Name: Paul Murariu

Teacher: Professor Jinhua Guo

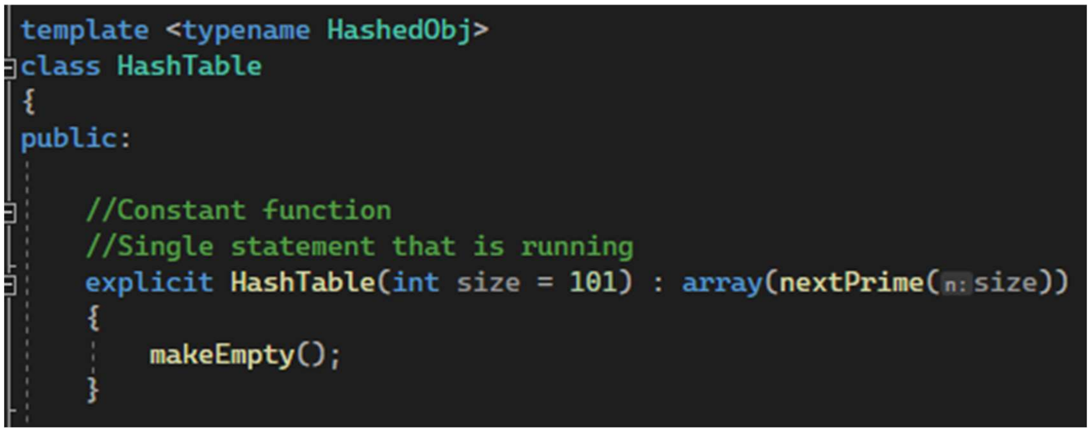
Course: CIS 350 – Section 102

Date: 5/29/2022

Description of the problem in own words: This problem tasks us with creating a spell checker that consists of three command line arguments which are specifically the number of words in the dictionary, a dictionary file name, and a text file name. The program that is created will produce a hash table, then read off the dictionary from the specific file, then insert the words into the created has table, and finally report the statistics of the collisions. Once the program has read the dictionary, the created spell checker will read a list of words from a created text file. Each specific word will be looked up in the dictionary and if the results end up being incorrect, the output will end up reverting to a standard output combined with a list of suggested corrections. Finally, you also need to find the time analysis worst case O() of each function with explanation, including the main function, and the average case O() of the contains, insert, remove, and findPos functions as well.

<u>Time Analysis Cases For Member Functions With Explaation</u>

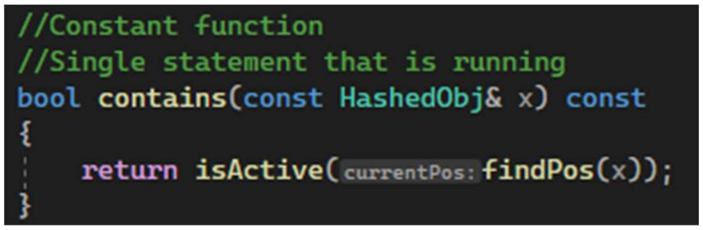
<u>QuadraticProbing.h Functions</u>



```
template <typename HashedObj>
class HashTable
{
public:

    //Constant function
    //Single statement that is running
    explicit HashTable(int size = 101) : array(nextPrime(n: size))
    {
        makeEmpty();
    }
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only one statement that is running.



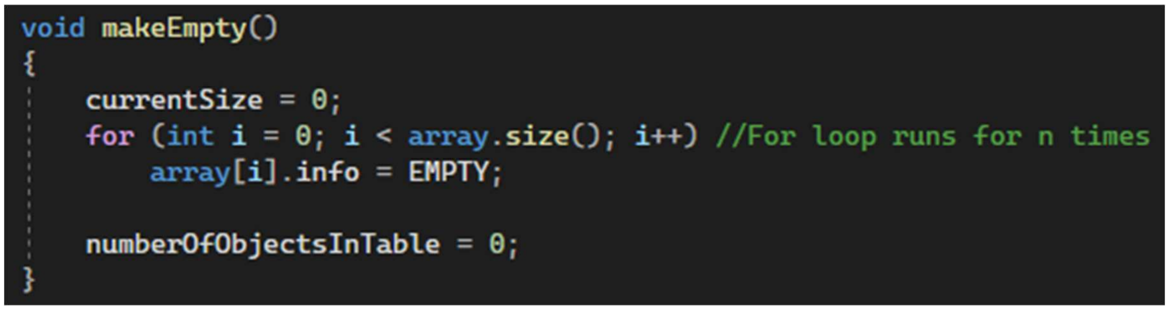
```
//Constant function
//Single statement that is running
bool contains(const HashedObj& x) const
{
    return isActive(currentPos: findPos(x));
}
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only one statement that is running.

Average case complexity is: O(1)

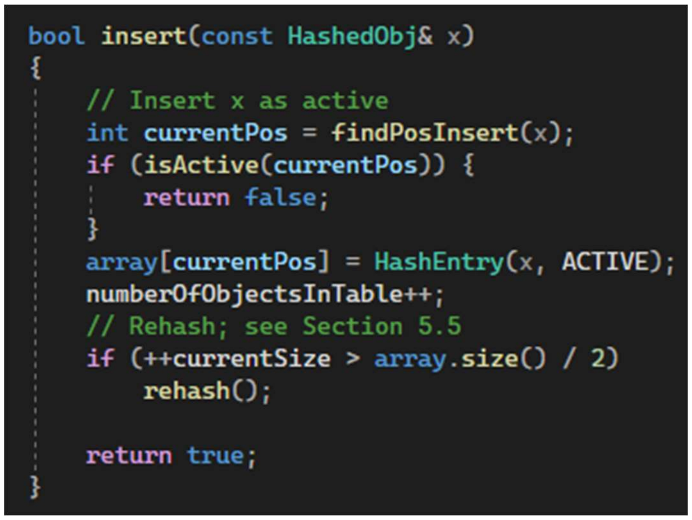
The average case complexity is O(1) because the function has only one statement that is running.



```
void makeEmpty()
{
    currentSize = 0;
    for (int i = 0; i < array.size(); i++) //For loop runs for n times
        array[i].info = EMPTY;

    numberOfObjectsInTable = 0;
}
```

Worst case complexity is: O(n)

The worst case time complexity is O(n) because the function has a for loop, which runs for n times.



