Zachariah Yoshikawa - A01417300

Michael Ward - A01755332

**Mini Search Engine Project**

    Using data structures to design a mini search engine was an enjoyable experience for us. The extra credit that we implemented was to eliminate stop words in the search. We learned a lot about hash tables and how to implement them. We had some issues along the way and learned how to conquer them through debugging, trial and error, and with a little bit of luck. To explain the content and experience of our project, we will divide this paper into four parts: Our data structures and the reasoning for using them, our algorithms and their time complexities, and our design issues and how we solved them.

1. **data structures you use, the decision behind selecting them**

    We decided to use a hash table to store the words from the documents that need to be searched. We first started with a Node class. This class has several variables including a string named data. This string stores the word from our readIn() function that will be used to match the Node for the search. The docNo[ ] is an array the size of the total search documents. It stores the number of the document the search word is found. And there is a next pointer to be used to implement a linked list.

    Our linked list class uses the Node class to make a list of the search word elements. This class enables us to use separate chaining to handle collisions for our hash table.

The hash table class has 521 buckets. It is a prime number. We use a hash function based on the value of the ascii characters of the search word to evenly disperse the words across the table.. This enables the table to avoid generally collisions for most common words in the English language. If there are collisions, it is easily handled as mentioned above. But we chose to design it this way to be able to retrieve and append the search words with O(1), or constant time for best case. The worst case will not usually be reached, but it is O(n). The main advantage of using a hash table over other table data structures is speed. This will become more apparent when the number of entries is large. This is why we made our table size 521 in hopes to optimize the size and speed.

**2) algorithms you employ, again with a justification of your decision particular emphasis should be placed on the running time of your algorithm**

The most important algorithms to finding the documents with the search words are within the following functions: bool checkWord(), void readIn(), int \*searchQuery(), and int \*userQuery().

In bool checkWord(), we utilize an array of the words that we read in from “stopwords.txt” (done in readInStop()), We check the words as they are read in. If the word has a ‘<’ character (XML tag) or is a stop word. It returns true only if it does not have these attributes.. This eliminates unnecessary words and characters within the hash table. readIn() uses this function and reads in the words from the documents and appends them to the hash table.

The function userQuery() enables the search of up to two words at a time using the boolean operations of “AND” and “OR”. The first word is read in and the document numbers are returned to an array. Then if the second word is an “and” or “or” (case insensitive), then it will take the third word and compare accordingly. If only one word exists, we assume that the user is only searching for one word, and perform a single search. We accomplish this by setting an int to 0, 1, or 2. We use 0 to search for one word, 1 to search for word1 AND word2, 2 to search for word1 OR word2.

In order to perform a search including AND or OR, both words are first searched for separately and both words return an array containing the document numbers that they are found in . Then, in the next part of the userQuery() function, the resulting lists are combined according to the user’s input. If the user input is AND, the execution is relatively simple. The array result from the first word will be stepped through, and the second array will be searched to see if it contains the same number from the first array. If the first number is found in the second array, that means both of the user’s words appear in that document and the number is then stored in a third and final “result” array. After every element in the first array has been stepped through, the third result array will contain all the documents that both words appear in and is then displayed to the console.

The OR logic is more complex, and therefore the implementation is more complex. The logic is as follows: step through the first array and add all elements to the result array. Then step through the second array and check if the number from the second array is already in the result array. If it is not in the result array, add it to the array. If it is already in the array, ignore and move on. In order to keep the result array sorted, if it needs to add a new number, it will add it before the first number larger than itself and push back every item in the array after that.

At first, we had used a very inefficient algorithm to perform this task. It had a terrible runtime efficiency of O(n^3). When we wrote the code, it made sense, but when looking back on it later, we could not figure out the logic behind this implementation, and therefore can not state it here. We rewrote this into a slightly more efficient algorithm with a runtime efficiency of O(n^2).

Although our hash table is very fast and efficient, our implementation of comparing two words together uses nested for loops to compare each element of word1 array to each element of word2 array. This creates a running time complexity of O(n^2). This could be optimized along with several other things, which will be discussed later.

Finally, the searchQuery() function  is used by userQuery() and searches the documents for the search word and returns the document number. It is used within the algorithm described above.

**3) design issues, what are the problems and how you solve them**

    One design issue we had was the implementation of the OR logic. Our first try used an implementation similar to the AND logic. It would loop through the result array for the first loop and then loop through the result array for the second word. In both loops,  every word would be added to the result array. While this implementation, in theory, would return all the words found in either array, it produced many undesirable results. First off, if a word was found in both arrays, the word would be returned twice. Also, since we capped the result array at 50 (the total number of documents, and therefore, in theory, the total number of results possible), sometimes a correct result would not appear in the array. This happened whenever both words were found in the same document and the total number of occurrences exceeded 50. Duplicate and unordered results was very bad, so we had to fix this.

    The second implementation we used achieved the desired result, but was very inefficient. It would loop through the first array, within that loop traverse the second array, and then call an add() function. This function takes a number, array, and size of array as arguments and then adds the number to the array. However, before adding it, it checks to see if the word is already contained in the array. This prevents duplication, but three nested for loops were needed in order to produce a sorted result, leading to a runtime of O(n^3). Although this implementation did work, it had a terrible runtime, and therefore we decide to improve it.

