Cecelia Dahlinger

Programming Lab 5 (2-5)

Linear Data Structures – EECS 2500

Dr. Thomas

12/9/17

**Introduction:**

The objective of labs 2 through 5 was to implement and analyze different data structures by placing words from a text file in different types of lists. This includes comparing total words, unique words, number of comparisons, number of reference changes, runtime, and various other statistics that may help in determining the best list type.

**Procedure and Data:**

First a text file input was parsed, and individual words delimited by spaces were stripped of any beginning or ending punctuation, forced into all lowercase, and placed into a queue. This queue now contained all of the words from the text file as individual objects, which made it easier to insert them into in the lists explored in this set of labs and to get a more accurate runtime for each list.

The first 5 lists to be implemented were all fairly similar. They all were made up of linked lists with certain properties. This meant that at the very basic level these lists would all perform similarly. The first list was the unsorted list. This list was simple to implement as it did nothing special to order the words in any way as they were inserted. This list will be used as a baseline for comparisons, kind of like a control in an experiment. The next was a sorted list that was sorted alphabetically by comparing the word to be inserted with the words already in the list. At first thought, this may seem like it would help speed the process up, but since linked lists inherently don’t have any way to jump to a node in the middle of the list, the extra comparisons will only slow things down. Next a new sorted list was implemented which was an extension of the previous sorted list. This new sorted list retained the previous node that was inserted into the list, and, by comparing the current node to the previous, the list would start searching either at that node or at the beginning of the list. This could possibly eliminate half of the time it takes to find the sorted item, thus being faster than the original sorted list. The next two lists implemented were lists that were meant to improve their speeds by adjusting the positions of the nodes within a list based on how often it is being looked for. In theory the more you search for a word, the more likely it will be towards the beginning of the list, thus lessening your searching time. The first self-adjusting list that was implemented was the heavy-handed or front self-adjusting list. This list took any node that was searched for and placed it at the beginning of the list. At first it seemed that a move this drastic may be too heavy-handed causing it to not be as efficient as the next list. Finally we had the single self-adjusting list which moved a node up one space if it was searched for. This does not work as well as the previous heavy-handed self-adjusting list because as new nodes are inserted at the beginning, the node that had just moved forward one now moves back one.

The 6th list that was implemented was the Skip List. The skip list is a probabilistic alternative to a binary tree and it offers a way to “skip” to the wanted node faster. Skip lists utilize parallel linked lists with 4 links on each node to create a sort of highway. The top level or lane has the fewest number of nodes, and as you travel down the levels each level contains the nodes from the level above plus more until you get to the bottom which contains all possible nodes in the list. A new level is determined by coin flip as nodes are inserted which is what makes it probabilistic. This list can run incredibly fast compared to the previous 5 lists which on average ran at a time complexity of O(N), while skip lists on average run at a time complexity of O(logN) with the possibility of degenerating to a complexity of O(N).

The graphs below depict the data gathered from implementing these 6 lists on 18 text files of varying sizes and vocabularies. All the way from Green Eggs and Ham which contains 188 total words to the complete works of Shakespeare which contains 881072 total words.

From the 2 graphs above, it can be determined that unique words more strongly determine run time than total words. This is because the speed is dependent on number of nodes which equates to unique words. One interesting anomaly in both graphs is the King James Bible. While there are about 790000 total words in it, it only has about 12800 unique words. The total amount of words being so high causes the small spike seen in the graph of Time vs Unique Words around the 13000 unique words mark.

The graphs above depict the number of reference changes and the number of comparisons which occur in each list vs total and unique words. The comparisons and the reference changes are a large component of the total time it takes to run, so this is a good visual representation of why each list took the amount of time that it did. It seems that the unsorted list was consistently performing the most comparisons, while the single self-adjusting list made the most reference changes with the sorted list not far behind. The unsorted list makes so many comparisons because it has a higher chance of searching the whole list before it finds what its looking for. The high amount of reference changes is because sorted inserts into the middle, and the single self-adjusting list is moving a node in the middle.

Here the new sorted list can also be analyzed. It does consistently make less reference changes and comparisons than the regular sorted list and the unsorted list. It still manages to be slower because of the extra pointer to the previously inserted node.

From these graphs, out of the 5 O(N) lists, the best in terms of runtime, reference changes, and comparisons is the front self-adjusting list. Because of its self-optimizing heavy-handed add method, it runs the fastest because it travels through the list fewer times.

It might be interesting to look at a comparison of the ratio of unique words to total words vs runtime since the runtime is affected by both differently. Unfortunately, this graph isn’t very helpful at all. The text files that have ratios above .1 are all relatively short, so there is no clear trend. However, there is another way to look at the affect of the ratio on runtime depicted in the graph below.

