Spell Checker Project

Proposal

**<Selena Turner>**

**CSC316: Data Structures for Computer Scientists**

**<snturner@ncsu.edu>**

**North Carolina State University**

**Department of Computer Science**

**<April 11, 2016>**

Executive Summary

In the Problem Statement, the problem to be addressed by this project will be stated and explained. This statement will elaborate how this project can be of use to the client and why this project would help the client’s current system.

The Project goals & Benefits section will detail the most important outcomes of the project. While the Problem Statement lists the problem to be fixed by the project, this section shows what the project will accomplish that is more effective for the client than the present arrangement.

The algorithm psuedocode of the project will be located in the Algorithm Design section, as well as the design rationale and analysis of the algorithm. The psuedocode will act as an easier way to read the code and understand what is going on in the project, while the rationale and analysis will further explain the reasoning behind the psuedocode. The estimated runtime of the project will also be located in this section.

The UML diagram and the rationale for its design pattern for the project will be inserted into the Software Design section. This section shows a visual depiction of the algorithm and shows the relationship between Java classes to further elaborate on the effectiveness of the project.

In the Black Box Test Plan section, there are five test cases to test on the external workings of the project to ensure that the algorithm works as it was written. The test plan will be a guide for the user so that he or she knows how to run and use the program.

Lastly, the Task Plan is an overview of the different tasks for the author of the algorithm before the project is completed. The major parts of creating the proposal and code are listed so that the author has a schedule of when different sections are completed.

Table of Contents

[Problem Statement 1](#_Toc440806182)

[Project goals & Benefits 2](#_Toc440806183)

[Algorithm Design 3](#_Toc440806184)

[Proposed algorithm 3](#_Toc440806185)

[Algorithm Design Rationale 4](#_Toc440806186)

[Algorithm Analysis 4](#_Toc440806187)

[Software Design 5](#_Toc440806188)

[Design Pattern(s) 5](#_Toc440806189)

[Black Box Test Plan 6](#_Toc440806190)

[Task Plan 8](#_Toc440806191)

# Problem Statement

For this project, the problem is to create a program that can use a hash table to read in over 25,000 words from a dictionary and then check a text file for spelling errors. Currently, there is not a program specified by the project that was created using a hash table to spell check, so the present system would be to manually search through the text file and compare each word to the entire dictionary. This current system is extremely time consuming, since the runtime would be the amount of words in the dictionary times however many words are in the text file. This program would simplify the current system’s runtime, so instead of taking hours or even days to complete a spell check on the given text file, the runtime would be a few seconds.

# Project goals & Benefits

The major goal of this project is to create a program that effectively uses a hash table to search through a dictionary text file to make sure that a given text file has words that are correctly spelled. If the word is possibly not spelled correctly, the program will flag that word. At the end of the check, the program will output the alphabetical list of these misspelled words, along with the total number of dictionary words, the total number of text file words, the total number of misspelled words, and different numbers of probes that the program made. Other goals for this project include: creating tests for the project that cover over eighty percent of the code, commenting the code correctly with Javadoc, and having no flagged errors in the code.

With these goals, the project will be much faster than the current system of manually searching through the text file and dictionary. The user will also easily be able to access the system through the Graphical User Interface, which simplifies the way the spell checker will be run. The client will be benefitted by this system, especially if they run a business that deals with checking the spelling of words in a text file. This program can also be of use to the client, since it deals with code that, if thought of generally, can search through items and find targets. Being able to search and find objects does not have to be limited to spell checking, so for the client, this program leads to other possible systems that can be created.

