Rio Bacon
CSE 332
May 8th, 2012
Project 2 write-up

1. Worked by self.

2. Assistance included textbook and lecture slides, for implementing several of the data structures and sorting algorithms.

3. I worked in many different smaller sessions, so it is hard to tell, but as a very rough estimate, probably about 25 hours total. The most difficult was implementing then debugging the AVL tree. This project could be better, by eliminating simple programming, such as identifying parameters for the program, since it was not anything new, and only took up time.

4. I did not do any.

5. I tested the data structures, but not sorting algorithms because being able to sort the word count of the two text files correctly should cover all other possible sorting cases, including sorting only 1 element, since the algorithms inherently sort 1 element at some point. For the data structures, I considered the case where the data structure is constructed, but no values entered, the case after adding only one, the case where adding many, and the case where adding duplicates. I also tested the iterator to make sure the proper values get returned.

6. Because every node needs to return its children unless it is a leaf, all nodes must be kep track of as the tree is traveresed. One way to do this is recursion, but this would not work with an iterator, so to preserve all nodes until all its children are returned, they are kept in a stack. To avoid resizing the stack, you can change the stack implementation so it can take a size parameter for its contructor, and pass in the size of the tree when creating a stack for iteration.

7. For the two trees, this could be implented fairly efficiently, because nodes could all be pushed into the stack in the order of 'in order' when the iterator is created, then just pop as next() is called. For MoveToFrontList, there is no efficient way of doing this, because the list does not store data in any particular order (although frequent counts will tend to be towards the front). Same with Hashtable, because the elements are placed according to the hash function, which results in seemingly random placements of the elements. For the latter two, the entire set of words would need to be sorted as the iterator is created.

8. For the hashtable to be correct, if the comparator returns a 0 for its compare function of two elements, then the hash function of the hasher must return two identical ints for the two elements.

9.

Testing Types of data structure and sorting using hamlet.txt:

| | Binary tree | AVL tree | MoveToFrontList | Hashtable |
|---|---|---|---|---|
| Insertion Sort | 701 ms | 549 ms | 2161 ms | 334 ms |
| Heap Sort | 670 ms | 872 ms | 1882 ms | 575 ms |
| Merge Sort | 123 ms | 99 ms | 1849 ms | 112 ms |

Testing Types of data structure and sorting using the-new-atlantis.txt

| | Binary tree | AVL tree | MoveToFrontList | Hashtable |
|---|---|---|---|---|
| Insertion Sort | 263 ms | 117 ms | 333 ms | 90 ms |
| Heap Sort | 219 ms | 223 ms | 683 ms | 258 ms |
| Merge Sort | 66 ms | 45 ms | 502 ms | 73 ms |

For the overall fastest result, looking at both the results of hamlet.txt and the-new-atlantis.txt, it seems that using an AVL tree with Merge Sort is the fastest. Interestingly, this does not mean confirm that AVL tree is the quickest data structure or that merge sort is the quickest sorting method. For example, when using insertion sort, it appears that using a hashtable is quicker than an AVL tree. There is also the case in the-new-atlantis.txt where quicksort performed quicker than merge sort when using a MoveToFrontList.

For change in performance, MoveToFrontList seemed to have the biggest increase in time with hamlet.txt from the-new-atlantis.txt. In the-new-atlantis.txt, it only performed roughly 2-5 times worse than the other data structures, where as in hamlet.txt it preformed roughly 3-18 times worse. Because of the way MoveToFrontList works, it may due to hamlet.txt having a wider variety of words. But this may simply be caused by having a larger amount of text, resulting in a larger variety. Similarly, merge sort seems to hold on to its quick sorting time with increase of text better than heap sort and insertion sort. There is a wider much wider gap between the times for merge sort in hamlet.txt than the-new-atlantis.txt, which suggests merge sort is able to sort large quantities better than the other two.

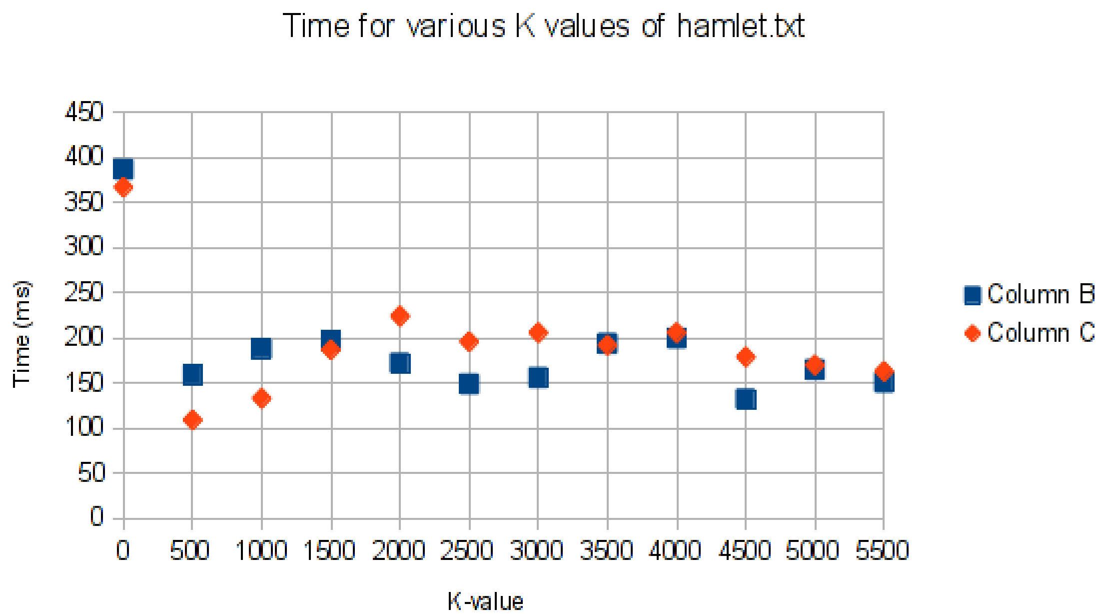
Changing the hash function for hamlet.txt

| | Normal hash function | Alternate hash function |
|---|---|---|
| Insertion Sort | 334 ms | 6797 ms |
| Heap Sort | 575 ms | 4500 ms |
| Merge Sort | 112 ms | 6580 ms |

The change in the hash function I made was that the hash always returns 0. This clearly caused a significant increase in time, especially since I used quadratic probing.

These experiments were run by creating a timer program that would check the time before executing WordCount, then checking the time upon finish, and printing the difference. The program ran this for multiple input parameters of WordCount.

10. Column B represents nlog(n) approach using merge sort, while column C represents the nlog(k) approach.




As seen in the graph, for value smaller than 1000, the nlog(k) approach seems to be fastre, but beyond that, just doing mergesort seems to be the quicker option. If this experiment were to be trusted, then the implementation should be changed so that for values of k above 1000, merge sort would be used on the entire set of words, where as the nlog(k) method would be used for smaller values of k.

11. With a correlation of roughly 0.5E-4, this is very close to 0. Because a correlation coefficient of close to 0 means that there is a very weak correlation between the two sets of data, we can say that it is likely that Bacon did not write Shakespeare's plays, or at least he did not write Hamlet.