Find Duplicate Entries in a Mailing List

Due at 11:59pm on Monday, 10 October 2016

1 Requirement

A mailing list is a collection of people's names and addresses, used for bulk mailings. Mailing lists from different sources are often merged. A merged mailing list might contain duplicate entries, with only trivial differences, such as alternative forms of names, missing parts of addresses, etc. Duplicate entries result in excessive postage costs, and irritation for recipients of multiple copies of a single mailing. A program is required to report potential duplicate entries in a given mailing list.

2 Specification

The program's function is to report potential duplicate entries in a given mailing list. Two entries are to be treated as potential duplicates if they have identical surnames, zip codes, and house numbers; these comparisons are made in a ***case-insensitive*** way.

A mailing list file is a text file in which each entry occupies three successive lines:

- full name
- street address
- city, ST zip-code

The full name consists of an individual's surname, followed by a comma (','), followed by an optional title, followed by one or more given names. The street address line consists of a house number and street name, or an organization name. The final line consists of the city, a separating comma (','), a 2-charater state abbreviation, white space, and a 5-digit zip code.

The program's output is a report on the standard output showing pairs of potentially duplicate entries. Both entries of a pair should be written out in full, **exactly as they appear in the mailing list file**. For example:

```
Potential duplicate
===================
Rogers, Fred
123 Neighbor Place
Latrobe, PA  15650
==========
Rogers, Mr
123 Neighbor Drive
Latrobe, PA  15650

Potential duplicate
===================
Sventek, Joe
Computer and Information Science Department
Eugene, OR  97403
==========
Sventek, Prof J
University of Oregon
Eugene, OR  97403
```

3 Design

I am providing you with the source file for main(), and header files for two abstract data types (ADT) – mentry.h and mlist.h.

3.1 mentry.h

```
#ifndef _MENTRY_H_
#define _MENTRY_H_

#include <stdio.h>

typedef struct mentry {
        char *surname;
        int house_number;
        char *zipcode;
        char *full_address;
} MEntry;

/* me_get returns the next file entry, or NULL if end of file*/
MEntry *me_get(FILE *fd);

/* me_hash computes a hash of the MEntry, mod size */
unsigned long me_hash(MEntry *me, unsigned long size);

/* me_print prints the full address on fd */
void me_print(MEntry *me, FILE *fd);

/* me_compare compares two mail entries, returning <0, 0, >0 if
 * me1<me2, me1==me2, me1>me2
 */
int me_compare(MEntry *me1, MEntry *me2);

/* me_destroy destroys the mail entry
 */
void me_destroy(MEntry *me);

#endif /* _MENTRY_H_ */
```

The `struct mentry`, and the corresponding typedef `MEntry`, define the data structure for a mailing list entry. For those members of the structure that are character pointers, you will have to use `malloc()` to allocate storage for these members, and store the pointers into the structure to return to the user. The entire 3-line address is stored as a single string at `full_address`; the `surname` and `zipcode` members are there to store single case versions of the surname and the zip code from line 3. The `house_number` member is to hold the integer house number at the beginning of the 2nd line of the address (0 if no number is supplied).

The constructor for this ADT is `me_get()`; it reads the next mailing list entry from `fd` and returns a pointer to an `MEntry` structure containing the mailing list entry; if there is an error, or you have

reached the end of file, NULL is returned. You will have to use malloc() to allocate the MEntry structure to return to the user.

me_hash() computes a hash value for the surname+zipcode+house_number, returning a value between 0 and size-1. The reference text, **The C Programming Language**, shows you how to build a simple hash table, and discusses hash functions for strings.

me_print() prints the full address on the specified file descriptor.

me_compare() compares two mail entries, returning <0, 0, >0 if me1<me2, me1==me2, me1>me2, respectively.

me_destroy() returns all heap-allocated storage associated with the mailing list entry.

3.2 mlist.h

```
#ifndef _MLIST_H_
#define _MLIST_H_

#include "mentry.h"

typedef struct mlist MList;

/* ml_create - created a new mailing list */
MList *ml_create(void);

/* ml_add - adds a new MEntry to the list;
 * returns 1 if successful, 0 if error (malloc)
 * returns 1 if it is a duplicate */
int ml_add(MList **ml, MEntry *me);

/* ml_lookup - looks for MEntry in the list, returns matching entry or NULL */
MEntry *ml_lookup(MList *ml, MEntry *me);

/* ml_destroy - destroy the mailing list */
void ml_destroy(MList *ml);

#endif /* _MLIST_H_ */
```

The typedef for MList is to hide the representation of a mailing list.

ml_create() is the constructor for this ADT, called with no arguments, and returns a pointer to an MList. It will return NULL if it is unsuccessful.

ml_add() adds an MEntry to the list, returning 1 if it is successful, 0 if not (due to malloc() failures); if you request that a duplicate to an existing entry be added, ml_add ignores the request, yet still returns 1.

ml_lookup() looks for an entry in the list that matches the 2nd argument; if found, it is returned as the function value; if not, NULL is returned.

ml_destroy() returns all heap-allocated storage associated with the entries in the list. The implementation must be sure to also return heap-allocated storage associated with the individual entries.

