Assignment 3: Report

This program is designed to take in a stream of words from a text file and process this for storage in to a custom hashtable. The hash table uses the ascii codes for each letter within each word, and this then determines the hash code for storage and retrieval. In cases where a collision has occurred, linear probing is used to find an empty position for storage. Great care has been taken to try and reduce the size of arrays so that only necessary space is used. In reference to the second part of this assignment - quicksort is used to order the data, before being output to screen.

Design

The program begins by asking the user to input a filename, which is then used to open a text file (assuming that the file is in the local project file). A string vector is used to store the output of this. Once this has been created, the 'removeDuplicate()' function, removes all of the duplicate words, and the size of the vector after this tells us how many array elements are needed for the hashtable. This number is used in comparisons later to choose which variant of the hashtable to use. In choosing the size of the hashtable, I have decided to try and make the minimum viable size, rather than prioritize rehash speeds. This means rehashing may be comparatively inefficient because values are bunched up in the hashtable to save space.

The design for the hashtable itself was relatively straight forward, and I did not need to look at example hashtables to implement it. In early versions of this program, a paired vector was used instead of the hashNode structure seen in the final version. Swapping to this helped with containment and modularity. Initially, there was only one hashtable class used, which was very large to account for large file inputs. After many revisions, this was then changed so that there were two child classes (hashtableLarge & hashtableHuge) which could be used if the number of unique words in the file exceeded the space available in the original base class. To implement different objects based on comparisons, I found that simple initialisation within an if/else statement was not possible due to some scope issues. After many different attempts, advice from a forum suggested to create a pointer for the base class, which could then be used to create a new instance of a child object (using the 'new' keyword) within the comparative statements (see line 45). This had been a significant challenge to debug and assert that it was working. The hash function which parses words is implemented as a method of the hashtable class.

The hash function used to create keys works like so: the sum of all ascii values from each letter in a word (minus 96 to each ascii value in order to simplify letter position) is then divided (with the modulo operator) by the number of unique words found at the start of the program. This then produces a key that can be checked in the table. This hash function seemed to be an efficient way to work with table sizes without having large gaps of empty data. In cases where a collision occurs the rehash function uses linear probing. The reason for this is that it should be more efficient in terms of the viable table size. If dynamic sizing were possible, I may have investigated other forms of rehash algorithm. The table sizes in each class variant are given some overhead free space, so that collisions can be handled without pushing words out of the available range. Ensuring an efficient table size has taken priority, at the cost of rehash speeds. Collisions may be lengthy depending on the input text.

In terms of handling text files of unknown lengths – I have decided to err on the side of larger is better. The largest text file successfully tested with this was 600 words, and this had 250 unique words. This was handled by the hashtableHuge derived class. After the hashtable has been created and counts made of the original text file, the data from this table is then moved to a paired string vector using the copyArray() function. The vector copy is then fed into the sorting algorithm. This program uses quicksort, because it is reportedly more common, and I wanted some practice with it. Quicksort is said to be faster than mergesort in cases where there are smaller data sets and works well with larger data sets also.

Testing

The testing strategy for this has been as varied as I could think of. Primarily this involved testing different characters at the input stages. Some special characters still produce some unexpected behaviours, though I have dedicated my time to refining the core hashtable/sorting related elements of this program.

Input	**Output**
Test file of 100 words – 57 unique words	Fine
Test file of 250 words – 100 unique words	Fine
Test file of 600 words – 250 unique words	Fine
Test file with special characters	Mostly handled, though some are still unpredictable. For instance the ‘ symbol is being output as Æ . Despite lots of time spent trying to rectify this, only some of the issues were resolved.
Test file with over 500 unique words	Message displayed to user – “Please re-size the hashtable class”
Enter file name that is not present	Message displayed “Error, file not found”