# Algorithm Design

## Proposed algorithm

**Algorithm** hashCode(word)  
 **Input** the word from either the dictionary or the text file  
 **Output** a hash coded number depending on the word that is input

p <- a prime number //1  
 c <- 1 //1  
 for i <- size(word)-1 to 0 do //n  
 c <- word(charAt(i)) + (c \* p) //n  
 return c //1

**Algorithm** lookup(word)  
 **Input** the word to look up in the dictionary  
 **Output** the word, if it is not found in the dictionary  
 otherwise, null  
 A is an array that holds the dictionary words

hNum <- 0 //1  
 if word has ending er then //1  
 word <- word without er //1  
 hNum <- hashCode(word) //T(n) hashCode  
 else if word has ending es then //1  
 word <- word without es //1  
 hNum <- hashCode(word) //T(n) hashCode  
 else if word has ending s then //1  
 word <- word without s //1  
 hNum <- hashCode(word) //T(n) hashCode  
 else //1  
 hNum <- hashCode(word) //T(n) hashCode

i <- (hNum)mod(size(A)) //2  
 while A has next space do //n+1  
 if A[i] does not equal word then //n-1  
 i++ //n-1  
 else //n-1  
 return A[i] //n-1  
 return null //1

**Algorithm** insert(word)  
 **Input** the word from the dictionary text file  
 **Output** no output

hNum <- hashCode(word) //T(n) hashCode  
 i <- (hNum)mod(size(A)) //2  
 A[SIZE] <- new array //1  
 while A has next space do //n+1  
 if A[i] is empty then //n-1  
 A[i] <- word //n-1  
 break //n-1  
 else //n-1  
 i++ //n-1

## Algorithm Design Rationale

The hashCode algorithm creates a specific number for each word so that they can be stored and looked at very easily.

The lookUp algorithm takes a word from the text file and then compares it to where the same word should be in the dictionary array depending on the hashCode value. This algorithm also makes sure that the word is a root word, so it doesn’t contain the extra end letters that can be written in normal text.

The insert algorithm takes a word from the dictionary text file and uses the generated hashCode value to put the word into an array. If the array space is already taken, it will go to the next array space. This linear probing takes away the need for collision resolution, because if there is a collision, the probe moves to the next spot in the array. Also, taking the hashCode value and modding it with the size of the dictionary array satisfies the need for compression in the HashTable because it still inserts into an unique space but makes sure the hashCode number does not get too large.

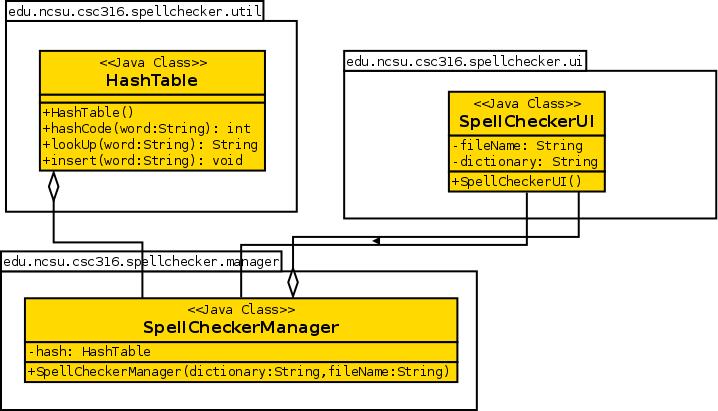
## Algorithm Analysis

Algorithm hashCode:  
T(n) hashCode = 2n+3  
T(n) hashCode is in O(n) because the highest bound of T(n) is n  
n represents the word size

Algorithm lookUp:  
T(n) lookUp = 4\*(T(n) hashCode)+8 = 4\*(2n+3)+8 = 8n+12+8 = 8n+20  
T(n) lookUp is in O(n) because the highest bound of T(n) is n  
n represents the dictionary array size

Algorithm insert:  
T(n) insert = (T(n) hashCode)+1 = (2n+3)+1 = 2n+4  
T(n) insert is in O(n) because the highest bound of T(n) is n  
n represents the dictionary array size

# Software Design



## Design Pattern(s)

For this project, I implemented the Model-View-Controller pattern. This pattern deals with the interaction between the user and the program, with the Model making up the data of the program, the View having the contents of the program, and the Controller being the way the user can interact with the program.

