CIS 415
Find anagrams in a dictionary

Due at 11:59pm on Friday, 12 October 2018

1 Introduction

A dictionary file contains a list of words with out duplicates. There are however capitalized and non-capitalized words, such as "A" and "a". Anagrams are words that can be spelled using the same set of letters, such as "restful" and "fluster". An anagram family is a sorted set of letters such as "eflrsu" for the previous anagram. The goal is to find all anagrams in the dictionary. Every word that can be spelled using the same set of letters will be grouped together.

2 Specification

Anagrams are case-insensitive, thus "Salem" and "meals" are the members of the same anagram family "aelms". The capitalization of the word must be preserved for output, so "salem" is not acceptable. Duplicate words such as "A" and "a" are both words in the same anagram family "a".

The program shall do the following:

- Open a file specified on the command line or read from stdin.
- Read each line as a word and remove the '$\backslash n$' from it.
- Add the word to an existing anagram family or create a new one.
- Enumerate the anagram families, printing to a specified file or stdout:
  print the sorted set followed by ':'
  print the number of anagrams followed by '$\backslash n$'
  print a tab '$\backslash t$' before each anagram, then anagram, followed by '$\backslash n$'
- Do not print out anagram families whose size is 1.
- Clean up all memory data structures and close all files, Detect and handle erroneous command line input.

Here is an example of expected output in a file:

```
aabeir:3
        Aberia
        Baeria
        Baiera
abet:5
        abet
        bate
        beat
        Beta
        beta
```

3 Design

A header specification for two data structures and functions will be provided, all functions need to be implemented appropriately. StringList is a structure to store a list of strings, AnagramList is a structure to store an anagram family and the associated words in a StringList.

```
/*List of strings structure*/
        struct StringList
        {
                struct  StringList* Next;
                char *Word;
        };
/*List of anagrams structure*/
        struct AnagramList
        {
                struct StringList *Words;
                struct AnagramList *Next;
                char *Anagram;
        };

/*Create a new string list node*/
        struct StringList *MallocSList(char *word);
/*Free a string list, including all children*/
        void FreeSList(struct StringList **node);
/*Append a string list node to the end/tail of a string list*/
        void AppendSList(struct StringList **head, struct StringList *node);
/*Format output to a file according to specification.*/
        void PrintSList(FILE *file,struct StringList *node);
/*Return the number of strings in the string list.*/
        int SListCount(struct StringList *node);
```

```
/*Create a new anagram node, store the word as sorted char array*/
/* add S list node with the word.*/
        struct AnagramList* MallocAList(char *word);

/*Free an anagram list, including anagram children and string list words.*/
        void FreeAList(struct AnagramList **node)

/*Format output to a file, print anagram list with words, according to spec*/
        void PrintAList(FILE *file,struct AnagramList *node);

/*Add a new word to the anagram list: Search the list and add the word*/
/*Search with a sorted lower case version of the word. */
        void AddWordAList(struct AnagramList **node, char *word);
```

The provided header file may not be changed. AddWordAList will create an AnagramList if (*node) is null, the first time it will be. You need to make sure that the anagram is stored in sorted lower case. You need to make sure that AddWordAList compares a lower case version of word, it should be easier for comparison if the word is also sorted. Make sure that SList stores the word with proper case and does not sort the word. You will need to make a copy the string and free it after usage to do this correctly. Make only one copy of the word while searching, otherwise N copies will be made which drastically increase the runtime length and A-Z will not run. Note: you will need to implement a function to un-capitalize a character array and a function to sort the character array. Quick sort is recommended, please cite your source.

A compiled working test version of the main driver program will be provided to test with. The program reads from input and writes to output adding each word to the anagram list, then prints the anagram list, and cleans up. The main program file consists of the following sudo code:

```
open input file or stdin
open output file or stdout
exit if either file is invalid
while getline from input file
        add word to a list
print a list
free a list
close input and output file.
```

Usage is as follows:

- ./anagram // read from stdin, write stdout
- ./anagram file1 // read from file1, write stdout
- ./anagram file1 file2 // read from file1, write file2.

This can be tested with such commands as:

- ./anagram <file1 >file2 //test for reading from stdin/ write stdout.
- ./anagram file1 >file2 // test for reading from file1 / write stdout.
- ./anagram file1 file2 // test for reading from file1 / write file2.

Several dictionaries will be provided: A, A-B, A-B-C, and A-Z. Output files for A,A-B,A-B-C,A-Z will be provided.

4 Implementation

You need to implement all of the functions in the header file along with the main driver program. Sample input/output files will be provided, the output file should match exactly. You can use the command line "diff file1 file2" to verify that two files are identical.

Note you will be penalized heavily if valgrind reports any memory errors. After you have a working version of the program, you need to test it using valgrind to make sure it has no memory errors. If valgrind indicates any problems with your codes use of heap memory, it is usually an indication that you are doing something very wrong that will bite you eventually; you must fix your code to remove all such problem reports.

A gzipped tar archive containing source files, a makefile, and test input and output files is available on Canvas. In addition to anagram.h and makefile there is also an object file with a working version of the main driver program. This will allow you to test your implementation of SList and Alist before implementing your own driver program.