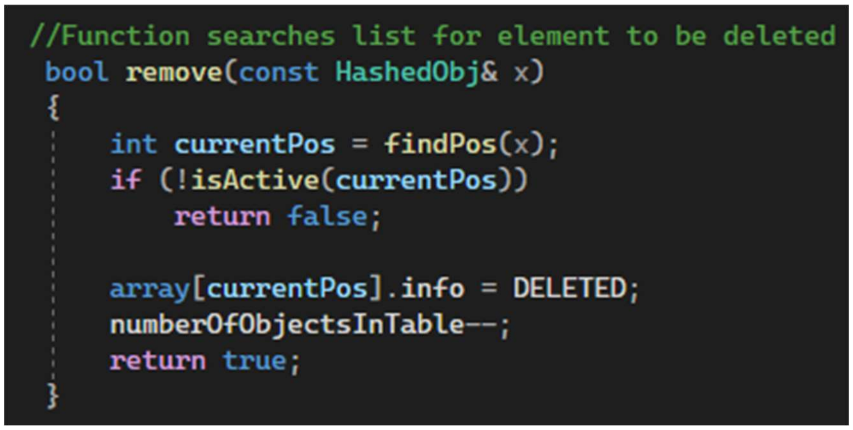
```
bool insert(const HashedObj& x)
{
    // Insert x as active
    int currentPos = findPosInsert(x);
    if (isActive(currentPos)) {
        return false;
    }
    array[currentPos] = HashEntry(x, ACTIVE);
    numberOfObjectsInTable++;
    // Rehash; see Section 5.5
    if (++currentSize > array.size() / 2)
        rehash();

    return true;
}
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because in the function, insertion in HashMap takes O(1) time.

Average case complexity is: O(1)

The average time complexity is O(1) because the length of the function is constant, and so as a result, inserting is done in constant time.



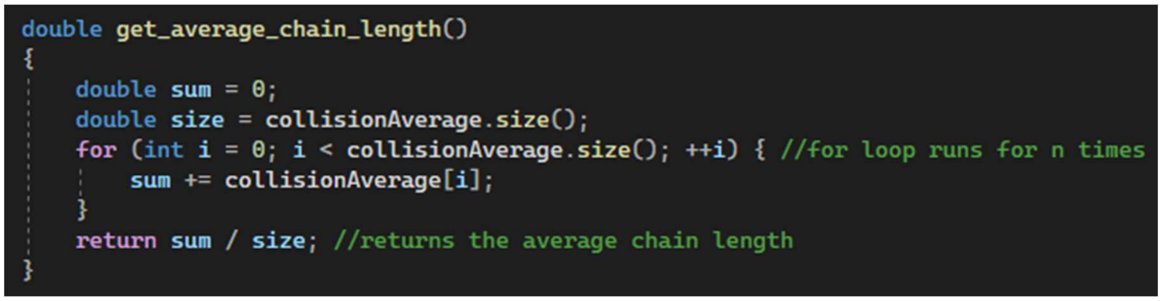
```cpp
//Function searches list for element to be deleted
 bool remove(const HashedObj& x)
 {
     int currentPos = findPos(x);
     if (!isActive(currentPos))
         return false;

     array[currentPos].info = DELETED;
     numberOfObjectsInTable--;
     return true;
 }
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because in the function, the deletion in a HashMap of a specific key will always be a O(1) statement.

Average case complexity is: O(1)

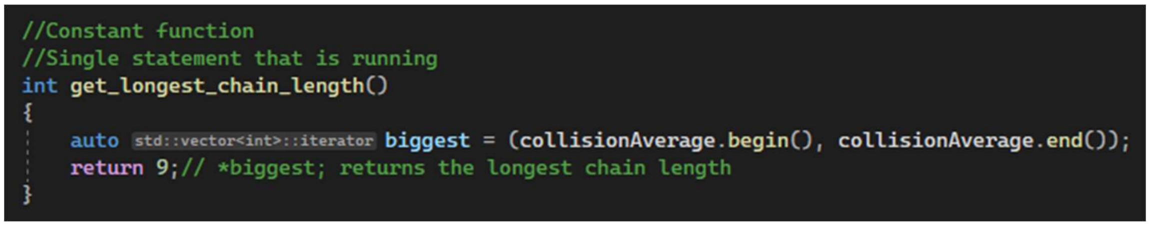
The average case complexity is O(1) because the function is only returning one statement.



```cpp
double get_average_chain_length()
{
    double sum = 0;
    double size = collisionAverage.size();
    for (int i = 0; i < collisionAverage.size(); ++i) { //for loop runs for n times
        sum += collisionAverage[i];
    }
    return sum / size; //returns the average chain length
}
```

Worst case complexity is: O(n)

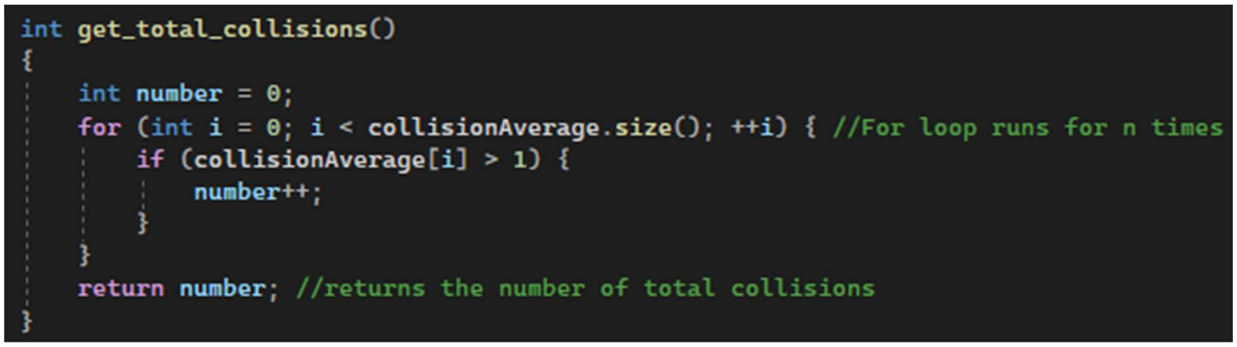
The worst case time complexity is O(n) because the function has a for loop, which runs for n times.



```cpp
//Constant function
//Single statement that is running
int get_longest_chain_length()
{
    auto std::vector<int>::iterator biggest = (collisionAverage.begin(), collisionAverage.end());
    return 9;// *biggest; returns the longest chain length
}
```

Worst case complexity is: O(1)

