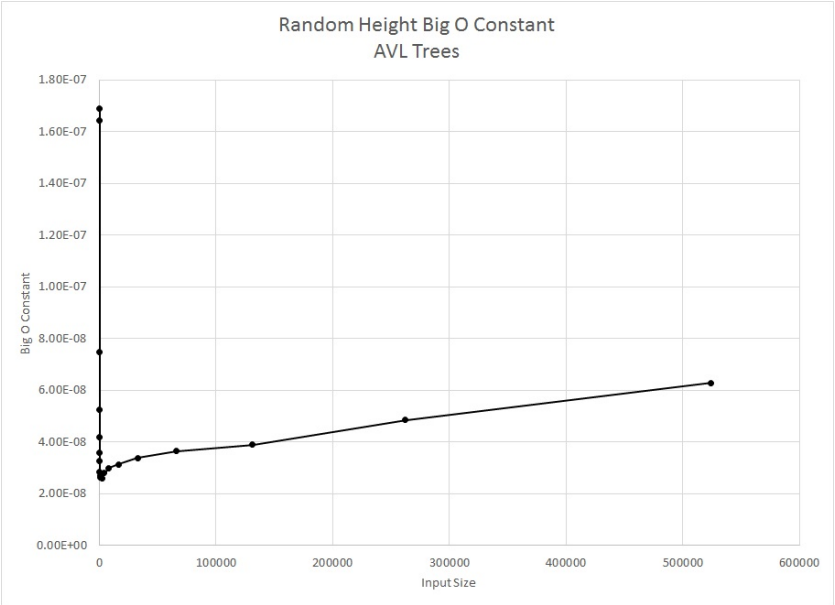

Figure 12: Graph of the Big O Constants for Different Input Sizes for a Random Order Added AVL Tree