Shown above is a comparison between The Three Musketeers and Moby Dick. They have a similar total word count, about 22000, but very different unique word counts, 11160 and 20041 respectively. Moby Dick, which has more unique words and therefore a higher ratio, performed worse with all list types including the skip list.

A simple unsorted remove function was also implemented and analyzed. This function was implemented on all 5 O(N) list types, and depicted in the graphs below.

The King James Bible remains an anomaly in the remove function as well, taking less time than those around its total word count and more time than those documents with similar unique word counts. This is again due to its limited vocabulary. The single self-adjusted list, sorted, and new sorted list all did consistently worse than the unsorted list, while the heavy-handed front self-adjusted list did much better. This is because it was already optimized from the add method.

The skip list remove function is also shown in these graphs. It is unfair to compare it to the O(N) lists’ remove function since it works the same way the skip list does, giving it the advantage an average of O(logN).

Since the heavy-handed front self-adjusting list did so well with the regular remove function, a second remove function was implemented. Remove2 mimicked the add function of the front self-adjusting list by moving a node to the front when it was removed if the number of times did not fall below 1.

The original remove function out-performed the new remove2 function in reference changes, comparisons, and runtime. This may be because the list was already optimized, so it didn’t need to be optimized again as it might have undone the previous optimization.

The next type of list is a hash table. An object being inserted or removed is converted to a hash value which corresponds to an entry on a hash table which is a specified sized array. Three different hashing methods are evaluated. At the hash value if there is already an object in the array, there are several ways to handle it. The way it is handled in this project is with what is called the chain and bucket method. If a collision occurs, the position in a hash table that corresponds to the current object already has an object, a linked list is implement at that position in the array and the object is added to the list. The heavy-handed front self-adjusting list is used since they performed the best in the earlier trials.

The 3rd method was the most efficient. Method 2 didn’t do as well as the others because it was limited by the hash values that it could produce, since it only looked at the first letter of the word. Method 3 is a variant on the java hashing function.

Here the hash tables are compared to a skip list. The skip list was the fastest list we had looked at previously, but it was still not as fast as method 1 and 3 of the hash tables.

One way to visualize the speed a hash table will operate at is by viewing the minimum and maximum number of nodes in each table position. This is seen in the average value charts below. The dot is the average number of nodes in each list and the line represents the distance between the minimum and the maximum.

Method 2 has clearly longer lines between the minimum and maximum, indicating that it will take a longer amount of time to run and be overall less efficient.

This graph is a visualization of the hash methods used on the works of Shakespeare. Methods 1 and 3 are fairly similar while method 2 is clearly concentrated in one area of the hash table. This causes the hash table to be inefficient and degenerate into a front self-adjusting list.

The remove method was also implemented on the hash tables. Method 3 remains the most efficient. Remove implements the same search function as the add function which is the primary time consumption in both remove and add. This similarity can be seen if both add and remove graphs are compared.

The hash tables’ remove function is compared to that of the skip lists in the following graph. The skip list is still between method 2 and method 1 and 3.

Finally the last list to be implemented for this project is the binary tree. This is unlike the previous lists because it is a nonlinear data structure. Each node has 2 child nodes which split off into a structure that looks like an upside-down tree, hence the name. The binary search theorem states that the left child node must point to something less than the parent and the right child node must point to something greater than the parent node. Every binary search tree must follow the binary search theorem. This means that as the tree is traverse, each level eliminates up to half of the remaining data.

To show how the binary tree works first, a different set of text files is used. The text files each have a set number of random unique “words” with no duplicates.

Runtime seems about linear, except for the file containing 1 million “words.” This seems to be an outlier at 1.75 seconds.

Height seems to be a logarithmic function according to the trendline. The R^2 is a measurement of how accurately the data follows the trendline, with it being more consistent as the R^2 value approaches 1.

Now the previous text files were analyzed using the binary tree and added in with the data gathered from the random text files. This causes data to skew away from linear and logarithmic trendlines as seen in the height graph below’s R^2 value.

Next the binary tree was put up against the hash table methods 1 and 3 in the following graph. The binary tree seems to run slower than method 1 and method 3. This may be because the binary tree is possibly unbalanced. Binary trees are affected by the sortedness of the initial list of words, which might make it a less effective approach.

**Conclusion**

After implementing these different types of lists, they have their pros and cons. The easiest to implement is not always the least efficient, such as the unsorted list compared to the sorted list. The unsorted outperformed the sorted list in almost every case. Though the list that was consistently the best was a version of the hash table. The hash method that was the most efficient was the variant of java’s hashing function. It makes sense that java’s creators chose that method since it returns more evenly distributed hash values. I was neat to see and understand how these lists work in order to gain a better understanding of which list to use when.