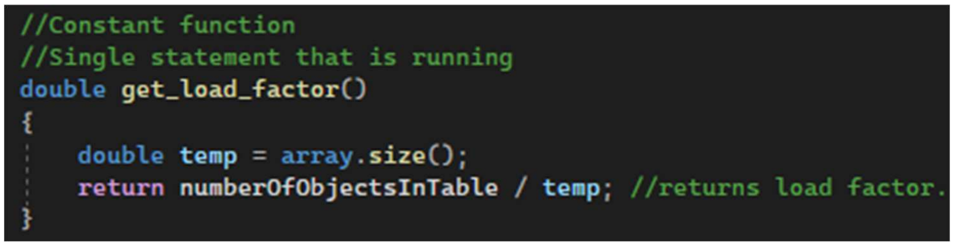
The worst case complexity is O(1) because the function has only one statement that is running.



```
int get_total_collisions()
{
    int number = 0;
    for (int i = 0; i < collisionAverage.size(); ++i) { //For loop runs for n times
        if (collisionAverage[i] > 1) {
            number++;
        }
    }
    return number; //returns the number of total collisions
}
```

Worst case complexity is: O(n)

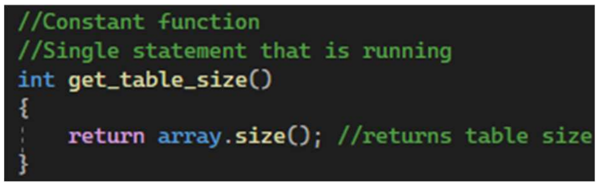
The worst case time complexity is O(n) because the function has a for loop, which runs for n times.



```
//Constant function
//Single statement that is running
double get_load_factor()
{
    double temp = array.size();
    return numberOfObjectsInTable / temp; //returns load factor.
}
```

Worst case complexity is: O(1)

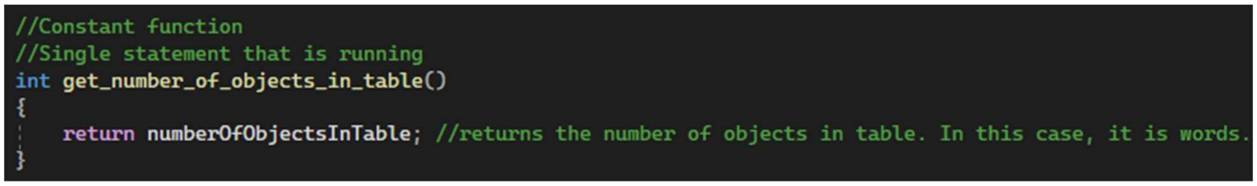
The worst case time complexity is O(1) because the function has only one statement that is running.



```
//Constant function
//Single statement that is running
int get_table_size()
{
    return array.size(); //returns table size
}
```

Worst case complexity is: O(1)

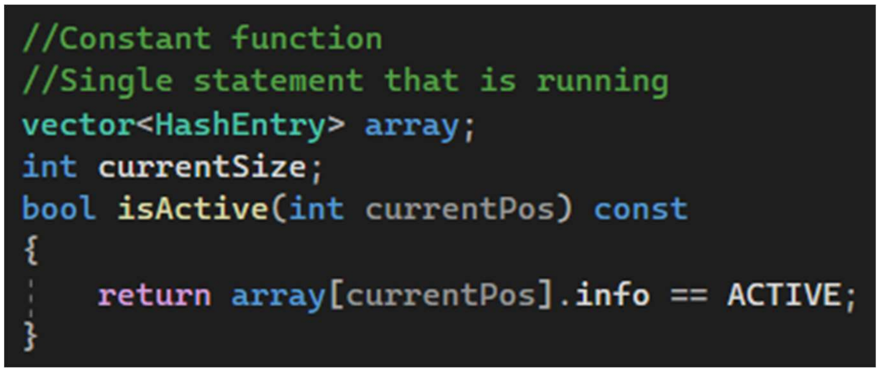
The worst case time complexity is O(1) because the function has only one statement that is running.



```
//Constant function
//Single statement that is running
int get_number_of_objects_in_table()
{
    return numberOfObjectsInTable; //returns the number of objects in table. In this case, it is words.
}
```

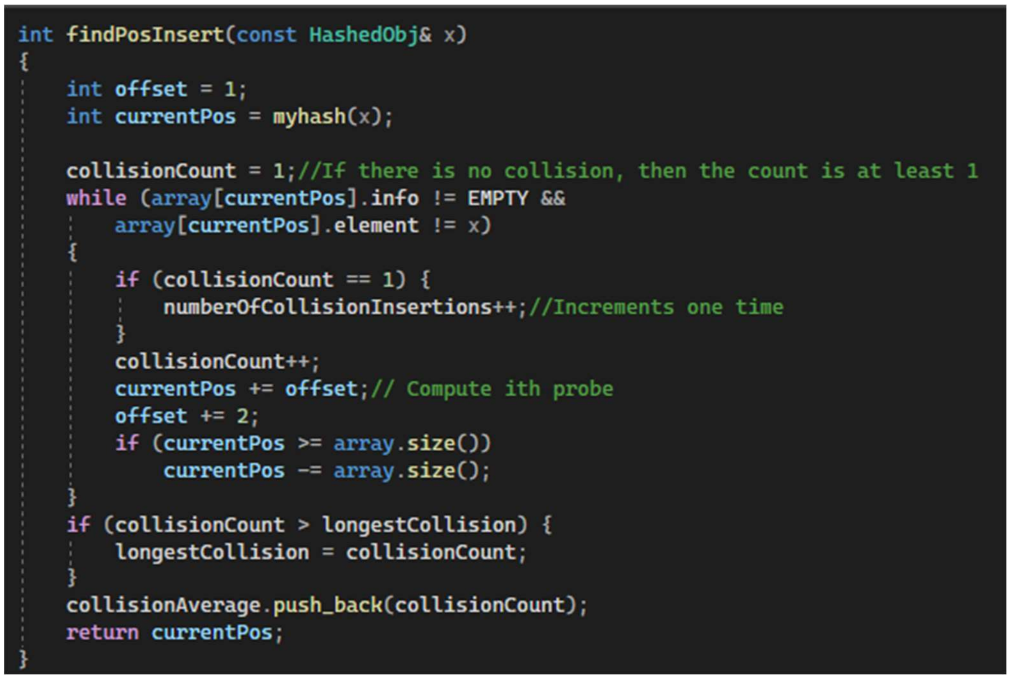
Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only one statement that is running.


```
//Constant function
//Single statement that is running
vector<HashEntry> array;
int currentSize;
bool isActive(int currentPos) const
{
    return array[currentPos].info == ACTIVE;
}
```

