Trevor Hickey
Data Structures and Algorithms
Dr. Bennett
December 2, 2013

Program 8: Analyze a Text Document

The Problem:
The problem states to read in words from a file, and store them in memory so that they can later be analyzed. The operations that will be performed on the words are as followed: "print the total number of words, and total number of unique words", "print the top 10 most frequently occurring words", "search for a word by its value or search for a list of words by its frequency". The goal is to make all of these operations perform efficiently. When choosing the appropriate data structures and algorithms, we must consider our initial start-up time, the time complexity of each operation stated above, and our overall space complexity.

My First Approach:
We know that the words read in will be unalphabetized, and that re-occurring words do not have to appear one after another. My first approach would be to use a hash table, or more specifically, an unordered map. The specifications never state that we will need to output the data in order. When it comes to searching, it is often advantageous to have the data in order, but if we are hashing to find if a word exists, unsorted data is not a problem. Taking this approach, we would still need to do a full iteration over the unsorted data and build a list of the top 10 most frequent words. Assuming we have a good hashing function, and an appropriately sized table, This method will provide constant time complexity for both insertion and searching of a word based on its value. The size complexity would hopefully be close to the number of unique words found. Again, this would be determined by the hashing function, table properties, and given data set. One thing is certain though, we would not waste space holding duplicate words. I imagine words such as "the" appear frequently, and it would be a waste of space, and possibly computation time, to store all of these duplicates before later reducing that data down into paired values containing words and frequencies. Searching by frequency would still take O(n) time complexity unless we decided to use a second data structure.

My Second Approach:
Hash tables(or maps that use hash tables) are not allowed. Since we can no longer hash words into locations, I've made the decision to sort the data. Sorted data, regardless of what data structure it's in, will allow a binary search approach to finding a given word. With unsorted data, it would take a time complexity of O(n) for

each look-up. However, after the added cost of sorting my data first, searching will now have a time complexity of O(log n). This time complexity occurs for both the average and worst case performance. Best case performance would be O(1).

Choosing A Data Structure:
Binary search does not necessarily mean constant time indexing. My choice to sort the data and use a binary search algorithm has not limited me to using an array or a vector. A binary tree, a Red-Black tree, and an AVL tree, all provide searching at a time complexity of O(log n). They also have an insertion cost of O(log n), but this is to be expected, as these trees are self-sorting. In fact, the advantage to using a tree over a vector in this instance, is that we can avoid storing duplicate entries and increment the frequency of a word immediately. Consider what would happen if we used a vector instead of a tree. The data would be out of order and for each new insertion, we would have to either do a linear search to discover if it was a duplicate word, or we would need to tack it on to the end of the vector; leaving us to sort a larger set of data before reducing the vector down and removing duplicates. Even sorting and reducing for each new insertion sounds more costly on a continuous set of data than it does re-arranging pointers on a tree.

Choosing A Tree:
A tree will maintain sorted data during each insertion, and ensure that we are never holding duplicated data to deal with later. A binary tree, might provide the fastest insertion, but it has the potential to degenerate into a linked list. Given our data, and the fact that we will be reading once, and searching many times, I think it would be advantageous to use a more balanced tree.

AVL trees maintain a more rigid balance than Red-Black trees. We can say that the the path from the root to the deepest leaf in an AVL tree is at most lg(n+2), while in red black trees it's at most lg (n+1). This means AVL trees are typically faster to search, but slower to insert and delete from. I would argue that although we are concerned with searching quickly, we are also concerned with our initial start-up time. Using an AVL tree means that every insertion, on average, will be slower than every insertion into a Red-Black tree. I recommend we use a Red-Black tree in order to keep the initial start-up cost low. This will only make searching a little bit longer on average(~1.44 compared to ~2). The decision here to choose a Red-Black tree over an AVL tree, really depends on how much data is expected to be read in, and how important searching time is compared to stat-up time. Since we are reading in big files, and not necessarily doing a large amount of continuous searches, I think a Red-Black tree fits best.

A Container For Associativity and Frequency Counting:
There is a higher level data structure that is usually implemented with a Red-Black

tree. That data structure is a set. Sets are containers that store unique elements following a specific order. If you try to insert an element that already exists in the set, it will simply not add it. Therefore, I will build a wrapper around the set class to increment a count on the words that already exist in the class; call it a frequency set. Once the frequency set is built, I will iterate over all the elements and build a vector whose comparison value is not the word but the frequency. I will quick sort that new list of data. The frequency set will be used to look up words by their values, and the newly created vector will be used to look up values by their frequencies. The newly created vector, could just be a list of iterators, and that would save space, but searching may be slower due to dereferencing. I don't consider a size complexity of 2n being a concern, where n is the number of unique words. Unless we are dealing with extremely large files, or a small amount of memory(or both), this size complexity is a cost we should be willing to take in order to get O(log n) time when searching with either a word or a frequency.

Implementation:

The important implementation detail is the frequency_set. The frequency_set wraps a set of words, and a list of words sorted by frequency. They each used two different Comparator types. Each Comparator type stored a std::string for the word and an unsigned int for the frequency. The Word_Comparator used only the word to make its relational comparisons, and the Frequency_Comparator used only the frequency to make its relational comparisons. I did this so that the set was designed to automatically sort by the word value, and the frequency list would be populated with set's data, only operating with a different comparison behavior(on the frequencies). The reason I didn't jump to functors or lambda functions was because the insert method of a set didn't allow for either.

Total Average Costs

Extracting Words into Memory (Unavoidable Cost)
O(*n*) extracting all the entries from a file, where n is size of the file.
O(*n*) Identifying whether the extracted datum meets our criteria of a word, where n is the length of the word.

Creating the Frequency_Set (The Costs I chose)
O(log *n*) insertion for each valid word into the set
O(n) to put all of the sorted word/frequency pairs into a vector
O(*n* log *n*) to sort the frequency list (quick sort)
O(2*n*) max size complexity at any given time, where n is the number of unique words

Operation Performance of the Frequency_Set
O(1) Printing Total Words
O(1) Printing Total Unique Words
O(log *n*) Obtaining a frequency, searching by word; n is the number of unique words
O(log *n*)+*k* Obtaining words, searching by frequency; n is the number of unique words, and k is the number of additional matches found
O(*k*) Getting Top k most frequent words

Difficulties/Problems:
There was a rigid detail I overlooked in the expected interface of a set. When you try to insert and element into a set, it will give you back an iterator to a constant object. This means, I would be unable to increment the frequency value inside the word/frequency pair because that would be considered transforming the object.
I don't like using the keyword mutable, because it always feels like cheating, however it was the easiest and most obvious solution.

Discoveries:
I never used std::equal_range before on an iterative container. I thought once I found a frequency in the sorted frequency list, I would continue looking left and right to grab the rest of the words with that matching frequency. Turns out, this function does that for you. I discovered this when I looked up the documentation for std::binary_search and found that it only returned a boolean and not an iterator.

I knew about C++11 delegating constructors, but I discovered there is a second way to write them which is easier to read.

A templated class **A** which derives from a templated base class **B** can not simply

access the protected members of class **B** in the same way had they not been templated. The trick is to add *"this->"* to the front of **B**'s member data when inside class **A** to make the member data a dependent name.