Sam Brendel

5/16/2017

TCSS342, Data Structures

Assignment 3, brendel\_pr3

SVN revision # 40

**Summary Report on Tree Performance**

**Introduction**: No stack overflows were experienced with my recursive implementations. The following graphs represent performance data, and their respective conclusions shed insight.

**Note:** The act of printing to the console involves several other Java sub-routines who’s overhead is [much] more costly than the actual process of assembling a tree in this experiment.

**Graph 1**: Insertion times using text files containing sorted words (no duplicates):

Text files used:

* s-sorted.txt 25,033 words that all happen to start with the letter ‘s’.
* dict\_polish.txt 109,852 words in a Polish Dictionary.
* dict\_danish.txt 519,986 words in a Danish Dictionary.

Data Structures used to process these text files:

* Binary Search Tree O(log(n)) for insertion, access, and remove.
* AVL Tree O(log(n)) for insertion, access, and remove.
* RedBlack Tree (Java’s implementation) O(log(n)) for insertion, access, and remove.
* Splay Tree O(log(n)) for insertion and remove.

**Conclusion**:

This graph has an over-all appearance of a triangle, but is only because of being a line graph. A better depiction would be a bar graph. It would appear either an anomaly occurred during the 3 times I processed the polish dictionary with BST and AVL, or somehow the put( ) algorithm is more efficient in a BST and AVL tree that already contains approximately 109,000 nodes. This is depicted by the incline from 25k to 109k, and a decline from 109k to 500k. I do not believe this is due to caching since all words put into the trees are unique. RedBlack and Splay show a more linear rate in comparison, but more in alignment with O(log(n)) as seen with the more flattened appearance. Not having duplicates should not matter because the same comparison work is done while traversing the trees.

**Graph 2**: Print times using text files containing sorted words (no duplicates).

Text files containing n words:

* s-sorted.txt 25,033 words
* dict\_polish.txt 109,852 words
* dict\_danish.txt 519,986 words

Data Structures used to process these text files:

* Binary Search Tree O(log(n)) for insertion, access, and remove.
* AVL Tree O(log(n)) for insertion, access, and remove.
* RedBlack Tree (Java’s implementation) O(log(n)) for insertion, access, and remove.
* Splay Tree O(log(n)) for insertion and remove.

My implementation for printing a tree involves many approximately 4n string concatenations (where n is the quantity of nodes in the tree). Many such string concatenations is a costly process because in memory a new array of characters must be created, and then the old array of characters is garbage collected. A less costly technique is to use Java’s StringBuilder, however doing so caused stack overflows in my overridden toString( ) method. Ideally, there would not be so many recursion happening in an algorithm, but this assignments’ specifications required recursion. Clearly, an iterative approach would greatly reduce the size of the stack during run time.

**Conclusion**: Again, there is possibly some anomaly with the Polish dictionary that is causing the Splay tree an exceedingly high passage of time just to print the tree of 109,000 nodes as compared with printing the Danish dictionary having almost 5 times more words. At a quick glance, the average length of words are nearly the same. I would expect BST and/or AVL to have such slow times to print to the console. AVL and Splay perform nearly the same, but still exhibit an anomaly where they both are 5 times slower to print 109,000 words from the Polish dictionary as compare to printing 500,000 words from the Danish dictionary. Not having duplicates should not matter because the same comparison work is done while traversing the trees.

**Graph 3**: Books of English text (with duplicates).

Text files containing n words:

* huckleberry.txt 14,084 words in an e-book, book author Mark Twain.
* gulliver.txt 34,343 words in an e-book, book author JONATHAN SWIFT, D.D.
* warandpeace.txt 627,749 words in an e-book, book author Leo Tolstoy.

Data Structures used to process these text files:

* Binary Search Tree O(log(n)) for insertion, access, and remove.
* AVL Tree O(log(n)) for insertion, access, and remove.
* RedBlack Tree (Java’s implementation) O(log(n)) for insertion, access, and remove.
* Splay Tree O(log(n)) for insertion and remove.

**Conclusion**:

These text files are organized differently with a higher words-per-line ratio, but I would not suspect that to explain how differently these graphs look as compared to the sorted text with a words-per-line ratio of 1. Overall this graph has a more typical appearance depicting slower times to finish putting a larger quantity of nodes into the trees. Having duplicates should not matter because the same comparison work is done while traversing the trees.

**Graph 4**: Books of English text (with duplicates).

Text files containing n words:

* huckleberry.txt 14,084 words
* gulliver.txt 34,343 words
* warandpeace.txt 627,749 words

Data Structures used to process these text files:

* Binary Search Tree O(log(n)) for insertion, access, and remove.
* AVL Tree O(log(n)) for insertion, access, and remove.
* RedBlack Tree (Java’s implementation) O(log(n)) for insertion, access, and remove.
* Splay Tree O(log(n)) for insertion and remove.

My implementation for printing a tree involves many approximately 4n string concatenations (where n is the quantity of nodes in the tree). Many such string concatenations is a costly process because in memory a new array of characters must be created, and then the old array of characters is garbage collected. A less costly technique is to use Java’s StringBuilder, however doing so caused stack overflows in my overridden toString( ) method. Ideally, there would not be so many recursion happening in an algorithm, but this assignments’ specifications required recursion. Clearly, an iterative approach would greatly reduce the size of the stack during run time.

**Conclusion**:

Anomalous behavior is suspected with such quick times to print books (with duplicate words) as compared to dictionaries of sorted words, yet the data shows proportional growth of slower times to print with higher quantity of nodes. Having duplicates should not matter because the same comparison work is done while traversing the trees.