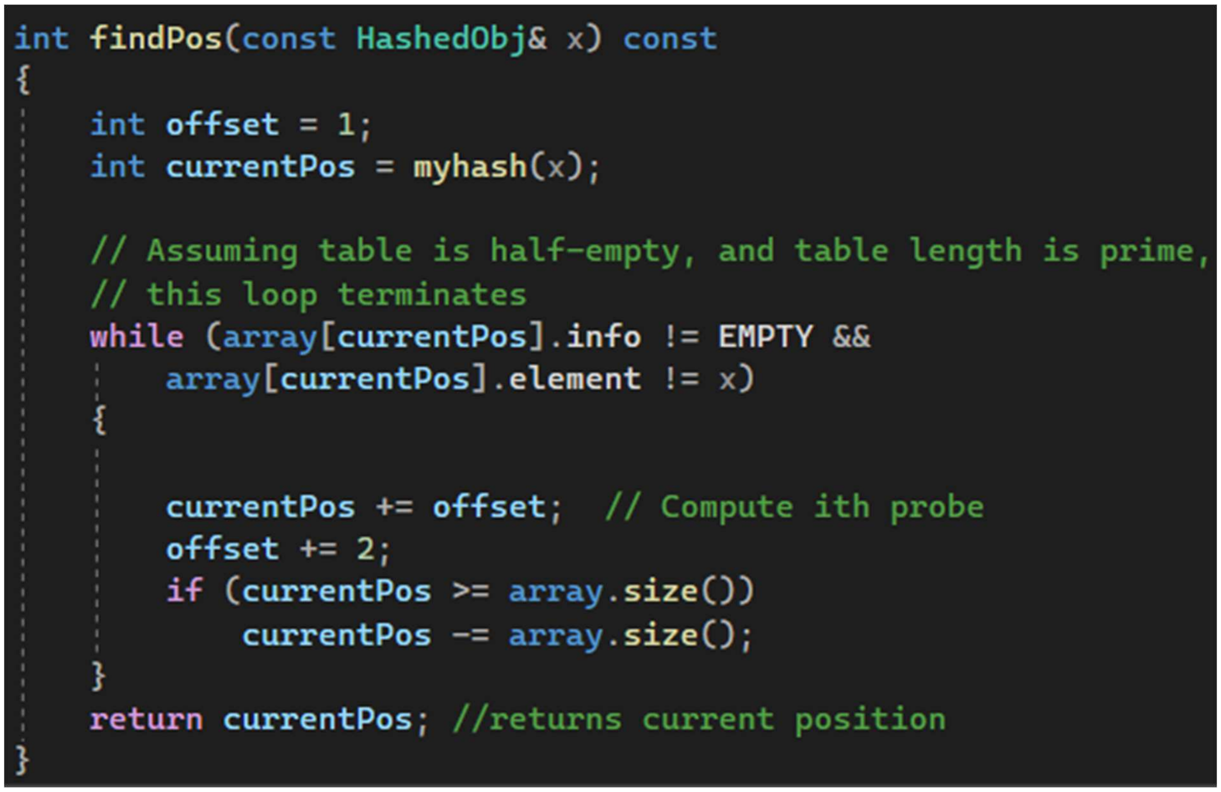
Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only statement that is running.


```
int findPosInsert(const HashedObj& x)
{
    int offset = 1;
    int currentPos = myhash(x);

    collisionCount = 1;//If there is no collision, then the count is at least 1
    while (array[currentPos].info != EMPTY &&
        array[currentPos].element != x)
    {
        if (collisionCount == 1) {
            numberOfCollisionInsertions++;//Increments one time
        }
        collisionCount++;
        currentPos += offset;// Compute ith probe
        offset += 2;
        if (currentPos >= array.size())
            currentPos -= array.size();
    }
    if (collisionCount > longestCollision) {
        longestCollision = collisionCount;
    }
    collisionAverage.push_back(collisionCount);
    return currentPos;
}
```

Worst case complexity is: O(n)

The worst case time complexity is O(n) because the function has a while loop running at O(n).



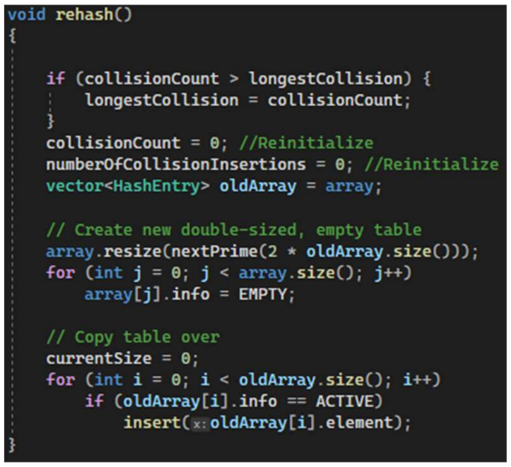
```cpp
int findPos(const HashedObj& x) const
{
    int offset = 1;
    int currentPos = myhash(x);

    // Assuming table is half-empty, and table length is prime,
    // this loop terminates
    while (array[currentPos].info != EMPTY &&
        array[currentPos].element != x)
    {

        currentPos += offset;  // Compute ith probe
        offset += 2;
        if (currentPos >= array.size())
            currentPos -= array.size();
    }
    return currentPos; //returns current position
}
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because in the function, searching for a key in HashMap takes O(1).

Average case complexity is: O(n)

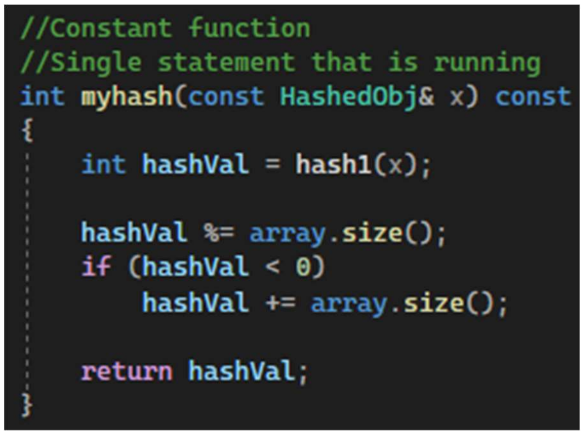
The average time complexity is O(n) because the function has a while loop, which runs for n times.