In the UML diagram above, the Model is made up of the SpellCheckerManager class, because all of the data will be stored in it. The View is made up of the HashTable class, because the contents that make up the data will be coded in it. The Controller is made up of the SpellCheckerUI class, because the Graphical User Interface is the way the user can interact with the program. There is an association relationship between SpellCheckerManager and SpellCheckerUI, and two “has-a” relationships, one between SpellCheckerManager and SpellCheckerUI, and the other between SpellCheckerManager and HashTable. These classes have “has-a” relationships because SpellCheckerUI “has-a” SpellCheckerManager variable and SpellCheckerManager “has-a” HashTable.

# Black Box Test Plan

To start up the program, the user will right click on the Graphical User Interface and select Run As -> Java Application. The system will then pop up with a file chooser. Select the file indicated by each of the test cases.

The first test deals with an “empty” dictionary file. This file will be named “emptyDictionary.txt” and will contain no text.

The second test deals with a text file with five misspelled words. It will be called “fiveWords.txt” and has the following input:

No ne wunts to sove th problom.

The third test deals with an “empty” text file. This file will be named “emptyText.txt” and will contain no text.

The fourth test deals with a text file with no misspelled words. This file will be named “correctWords.txt” and has the following input:

Having fun yet? I am!

The fifth test deals with a dictionary file with only ten words in it. This file will be named “smallDictionary.txt” and has the following input:

have never will to the start I a

The text file for this test will be called “smallText.txt” and contains the following input:

I am a great file maker. I have never done anything incorrectly.

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Expected Results | Actual Results |
| Empty dictionary file | The user starts up the program.  The user selects file “emptyDictionary.txt” from the file chooser.  The user quits the system. | An error message is displayed. |  |
| Five misspelled words | The user starts up the program.  The user selects the dictionary from the file chooser.  The user selects file “fiveWords.txt” from the file chooser.  The user quits the system. | The five misspelled words are printed alphabetically. The number of dictionary words, text file words, and misspelled words are printed. The total number of probes, the average number of probes per word, and the average number of probes per lookup operation are printed. |  |
| Empty text file | The user starts up the program.  The user selects the dictionary from the file chooser.  The user selects file “emptyText.txt” from the file chooser.  The user quits the system. | An error message is displayed. |  |
| No misspelled words | The user starts up the program.  The user selects the dictionary from the file chooser.  The user selects file “correctWords.txt” from the file chooser.  The user quits the system. | The number of dictionary words, text file words, and misspelled words are printed. The total number of probes, the average number of probes per word, and the average number of probes per lookup operation are printed. |  |
| Small dictionary file | The user starts up the program.  The user selects the dictionary “smallDictionary.txt” from the file chooser.  The user selects file “smallText.txt” from the file chooser.  The user quits the system. | The seven “misspelled” words are printed alphabetically. The number of dictionary words, text file words, and misspelled words are printed. The total number of probes, the average number of probes per word, and the average number of probes per lookup operation are printed. |  |

# Task Plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task Description | Owner | Planned  Start Date | Planned  End Date | Status |
| Part 1: Start   * Start Executive Summary, Problem Statement, Project Goals & Benefits, Algorithm Design, Software Design, and Black Box Text Plan | Selena T | 4/6/16 | 4/9/16 | Complete |
| Part 1: Finish   * Finish any proposal items that aren’t done yet | Selena T | 4/9/16 | 4/11/16 | Complete |
| Part 2: Start   * Create code skeleton | Selena T | 4/12/16 | 4/15/16 | NA |
| Part 2: Test   * “Complete” code * Start testing for bugs | Selena T | 4/15/16 | 4/19/16 | NA |
| Part 2: Finalize   * Finish the code * Add/fix comments * Make sure all Jenkins tests pass | Selena T | 4/19/16 | 4/22/16 | NA |

**Team Contact Information**

Selena Turner, snturner@ncsu.edu