Chelsea Valentine

Friday, November 20, 2015

Project 6: Performance Analysis

Before making any changes to the data structures used, I timed the performance of the program to create a constant for all of the other data structure changes to be compared to. I ran the program 150 times and averaged the results. I used a large number of trials in hopes of minimizing random error.

For the 2014-2015 collisions, the average results were 1,175,156,786 nanoseconds for reading and storing data, and 51,179,486 nanoseconds for computation of results. For the 2013-2015 collisions, the average results were 2,213,798,846 nanoseconds and 82,933,326 nanoseconds for data processing and computation of results, respectively. For the 2012-2015 collisions, the average results were 3,384,401,080 nanoseconds for reading and storing the data, and 113,354,393 nanoseconds for computation of results.

My first decision was to switch out all of the array lists with binary search trees, since binary search trees take O(logN) time to search in the worst case scenario. In comparison, array lists could take O(N) time if we search linearly through the list. Since binary search trees are sorted, I figured that we could just easily look at the first or last elements to find the least or most, respectively (assuming that it is sorted from least to greatest). This way, we would get much faster accesses. And because order mattered in this case, hash tables or queues weren't viable options.

All of the changes that I decided to make were from ArrayLists to BSTs. The only place where I kept an ArrayList is in the file data input part of the main method. This is because the current order of the file is important in telling us which column each data point belongs to.

One of the problems with using a TreeSet was that since it's a set, any ties in least collisions between zip codes were not recorded, since each zip code list was compared by number of collisions rather than a unique identifier. Additionally, switching the list of ZipCodeLists from an array list to a BST meant that I had to switch two methods from dividing by the total number of collisions, to by the total number of zip code lists. This is because TreeSets are sets–and there are no duplicate entries. Because there are no duplicate entries in both cases, this improved performance since there were less elements to iterate through.

Consequently, I got the following results when running the program 150 times at each output level. For the 2014-2015 collisions, the average results were 1,188,803,006 nanoseconds for reading and storing data, and 1,533,413 nanoseconds for computation of results. For the 2013-2015 collisions, the average results were 2,225,589,220 nanoseconds for reading and storing data, and 1,653,400 nanoseconds for computation of results. And, finally, for the 2012-2015 collisions, the average results were 3,278,840,626 nanoseconds for reading and storing data, and 1,706,166 nanoseconds for computation of results.

To conclude, for the 2014-2015 data set, the program using BSTs was 33.376x faster at computation. For the 2013-2015 data set, it was 50.159x faster. And finally, for the 2012-2015 data set, it was 66.438x faster at computing results than the constant program. The changes in reading and storing data are too insignificant to be accounted reported.