Thomas Ricketts

20th November, 2014

CSE 274

Assignment 5

We were assigned to create a program that allowed us to input four types of data structures; Linear Hash Table, Chained Hash Table, Binary Search Tree, and Red Black Tree, and to test their methods and discover how long it took to complete their methods using a sequential input of numbers, and a pseudo-random input of numbers. Each of the data structures went through 50,000 add, find, and remove functions 20 times and an average time was calculated of those 20 times for each of the functions we created in the class. The return type of all of these methods were integer, and there were 6 in total. 3 for the sequential add, find and remove, and 3 for the random add, find, and remove. In total there were 24 averages computed, and this paper will show the weaknesses of these methods, as well as the strengths.

The first data structure to be brought up will be the Binary Search Tree. This is a very strong data structure that is used in many real world applications in many different variations of the class. One of the largest strengths I would have to bring to light from observing the random functions data is that all 3 of the functions completed relatively fast. The random add function took 20ms, random find took 10ms, and the random remove took 20ms on average to execute. This is due to the fact that binary trees are organized in a way that it is easier to determine which node a value is stored. Binary trees add things to the left of a node if it is less than the root or node it is visiting, and to the right if it is greater. If it encounters a node that doesn’t contain the value the user is looking for, it iterates the process. If a tree is fairly well balanced, then the complexity should most certainly be O (log (n)).

The weakness discovered was most expected, but now verified due to this testing. When the sequential functions were executed, the add function yielded 11080ms, find yielded 9073, and remove yielded 10ms on average. This is because the numbers added were sequential. Therefore, all the numbers would have been stacked to the right of the root, rendering the structure of the tree pointless. It would have looked like a straight line to the right in a diagram. The 10ms time to execute the sequential remove on average was due to the fact that when you removed the number, it was next in the list. So this means the computer had to spend almost no time searching for the value.

The Linear Hash Table on the other hand was very quick in all ways of inserting data. None of the functions for the randomized insertion or the sequential insertion took longer than 10ms on average. I am assuming if we did further testing with the different types of open addressing such as linear probing, double hashing, and quadratic probing, it would yield different results.

The downside to this structure is that with larger amounts of data, collisions might become unavoidable. If we tested with a larger input of numbers, we might begin to see a larger rise in time to take to complete as the items inserted increase. Also, we might find out while we are testing all the other types of open addressing, that some methods might be more ineffective than others.

As for the Chained Hash Table, the execution time on average was very slow compared to the other data structures. The 3 random add, find, and remove functions took 1120ms, 1540ms, and 1150ms respectively. The sequential add, find, and remove functions took 3480ms, 860ms, and 1620ms respectively as well. A plus that I can pull out of this data is that at least it takes a consistent time to complete all of the functions. It might have been faster than the other data structures if we tested it instead with objects rather than integer values.

The down side is very noticeable when taking a glance at the data. Nearly all of the methods took over a second on average. This is poor performance compared to the other data structures. This may be due to the type of data that is entered, or the type of open addressing that was implemented. These factors could potentially have a dramatic effect on performance.

The final data structure tested was the Red Black Tree. The performance was very similar to the Linear Hash Table. The random add, find, and remove functions executed on average in 20ms, 20ms, and 10ms. The sequential add, find, and remove functions executed on average in 20ms, 30ms, and 0ms (wasn’t actually 0, just a fraction of a millisecond). The averages for the sequential functions were different from the Binary Search Tree’s sequential functions because Red Black Trees are self-balancing. Since this occurs, the sequential numbers don’t form a straight line in a diagram if it were drawn out, but would appear more like a tree. This would make it very easy to navigate.

Weaknesses of a Red Black Tree appear non-existent at first, but when you look at the way they are set up, it becomes a little more obvious. Red Black Trees must compare the data in order to sort it into the tree. If you are inserting objects that cannot be compared, then this structure is useless and cannot be used. Otherwise, this data structure seems like one of the best on the list of data structures tested.

Exercise 9.5:

First off, let us define what the pushBlack(u) and pullBlack(u) functions do. The first takes in a black node that has two red children and reverses their colors. So the root would become red, and the children would become black. The second listed function does the exact opposite. 2-4 trees can store multiple values in their nodes, but it is dependent on how many children they have. A parent node with one element must have 2 children, a parent node with 2 elements must have 3 children, and a parent node with 3 elements must have 4 children. Now, the pullBlack(u) function change the 2-4 tree simulated by the Red Black Tree by taking the numbers in the u node and distributing them among the children nodes. The pullBlack(u) function changes the 2-4 tree simulated by the Red Black Tree by taking the u node’s children’s extra values and absorbing them into u, being the parent node in this situation.