```cpp
void rehash()
{

    if (collisionCount > longestCollision) {
        longestCollision = collisionCount;
    }
    collisionCount = 0; //Reinitialize
    numberOfCollisionInsertions = 0; //Reinitialize
    vector<HashEntry> oldArray = array;

    // Create new double-sized, empty table
    array.resize(nextPrime(2 * oldArray.size()));
    for (int j = 0; j < array.size(); j++)
        array[j].info = EMPTY;

    // Copy table over
    currentSize = 0;
    for (int i = 0; i < oldArray.size(); i++)
        if (oldArray[i].info == ACTIVE)
            insert(x:oldArray[i].element);
}
```

Worst case complexity is: O(n)

The worst case time complexity is O(n) because the functions has a for loop that runs n times.

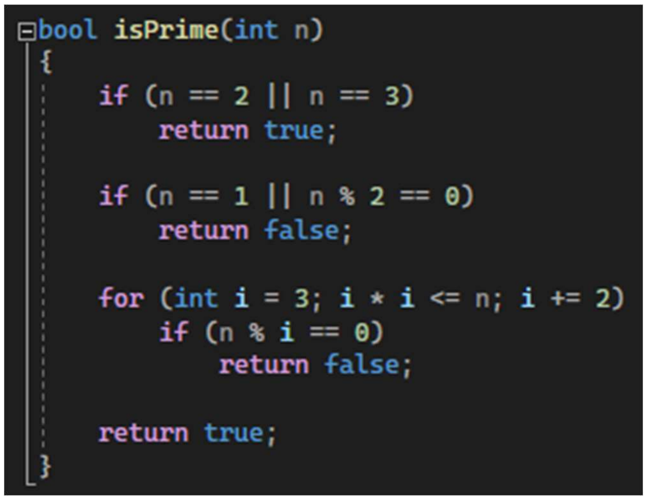


```cpp
//Constant function
//Single statement that is running
int myhash(const HashedObj& x) const
{
    int hashVal = hash1(x);

    hashVal %= array.size();
    if (hashVal < 0)
        hashVal += array.size();

    return hashVal;
}
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only one statement that is running.

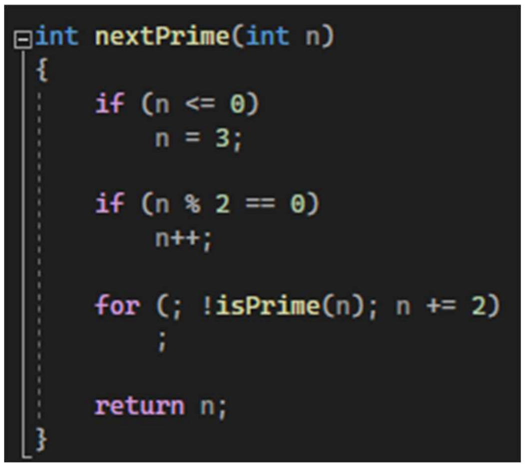
<u>QuadraticProbing.cpp Functions</u>



```cpp
bool isPrime(int n)
{
    if (n == 2 || n == 3)
        return true;

    if (n == 1 || n % 2 == 0)
        return false;

    for (int i = 3; i * i <= n; i += 2)
        if (n % i == 0)
            return false;

    return true;
}
```

Worst case complexity is: O(n)

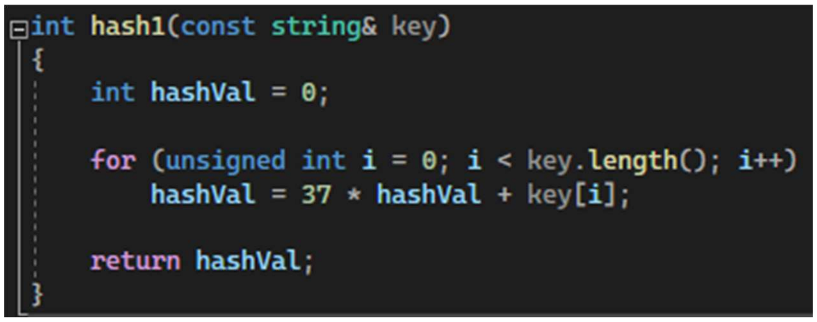
The worst case time complexity is O(n) because the function has a for loop, which runs for n times.



```
int nextPrime(int n)
{
    if (n <= 0)
        n = 3;

    if (n % 2 == 0)
        n++;

    for (; !isPrime(n); n += 2)
        ;

    return n;
}
```

Worst case complexity is: O(n)

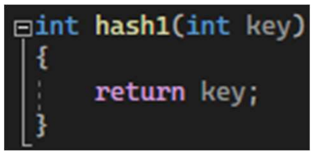
The worst case time complexity is O(n) because the function has a for loop, which runs for n times.



```
int hash1(const string& key)
{
    int hashVal = 0;

    for (unsigned int i = 0; i < key.length(); i++)
        hashVal = 37 * hashVal + key[i];

    return hashVal;
}
```

Worst case complexity is: O(n)

The worst case time complexity is O(n) because the function has a for loop, which runs for n times.