5 Approach: Building and debugging anagram

You should first study the anagram.h structures and functions to understand what they provide, then analyze the sudo code to see how it uses the functions. First construct anagram.c by copy pasting anagram.h and deleting the # statements and structure declarations. You will then need to implement each function, some functions will call other functions, such as MallocAList will call MallocSList. Once you have all the functions implemented you can run the following for verification:

```
./anagramtest <A.dict | diff - A.out
./anagramtest <A-B.dict | diff - A-B.out
```

Once you have this working then implement your own main.c and call the functions in anagram.h. The make file will build the two file anagram and anagramtest.

6 Submission

You will submit your solutions electronically by uploading a gzipped tar archive via Canvas. Your TGZ archive should be named "¡duckid¿-project0.tgz", where "¡duckid¿" is your duckid. It should contain your "anagram.c", your "main.c" and a document named "report.pdf" or "report.txt", describing the state of your solution, and documenting anything of which we should be aware when marking your submission. Do not include any other files in the archive; in particular, this means that UNDER NO CIRCUMSTANCES should you change anagram.h. Your submission will be tested against the issued versions of these files; if you change them, then your code will not work correctly and you will lose a significant number of marks. A 20% penalty will be assessed if you do not follow these submission instructions. Section 7 describes how to follow these directions for those that are unsure.

These files should be contained in a folder named "¡duckid¿". Thus, if you upload "rcasitaproject0.tgz", then we should see something like the following when we execute the following command:

```
tar -ztvf rcasita-project0.tgz
-rw-rw-r-- rcasita/None 3670     2015-03-30 16:30 rcasita/main.c
-rw-rw-r-- rcasita/None 5125     2015-03-30 16:37 rcasita/anagram.c
-rw-rw-r-- rcasita/None 629454   2015-03-30 16:30 rcasita/report.pdf
```

Each of your source files must start with an "authorship statement", contained in C comments, as follows:

- state your name, your duckid, and the tile of the assignment (CIS 415 Project 0)
- state either "This is my own work." or "This is my own work except that ..." as appropriate.

You code will be compiled and tested against an unseen dictionary. We will also be checking for collusion; if collusion between two or more students is detected, each student involved will immediately receive an F in the course. We have extremely good tools for detecting collusion; if you collude, we will catch you and you will fail the course

7 How to create the correct archive to submit

- First, determine your duckid (your uoregon.edu email without the "@uoregon.edu"). In the following I use my duckid, rcasita.
- Assume that your current working directory is the source directory for project 0 (it does not matter what you call that directory)
- Create a subdirectory name rcasita in the current working directory by executing the following command in the shell:

  mkdir rcasita

- Create a text file name manifest with the following lines in it:

  rcasita/anagram.c

  rcasita/main.c

  rcasita/report.pdf or rcasita/report.txt

- Now execute the following lines in the shell:

  cp main.c anagram.c report.pdf rcasita

  tar -zcvf rcasita-project0.tgz $(cat manifest)

  tar -ztvf rcasita-project0.tgz

- The last command above should generate a listing that looks like the following:

```
-rw-r--r-- rcasita/<group> 3670  2015-03-30 16:30 rcasita/main.c
-rw-r--r-- rcasita/<group> 5125  2015-03-30 16:37 rcasita/anagram.c
-rw-r--r-- rcasita/<group> 629454        2015-03-30 16:30 rcasita/report.pdf
```

The use of my duckid above is simply to show the form of the commands and the type of output to expect when you list the archive. You must use your own duckid!

8 Marking Scheme for CIS 415 Project 0

Your submission will be marked on a 100 point scale. I place substantial emphasis upon WORKING submissions, and you will note that a large fraction of the points are reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles, links, and runs correctly. The information returned to you will indicate the number of points awarded for the submission. You must be sure that your code works correctly on the virtual machine under VirtualBox, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the virtual machine before submission. The marking scheme is as follows:

Points	Description
10	Your report - honestly describe the state of your submission
20	Anagram.c 6 for workable solution (looks like it should work) 2 if it successfully compiles 2 if it compiles with no warnings 6 if it works correctly (when test with an unsee dictionary) 4 if valgrind reports no memory errors
70	Main.c 18 for workable solution (looks like it should work) 2 if it successfully compiles 4 if it compiles with no warnings 2 if it successfully links 4 if it successfully links with no warnings 3 if it works correctly with A.dict 5 if it works correctly with A-B-C.dict 20 if it works correctly with A-Z.dict 12 if valgrind reports no memory errors

Several things should be noted about the marking schemes:

- Your report needs to be honest: Stating that everything compiles and it doesn't on Arch means you didn't compile it. This is the easiest 10 points you will earn if your honest.
- If your solution does not look workable, then the points associated with successful compilation and lack of compilation errors are not available to you. This prevents you from handing in a stub implementation for each of the methods and receiving points because they compile without errors, but do nothing.
- The points associated with workable solution are the maximum number of points that can be awarded. If I deem that only part of the solution looks workable, then you will be awarded a portion of the points in that category.
- Programs that segfault on all input will be docked for 100% of all working points.
- Programs that segfault on bad input will be docked for 50% of all working points.