COSC2320: Data Structures and Algorithms
HW5: Sorting and Searching

1 Introduction

You will create a C++ program that implements a primitive spell-checker, using an efficient sorting and searching algorithms.

In essence a spell checker works by comparing all the words in a document to words in a ”dictionary”. If a word is not found in the dictionary, the program assumes that it is a misspelling. In order to efficiently spell check a document you need to have your dictionary in a sorted array, and you need an efficient way of determining if a word can be found or not in the dictionary array.

For this homework you will read a dictionary file into a linked list. Once the file is read and the number of words is known, you should allocate an array and use it to sort your dictionary file if it is not already sorted (this could be verified while you move it to the array). If there are duplicate words, only one instance of the word should be kept. You should also delete the original list to free memory.

The file to be checked will be loaded into memory in the same way as in previous homeworks. You should use an efficient search algorithm (Binary Search or AVL Binary Search Trees) to determine if each word of the input file is present in the dictionary. You should produce a list with all the words that are misspelled in the document.

2 Examples

INPUTS

```
dictionary.txt
-----------------------------------------------
alpha beta gamma delta epsilon zeta eta theta
iota kappa lambda mu nu xi omicron pi
rho sigma tau upsilon phi chi psi omega
<EOF>

input1.txt
-----------------------------------------------
alfa alpha. halpha beta veta, kapa alfa
<EOF>
```

```
input2.txt
-------------------------------------------
alphA?

mega omega

<EOF>

input3.txt
-------------------------------------------
alpha beta delta zeta epsilon
theta
mu
nu<EOF>

input.script
-------------------------------------------
read(L1,'input1.txt')
read(L2,'input2.txt')
read(L3,'input3.txt')
check(R1,L1)
check(R2,L2)
check(R3,L3)
write(R1,'out1.txt',forward)
write(R2,'out2.txt',reverse)
write(R3,'out3.txt')
union(R1,R2,U1)
write(U1,'out4.txt')
intersection(R1,R3,I1)
write(U1,'out5.txt')
```

RESULTS

```
out1.txt
-------------------------------------------
alfa 2
halfa 1
kapa 1
veta 1
<EOF>

out2.txt
-------------------------------------------
mega 1
<EOF>

out3.txt
-------------------------------------------
<EOF>
```

```
out4.txt
-----------------------------------------------
alfa 2
halfa 1
kapa 1
mega 1
veta 1
<EOF>

out5.txt
-----------------------------------------------
<EOF>
```

3 Program and output specification

The main program should be called "spellchecker". Syntax:

```
spellchecker script=input.script;dictionary=dictionary.txt
```

When asked to output an error list to a file you must write one word per line followed by a space and the number of occurrences of the misspelt word. Please note that there is no change from previous homeworks.

4 Requirements

- Your program should first verificate that the dictionary is sorted ($O(n)$ time). If the dictionary is not sorted, you must use an efficient sorting algorithm to order it. Efficient means an $O(n \log n)$ algorithm. You can use any algorithm that was shown in class, but Merge Sort would be preferred. Keep in mind that the dictionary can contain thousands of unique words, so you should also try to optimize the amount of memory you will be using.
- You should also use an efficient searching algorithm in order to find out if a word is misspelt. Binary Search or AVL Binary Search Trees are allowed, but the latter will result in bonus points.
- Please specify in the README file what sorting and searching algorithms you are using, as well as a short description of their characteristics.
- Since it is possible that some letters will be capitalized, you should convert all words to lower case.
- As in Homework 0, there could be punctuation signs at the end of the words. Therefore "alpha" and "alpha??" should be considered the same word for the purposes of the spell checker. We will not include words with punctuation in the middle such as "we're" or "part-time".
- The aim of this program is only to find words that are not in the dictionary. You are not required to do any approximation matching (suggesting correct spelling). This goes beyond the scope of this assignment.
- You will modify the script parser to accept the new instruction "check". "check" will have two arguments: the name of the destination list where you should store all the misspelt words and the name of the list to be checked. We will provide guidance, but we will not generate a new version of the parser.

- **Note** that the sample script includes a union and an intersection. We will check that those functions still work in the hard cases. Remember to remove the extra variables used in the previous homework to count instructions.
- When writing to a file you will be asked to write either in alphabetical order or in reverse alphabetical order. Since your misspelling lists can be created in order, this will involve writing starting from the head or from the tail. If no direction is specified you can assume alphabetical order
- The program should not halt when encountering errors in the script. It should just send a message to the log file and continue with the next line. The only error that is unrecoverable is a missing script file or a missing dictionary.
- Do not use the STL library.
- Your program should write error messages to a log file (and optionally to the screen). Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, zeroes and inconsistent information. Each exception will be -10.
- Test cases: Your program will be tested with 10 test scripts, going from easy to difficult. You can assume 80% of test cases will be clean, valid input files. If your program fails an easy test script 10-20 points will be deducted. A medium difficulty test case is 10 points off. Difficult cases with specific input issues or complex algorithmic aspects are worth 5 points.
- A program not submitted by the deadline is zero points. A non-working program is worth 10 points. A program that does some computations correctly, but fails several test cases (especially the easy ones) is worth 50 points. Only programs that work correctly with most input files that produce correct results will get a score of 80 or higher. In general, correctness is more important than speed.