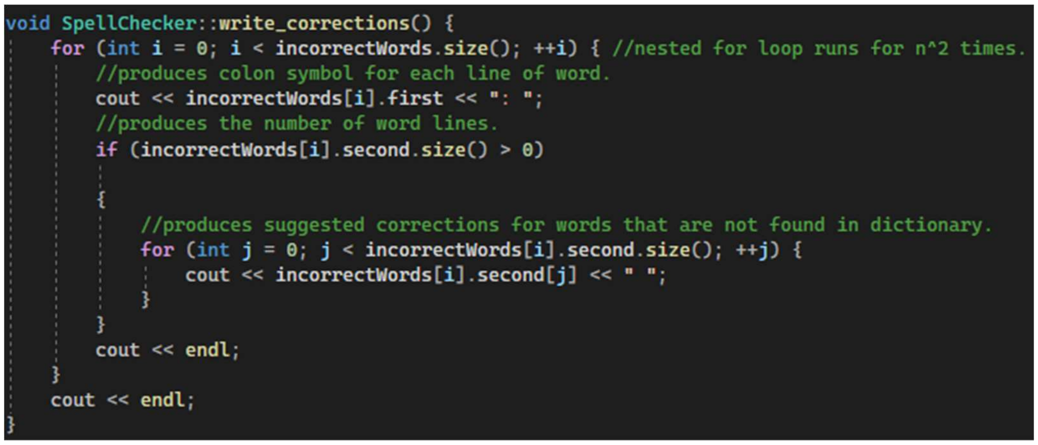


```
int hash1(int key)
{
    return key;
}
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only one statement that is running.

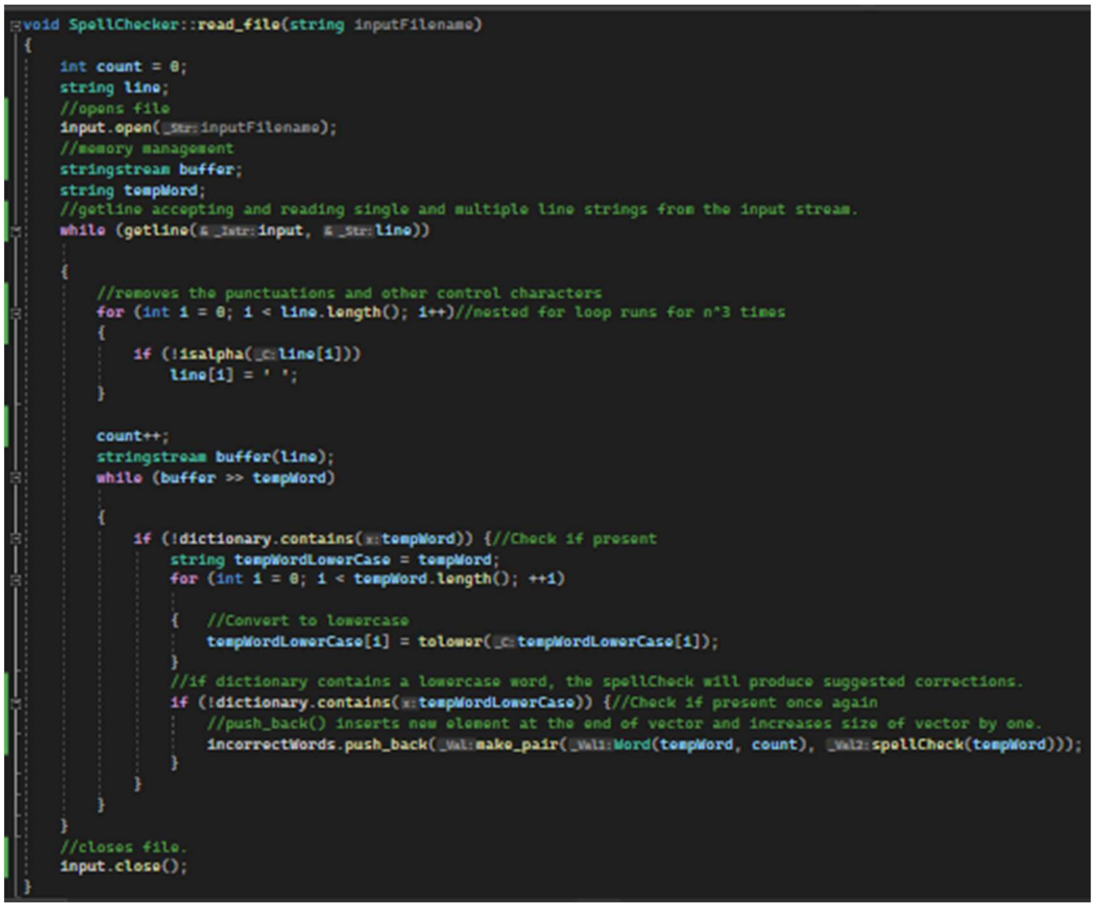
<u>SpellChecker.cpp Functions</u>



```cpp
void SpellChecker::write_corrections() {
    for (int i = 0; i < incorrectWords.size(); ++i) { //nested for loop runs for n^2 times.
        //produces colon symbol for each line of word.
        cout << incorrectWords[i].first << ": ";
        //produces the number of word lines.
        if (incorrectWords[i].second.size() > 0)

        {
            //produces suggested corrections for words that are not found in dictionary.
            for (int j = 0; j < incorrectWords[i].second.size(); ++j) {
                cout << incorrectWords[i].second[j] << " ";
            }
        }
        cout << endl;
    }
    cout << endl;
}
```

Worst case complexity is: O(n^2)

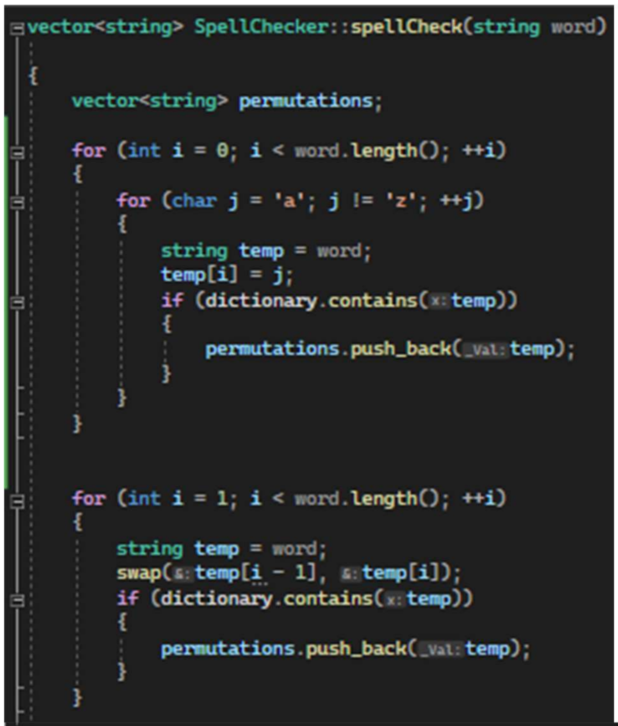
The worst case time complexity is O(n^2) because the function has a nested for loop which runs for n^2 times.



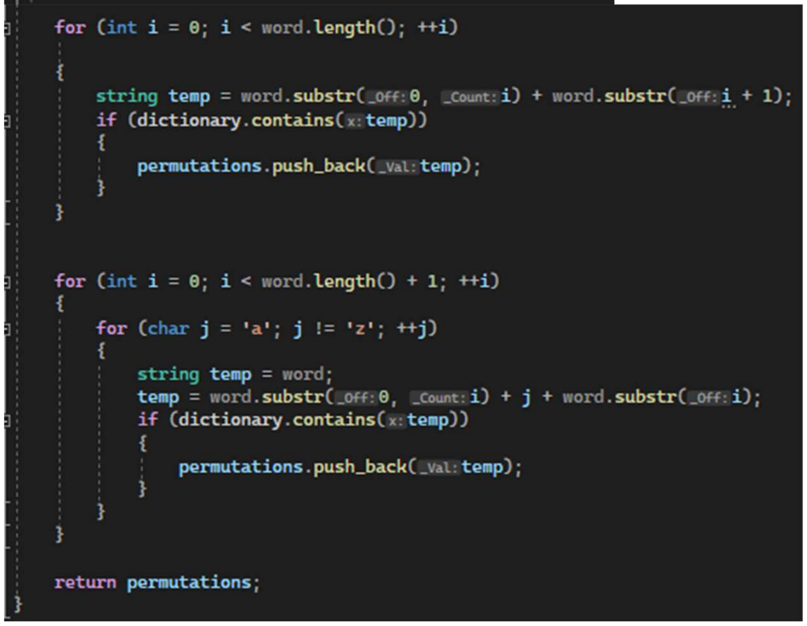
```cpp
void SpellChecker::read_file(string inputFilename)
{
    int count = 0;
    string line;
    //opens file
    input.open(_Str: inputFilename);
    //memory management
    stringstream buffer;
    string tempWord;
    //getline accepting and reading single and multiple line strings from the input stream.
    while (getline(&_Istr: input, &_Str: line))

