**Name:** Erika Valle-Baird

**Description of Program:** This program takes two input files and a word count for the first file via command line. The first file, a dictionary, is read into the program and creates a hash table, of which the second file can be quickly compared against to check for spelling. During creation of the hash table, different statistics will be tallied up to display in individual text file. After this, the text file, third of the command line arguments, will be read into file. Initially all special characters will be removed, and lines will be split. After this, each line will be read word by word, checked and changed to lowercase if uppercase, then checked against the dictionary file to determine if word is in the dictionary. Should the word not be in the dictionary, different spelling suggestions will be displayed, along with the words and the lines they fall within, in different text file.

**Time Analysis:**

Let’s say:

**n**=Number of words in the dictionary. **m**=Number of words in the input file for spell checking. **k**=Length-Number of characters in current word. **v**=Number of elements in a vector  **i=**Integer value

**explicit HashTable(int size = 101) : array(nextPrime(size)) -**

O(1):

2 Data Member initializations in constant time.

**bool contains(const HashedObj& x) const-**

O(1):

Constant time lookup to return True or False Value.

**void makeEmpty()-**

O(N):

The for loop executes at most v times.

**bool insert(const HashedObj& x) -**

O(1):

if statement executes in O(1) time.

Check v position for Active call executes in O(1) time.

if statement executes in O(1) time.

**bool remove(const HashedObj& x) -**

O(1):

if statement executes in O(1) time.

Check v position for Active call executes in O(1) time.

**int getCurrentSize()-**

O(1):

Constant time lookup to return value current size.

**int getTotalSize()-**

O(1):

Constant time lookup to return value of total size.

**double getLoadFactor()-**

O(1):

Constant time lookup to return value of load factor.

**int getNumberCollisions()-**

O(1):

Constant time lookup to return value of the number of collisions.

**double getAverageLength()-**

O(1):

Constant time lookup to return value of the average length.

**bool isActive(int currentPos) const-**

O(1):

Constant time lookup to return value type in v position.

**int findPos(const HashedObj& x) const-**

O(N):

The while loop executes at most v times.

The if statement simply wraps around the vector, meaning it can only execute as many times as the while loop is true.

**int findInsertPos(const HashedObj& x) const-**

O(N):

The while loop executes at most v times.

The first if statement simply wraps around vector, meaning it can only execute as many times as the while loop is true.

The second if statement is a count for longest chain and only executes as many times as the while loop is true.

**void rehash()-**

O(N):

First for loop creates larger hash table and can only be as big as 2(n)

Second for loop executes at most n times for information copy into newly created hash table.

if statement executes only if for loop is true.

**int myhash(const HashedObj& x) const-**

O(1):

Simple integer value check against v and integer return.

**bool isPrime(int n) -**

O(N):

The for loop executes by making calls and incrementing until i is no longer prime.

**int nextPrime(int n) -**

O(N):

The if statement integer check O(1).

The next if statement integer check O(1).

The for loop may have to make multiple calls to isPrime along with incrementation resulting in large number possibilities O(N).

**int hash1(const string& key) -**

O(N):

The for loop executes k times, converting string to integer value.

**int hash1(int key) -**

O(1):

Constant time lookup to return value.

**int main(int argc, char\* argv[])-**

O(N^2):

The if statement-executes n times.

Nested while loop depends on if statement being true.

The while loop executes at most m times.

The for loop depends on while loop being true-checks each line deleting special characters and executes m times.

The if statement executes m times, as long as for loop is true.

The nested while loop-executes at most m times.

The if statement executes at most m times.

The nested if statement executes at most n times.

**string toLowerCase(string) –**

O(N):

The while loop executes at most k times. This traverses the word, checking one character at a time against ASCII table for possible addition change. This takes O(k)=O(N) time.

**void letterDelete(HashTable<string>& dictionary, string word) -**

O(N):

The for loop executes at most k times. This traverses the word, deleting one letter in word at time.

if statement checks against dictionary and is executed at most n times.

**void neighborSwap(HashTable<string>& dictionary, string word) -**

O(N):

The for loop executes at most k times. This traverses the word, swapping neighbors.

The swap function executes in O(1) time.

The if statement call to contains method.

**void letSwap(HashTable<string>& dictionary, string word) -**

O(N^2):

Outermost for loop executes at most k times.

Second for loop executes (26\*(k+1)) times, due to alphabetic possibilities.

The if statement call to contains method.

**void replaceLetter(HashTable<string>& dictionary, string word) -**

O(N^2):

The for loop executes at most k times.

Second for loop executes (26\*(k+1)) times, due to alphabetic possibilities.

The if statement call to contains method.