Assignment 2

Problem 1

Data Structures and Algorithms Report

2401ICT

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Student #** s2916255  **Name** Scott McMullan | scott.mcmullan@griffithuni.edu.au |  |
|  |  |  |

Contents

[Problem Statement 1](#_Toc321140622)

[User Requirements 2](#_Toc321140623)

[Software Requirments 3](#_Toc321140624)

[Software Design 3](#_Toc321140625)

[Requirment Acceptance Tests 3](#_Toc321140626)

[Detailed Softare Testing 3](#_Toc321140627)

[User Instructions 3](#_Toc321140628)

# Problem Statement

*This section contains a few statements on what was needed to be done for the assignment.*

## Tree Dictionary Generator

In this part of the assignment, the goal was to create a Python program that accepts a text file from input and uses three different tree structures (Trie, BST and AVL) to generate python objects from the words and judge the performances of all 3 compared to each other.

## Spell Checker

As an extension of the tree dictionary generator, the previously generated Trie tree object must be saved to disk using the Python pickle package. This tree will be loaded into another Python program, the spell checker, and this “dictionary” will be used to compare against a second word file with a number of mistakes. If the program picks up a mistake, it must provide suggestions on how to correct the mistake.

# User Requirements

*This section details how the user is meant to interact with the program.*

Both the tree dictionary generator and the spell checker are written to be highly automated. Really, all that is required from the user in this project is to press the run button in the IDE. However, if the user wants to change the input files to test different texts or specific words, then they must change the file path in the source code.

# Software Requirements

*Detailed are the requirements of the program, based on the assignment handout.*

## Dictionary Generator

1. The program must generate 3 tree objects from pre-written classes, Trie, AVL and BST (Binary Search Tree).
2. The program must read in words from an input file, sanitizing all words skipping numbers, words with numbers and concatenating hyphenated words.
3. The program shall generate a Trie tree from the input words and from that tree, a list will be generated from the sanitized words. These words in turn will be added to the BST and AVL tree.
4. The words will be read in in chunks, starting from 10% and incrementing 10% to 100% of the word file.
5. As the words are read into the different trees, statistics will be recorded and then displayed in 5 different plots for each tree:

* Time taken to read in words
* Time taken to read out words
* Total number of nodes
* Total height of tree
* Tree size once pickled to disk.

## Spell Checker

1. From the previous program, a Trie tree dictionary was generated. The program will un-pickle the object that was saved to disk previously and load it to a variable.
2. Next, the program will read in another word file that contains mistakes. This file will be read in using the same method as the previous program.
3. One those steps are completed the program will detect spelling errors in any of the words that have been read from the new file.
4. If misspelled words have been detected, then the word will be underlined and highlighted, along with being reported to the user as well as the line it is in.
5. Finally, suggested spelling corrections will be reported to the user based on a detection algorithm.

# Software Design

*Diagrams and pseudo code for the programs.*

## List of All Functions in The Dictionary Generator

* read\_file()
* This function reads, sanitizes and returns a list of words from an inputted word file and returns then as a list.
* The first and only input parameter is “file”. It is of type file and it used to read the lines and then words, adding them to a list and returning it.
* This function creates a list of strings and returns it, which is assigned to an object in the main program.
* The return value of the function is a list of strings.
* pickle\_object()
* This function takes an object from main and uses Python’s pickle package to save that object to disk.
* This function takes a tree object, “object’ and also a string, “fileName”. The object is the actual tree object that is to be pickled (saved to disk). “fileName” is the path that the object is to be saved to.
* The side effect of this function is quite major. It creates and saves a new file to disk, which is an object that was created in the main program.
* The function returns a pickled object. (I don’t even know why I need this function)

## List of All Functions in The Spell Checker

* corrections\_helper()
* This function is a function that is used in another function, corrections(). It is a helper function that test 2 words to see if they are close in terms of what letters comprise each word, and if the words are “close” then the function will count by how many letters the 2 words differ (allowing for approximately a third of the length of the word being tested) and return that count.
* 2 strings are parsed into this function, “word” and testWord”. “testWord” is the word that is being tested against the misspelled word, “word”.
* The one side effect of this function is it controls whether a word is added to a “suggestions” list in another function. It judges whether a word is “close” enough to the tester.
* This function returns either the count of how many words are *not* similar or a word greater than 4.

## List of All Data Structures in The Dictionary

* wordFile, file
* wordlist, list
* baseTrie, Trie
* treeStats, dictionary.
* percentage, float.
* sanitisedWordCount, int
* bst, Binary Seach Tree
* avl, AVL tree
* newLength, int
* newList, list
* start, timer
* end, timer

## List of All Data Structures in The Spell Checker

* count, int
* charCount, int
* suggestions, list
* wordFile, file
* wordlist, lit
* misspelled, list
* linesInFileSantised, dictionary.
* linesInFileOG, dictionary.
* lineCounter, int

# Detailed Design – Tree Dictionary

**Read\_file** {

Create a list

For each line in the file, split the lines into words.

Sanitise the words and return them

}

**Pickle**

Save the object to disk.

For each word in wordlist, add it to base trie.

While the count of words is less than 100% {

* add the current percentage to treeStats
* create the new word list, of the new length
* time how long it takes the read in all words for each tree
* append it to treeStats
* time how logn it takes to read out all words, for each tree
* append it to treeStats
* count the total amount of nodes for each tree
* append it to treeStats
* judge the height of each tree
* append it to treeStats
* pickle each tree and get the size of each on disk
* append it to treeStats

}

Display all information in treeStats using 5 different plots.

# Detailed Design – Spell Checker

**Corrections\_helper() {**

For each letter in the word, if it’s in testWord, increment the letter counter

If (the letter count greater than or equal to approximately a third of the length of the testWord) {,  
then count how many words *aren’t* matching.

}

Else {

Return 4

}

Return count

**}**

**Corrections() {**

For each word in the trie list

If (the length of that word is the same as the length of the test word) and  
if ( the return value of Corrections\_helper is less than or equals to 3 (as in, if there’s no more than 3 mitakes))  
Add the word to the suggestion list.

Finally, print all the suggested words highlighted green.

**}**

**Read in the word file, sanitizing all lines.**

**Blah blah blah, just print the errors in a special box.**

# User Instructions

Click the “run” button.