    {
        //removes the punctuations and other control characters
        for (int i = 0; i < line.length(); i++)//nested for loop runs for n^3 times
        {
            if (!isalpha(_C: line[i]))
                line[i] = ' ';
        }

        count++;
        stringstream buffer(line);
        while (buffer >> tempWord)

        {
            if (!dictionary.contains(x: tempWord)) {//Check if present
                string tempWordLowerCase = tempWord;
                for (int i = 0; i < tempWord.length(); ++i)

                {   //Convert to lowercase
                    tempWordLowerCase[i] = tolower(_C: tempWordLowerCase[i]);
                }
                //if dictionary contains a lowercase word, the spellCheck will produce suggested corrections.
                if (!dictionary.contains(x: tempWordLowerCase)) {//Check if present once again
                    //push_back() inserts new element at the end of vector and increases size of vector by one.
                    incorrectWords.push_back(_Val: make_pair(_Val1: Word(tempWord, count), _Val2: spellCheck(tempWord)));
                }
            }
        }
    }
    //closes file.
    input.close();
}
```

Worst case complexity is: O(n^2)

The worst case time complexity is O(n^2) because in the function, inside the while loop there is nested for loop for doing operations.



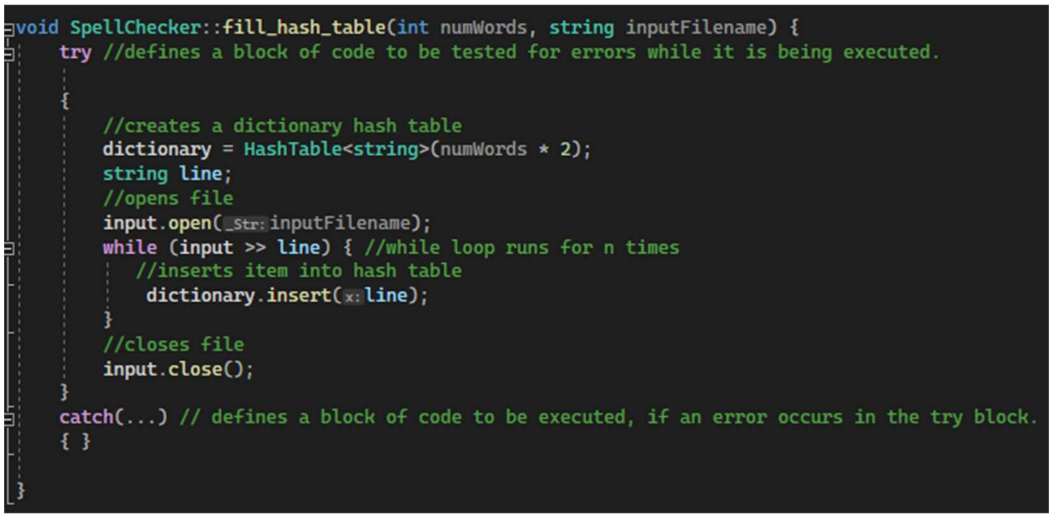
```
vector<string> SpellChecker::spellCheck(string word)

{
    vector<string> permutations;

    for (int i = 0; i < word.length(); ++i)
    {
        for (char j = 'a'; j != 'z'; ++j)
        {
            string temp = word;
            temp[i] = j;
            if (dictionary.contains(x: temp))
            {
                permutations.push_back(_Val: temp);
            }
        }
    }


    for (int i = 1; i < word.length(); ++i)
    {
        string temp = word;
        swap(&: temp[i - 1], &: temp[i]);
        if (dictionary.contains(x: temp))
        {
            permutations.push_back(_Val: temp);
        }
    }
```



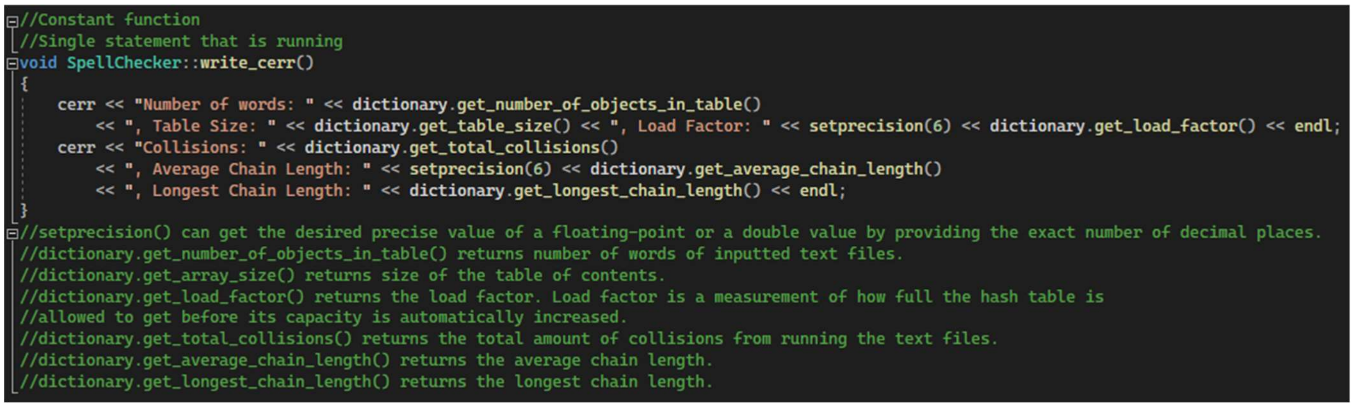
```
    for (int i = 0; i < word.length(); ++i)