    Our third and final implementation was a combination of both of the first two. It included the two seperate for loops from the first implementation and the add() function from the second one. However, to keep the result from being unordered, we needed to change the add() function slightly. Originally, the function would add the new number to the end of the stored numbers in the array (at the index of SIZE in the array), but we changed this to add find the index (stored in i) in the array of the first number that is greater than the number that we are adding. All subsequent entries in the array are then pushed back one place and the size variable for the array is incremented by one. This created a result that contains all the correct items and is sorted as well as having a slightly better runtime efficiency of O(n^2). While this is still a very bad efficiency and ideally would be improved, time constraints prevented us from creating a fourth implementation that worked, so we decided to use the most efficient working implementation we had made.

    Another challenge we had was reading the input files. At first, we wrote something along the order of:  
1. loop through integers starting at 1 and ending at 50 and set to i

2. call readIn function with “DOCUMENT INDEX\\cranfield000.txt" + i

However, this did not work at all and we were perplexed as to why. After spending about an hour trying to change different things, we realized that the input files did not have a .txt extension! After removing the .txt from the files, the queries worked, but no query returned a document with a number greater than 9. We decided it was impossible that documents 10 - 50 might not contain any words we searched for and verified this by taking a look at the contents of these files. After looking into this for far longer than necessary, we found that we had one extra 0 in the filename for documents 10 - 50. For example, we were trying to open “cranfield00010” instead of “cranfield 0010”. We fixed this by creating two separate for loops, one for 1 - 9 and one for 10 - 50. This does not affect efficiently at all. While the readIn challenges may seem very insignificant, it took us a long time to figure out, and therefore is worth noting.

**4) optimization issues: what could you do to further optimize your algorithm**

**you need to specifically address the problem of scalability: would your implementation be efficient in the case of very large text collections?**

    First of all, our hash table could be improved for handling very large text collections. We used a very simple hash function where the ascii values of each letter in the word were added up. The result was then divided by the size of the hash table using C++’s modulus function and the remainder returned was the result of the hash function, and therefore the bucket that the word would be place into. The words seemed fairly evenly distributed throughout the hash table, so we decided that for this size of documents, this hash function worked well. However, we did increase our size table to 521 from 128 to improve efficiency.

    For very large amounts of data, a quadratic hash function would be more efficient. Additionally, the size of the hash function would be increased to accommodate for more words. Another change that could help in this scenario would be to replace the linked list for hash bucket collisions with a more efficient data structure such as a binary tree. Additionally, the data structure we used to hold the results of queries (arrays) could definitely be improved for more efficient runtimes with large text collections.

    Another change that would be equally beneficial for large or small text collections would be to improve the AND and OR logic function. Our current runtime efficiency is O(n^2), which is quite bad. However, this is the best efficiency we could achieve for a working implementation. Overall, our efficiency is satisfactory for this data size, since all runtimes are almost instantaneous. However, the above stated improvements would be very good for larger data sizes.

**6) Query results:**

flow

Doc1, Doc2, Doc3, Doc4, Doc6, Doc7, Doc9, Doc16, Doc17, Doc18, Doc19, Doc21, Doc22, Doc23, Doc24, Doc25, Doc26, Doc27, Doc28, Doc33, Doc34, Doc35, Doc36, Doc37, Doc38, Doc39, Doc44, Doc45, Doc48, Doc50

flow AND stream

Doc1, Doc2, Doc16, Doc39

flow OR stream

Doc1, Doc2, Doc3, Doc4, Doc6, Doc7, Doc9, Doc16, Doc17, Doc18, Doc19, Doc21, Doc22, Doc23, Doc24, Doc25, Doc26, Doc27, Doc28, Doc33, Doc34, Doc35, Doc36, Doc37, Doc38, Doc39, Doc44, Doc45, Doc48, Doc50

supersonic AND speeds

Doc7, Doc40, Doc41

the AND boundary AND layer

NOT FOUND

(note: since ‘the’ is a stop word, it is eliminated from all documents, so any AND query including ‘the’ will return nothing)

boundary-layer

Doc2, Doc3, Doc4, Doc7, Doc8, Doc16, Doc17, Doc21, Doc24, Doc25, Doc34, Doc36, Doc37, Doc40

velocity OR speed

Doc1, Doc4, Doc7, Doc12, Doc14, Doc15, Doc18, Doc19, Doc24, Doc32, Doc33, Doc36, Doc43, Doc44, Doc49

stream AND boundary-layer

Doc2, Doc16

speed AND number

Doc24, Doc33

number OR boundary

Doc2, Doc3, Doc4, Doc7, Doc8, Doc9, Doc10, Doc12, Doc14, Doc16, Doc17, Doc18, Doc21, Doc22, Doc24, Doc25, Doc30, Doc33, Doc34, Doc37, Doc40, Doc41, Doc43, Doc45, Doc46, Doc47, Doc49, Doc50

science OR number

Doc7, Doc8, Doc9, Doc10, Doc14, Doc21, Doc22, Doc23, Doc24, Doc25, Doc30, Doc33, Doc40, Doc41, Doc43, Doc45, Doc46, Doc50

velocity AND science

NOT FOUND

velocity OR science

Doc1, Doc4, Doc7, Doc14, Doc15, Doc17, Doc18, Doc36, Doc43, Doc44, Doc49

boundary

Doc2, Doc3, Doc4, Doc7, Doc8, Doc9, Doc12, Doc16, Doc17, Doc18, Doc21, Doc22, Doc23, Doc24, Doc34, Doc37, Doc40, Doc43, Doc45, Doc47, Doc49, Doc50

the OR boundary

Doc2, Doc3, Doc4, Doc7, Doc8, Doc9, Doc12, Doc16, Doc17, Doc18, Doc21, Doc22, Doc23, Doc24, Doc34, Doc37, Doc40, Doc43, Doc45, Doc47, Doc49, Doc50

(note: the above two are exactly the same since ‘the’ is a stop word and is therefore eliminated)