3.3 finddupl.c

```
#include <stdio.h>
#include "mentry.h"
#include "mlist.h"

static void usage(char *prog) {
        fprintf(stderr, "usage: %s [file]\n", prog);
}

int main(int argc, char *argv[]) {
        MEntry *mep, *meq;
        MList *ml;
        FILE *fd;

        if (argc > 2) {
                usage(argv[0]); return -1;
        }
        if (argc > 1) {
                fd = fopen(argv[1], "r");
                if (fd == NULL) {
                        fprintf(stderr, "Error opening %s\n", argv[1]);
                        return -1;
                }
        } else
                fd = stdin;
        ml = ml_create();
        while ((mep = me_get(fd)) != NULL) {
                meq = ml_lookup(ml, mep);
                if (meq == NULL)
                        (void) ml_add(&ml, mep);
                else {
                        printf("Potential duplicate\n");
                        printf("==================\n");
                        me_print(mep, stdout);
                        printf("==========\n");
                        me_print(meq, stdout);
                        printf("\n");
                }
        }
        ml_destroy(ml);
        if (fd != stdin)
                fclose(fd);
        return 0;
}
```

The main program is invoked as

```
./finddupl [filename]
```

If you do not specify the filename as an argument, then you must redirect standard input to that file – i.e., the following two invocations perform identically:

```
./finddupl S.txt
./finddupl <S.txt
```

The mainline functionality consists of the following pseudocode:

```
create a mailing list
while (get next entry is successful)
    lookup this entry
    if a potential duplicate is found
        print out the two potential duplicates
    else
        add this entry to the list
destroy the mailing list
```

4 Implementation

You are to implement mentry.c and mlist.c. The implementations must match the function prototypes in the headers listed in section 3 above. You may ***not*** modify the header files.

The implementation of mlist must be a hash table. The hash table must resize itself whenever any one hash bucket exceeds 20 entries. You should print diagnostics on stderr whenever you have to resize the table.

Note that you will be heavily penalized if your program leaks heap memory. After you have a working version of the program, you need to test it using "valgrind" to make sure it does not leak heap memory. If "valgrind" indicates ***any*** problems with your code's use of heap memory, it is usually an indication that you are doing something very wrong that will bite you eventually; you should 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 finddupl.c, mentry.h and mlist.h, I have also provided a linked list implementation of mlist.c in the archive. This will permit you to test your implementation of mentry.c against a working, albeit inefficient, implementation of MList.

5 Submission[1]

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 "mentry.c", your "mlist.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 mentry.h,

[1] ***A 20% penalty will be assessed if you do not follow these submission instructions. Section 6 describes how to follow these directions for those that are unsure.***

mlist.h, or finddupl.c. 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 marks.

These files should be contained in a folder named "<duckid>". Thus, if you upload "jsventek-project0.tgz", then we should see something like the following when we execute the following command:

```
$ tar –ztvf jsventek-project0.tgz
-rw-rw-r-- jsventek/None      3670 2015-03-30 16:30 jsventek/mentry.c
-rw-rw-r-- jsventek/None      5125 2015-03-30 16:37 jsventek/mlist.c
-rw-rw-r-- jsventek/None    629454 2015-03-30 16:30 jsventek/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 title of the assignment (CIS 415 Project 0)
- state either "This is my own work." or "This is my own work except that …", as appropriate.

We will be compiling your code and testing against an unseen set of addresses. We will also be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion.

6 How to create the correct archive to submit

- First, determine your duckid (your uoregon.edu email ***without*** the "@uoregon.edu". In the following, I refer to it as *duckid*.
- 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 named *duckid* in the current working directory by executing the following command in the shell:

```
$ mkdir duckid
```

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

```
duckid/mentry.c
duckid/mlist.c
duckid/report.pdf
```

(of course, if you are submitting `report.txt`, you should replace the last line with `duckid/report.txt`)

- Now execute the following lines in the shell:

```
$ cp mentry.c mlist.c report.pdf duckid
$# the previous command makes copies of the files in duckid
$ tar -zcvf duckid-project0.tgz $(cat manifest)
$ tar -ztvf duckid-project0.tgz
```

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

```
-rw-rw-r-- duckid/<group>   3670 2015-03-30 16:30 duckid/mentry.c
-rw-rw-r-- duckid/<group>   5125 2015-03-30 16:37 duckid/mlist.c
-rw-rw-r-- duckid/<group> 629454 2015-03-30 16:30 duckid/report.pdf
```

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.

As indicated in the handout, `mlist` must be implemented as a dynamic hash table, resizing itself whenever one of the hash buckets exceeds 20 entries.

The marking scheme is as follows:

Points	Description
10	Your report – honestly describes the state of your submission
20	MEntry ADT 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 tested with an unseen driver program) 4 if there are no memory leaks
70	MList ADT 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 with finddupl 4 if it links with no warnings 3 if it works correctly with S.txt and M.txt 5 if it works correctly with 10,000 entry unseen file 10 if it works correctly with 1,000,000 entry unseen file 10 if it works correctly with 10,000,000 entry unseen file 12 if there are no memory leaks

Several things should be noted about the marking schemes:

- Your report needs to be honest. Stating to me that everything works and then finding that it won't even compile offends me. The 10 points associated with the report are probably the easiest 10 points you will ever earn as long as you are 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 in each ADT 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. In particular, if your hash table does not resize as required, you will lose 10 of the 18 marks associated with this category.