    {
        string temp = word.substr(_Off: 0, _Count: i) + word.substr(_Off: i + 1);
        if (dictionary.contains(x: temp))
        {
            permutations.push_back(_Val: temp);
        }
    }


    for (int i = 0; i < word.length() + 1; ++i)
    {
        for (char j = 'a'; j != 'z'; ++j)
        {
            string temp = word;
            temp = word.substr(_Off: 0, _Count: i) + j + word.substr(_Off: i);
            if (dictionary.contains(x: temp))
            {
                permutations.push_back(_Val: temp);
            }
        }
    }

    return permutations;
}
```

Worst case complexity is: O(n^2)

The worst case time complexity is O(n^2) because the function has a nested for loop which runs for n^2 times.

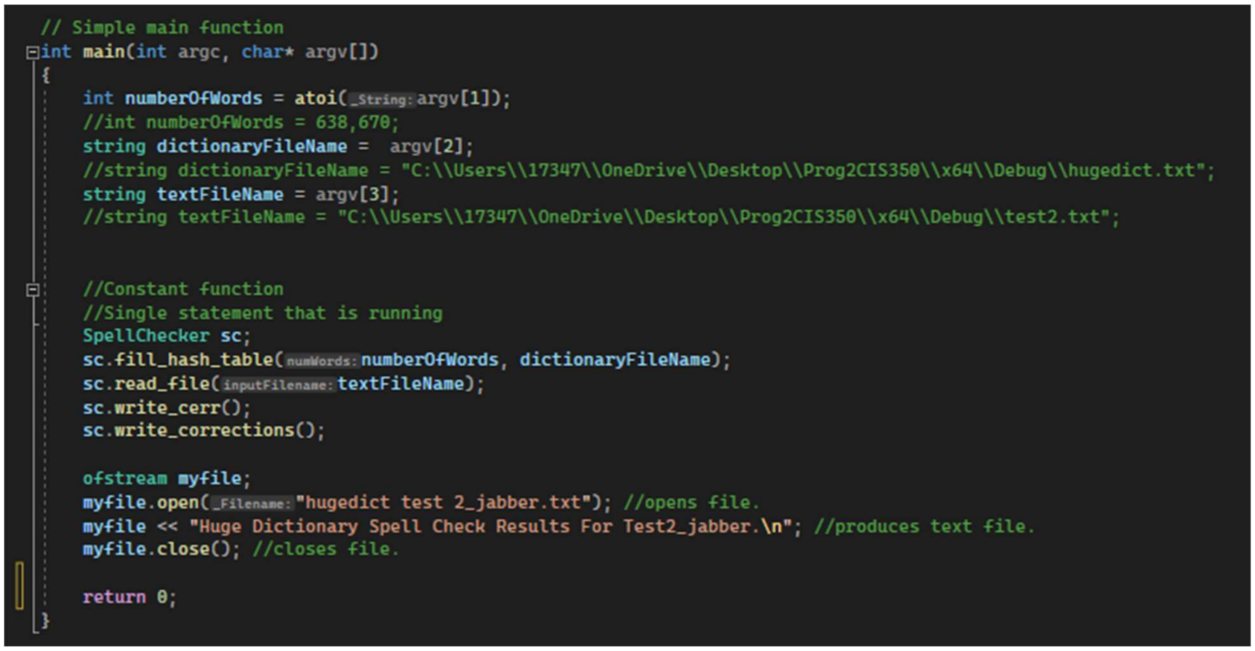

```
void SpellChecker::fill_hash_table(int numWords, string inputFilename) {
    try //defines a block of code to be tested for errors while it is being executed.

    {
        //creates a dictionary hash table
        dictionary = HashTable<string>(numWords * 2);
        string line;
        //opens file
        input.open(_Str: inputFilename);
        while (input >> line) { //while loop runs for n times
           //inserts item into hash table
            dictionary.insert(x: line);
        }
        //closes file
        input.close();
    }
    catch(...) // defines a block of code to be executed, if an error occurs in the try block.
    { }

}
```
Worst case complexity is: O(n)

The worst case time complexity is O(n) because the function has a while loop, which runs for n times.


```
//Constant function
//Single statement that is running
void SpellChecker::write_cerr()
{
    cerr << "Number of words: " << dictionary.get_number_of_objects_in_table()
        << ", Table Size: " << dictionary.get_table_size() << ", Load Factor: " << setprecision(6) << dictionary.get_load_factor() << endl;
    cerr << "Collisions: " << dictionary.get_total_collisions()
        << ", Average Chain Length: " << setprecision(6) << dictionary.get_average_chain_length()
        << ", Longest Chain Length: " << dictionary.get_longest_chain_length() << endl;
}
//setprecision() can get the desired precise value of a floating-point or a double value by providing the exact number of decimal places.
//dictionary.get_number_of_objects_in_table() returns number of words of inputted text files.
//dictionary.get_array_size() returns size of the table of contents.
//dictionary.get_load_factor() returns the load factor. Load factor is a measurement of how full the hash table is
//allowed to get before its capacity is automatically increased.
//dictionary.get_total_collisions() returns the total amount of collisions from running the text files.
//dictionary.get_average_chain_length() returns the average chain length.
//dictionary.get_longest_chain_length() returns the longest chain length.
```
Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only one statement that is running. The function is simply just outputting.

TestQuadraticProbing.cpp Function



```
// Simple main function
int main(int argc, char* argv[])
{
    int numberOfWords = atoi(_String: argv[1]);
    //int numberOfWords = 638,670;
    string dictionaryFileName =  argv[2];
    //string dictionaryFileName = "C:\\Users\\17347\\OneDrive\\Desktop\\Prog2CIS350\\x64\\Debug\\hugedict.txt";
    string textFileName = argv[3];
    //string textFileName = "C:\\Users\\17347\\OneDrive\\Desktop\\Prog2CIS350\\x64\\Debug\\test2.txt";


    //Constant function
    //Single statement that is running
    SpellChecker sc;
    sc.fill_hash_table(numWords: numberOfWords, dictionaryFileName);
    sc.read_file(inputFilename: textFileName);
    sc.write_cerr();
    sc.write_corrections();

    ofstream myfile;
    myfile.open(_Filename: "hugedict test 2_jabber.txt"); //opens file.
    myfile << "Huge Dictionary Spell Check Results For Test2_jabber.\n"; //produces text file.
    myfile.close(); //closes file.

    return 0;
}
```

Worst case complexity is: O(1)

The worst case time complexity is O(1) because the function has only statement that is running.