CS 240: Programming in C
Spring 2019
Homework 10

Prof. Turkstra

Due April 3rd, 2019 9:00 PM

1 Goals

The purpose of this assignment is to reinforce the ideas of doubly-linked lists as well as introduce you to pointers to pointers. We will be adapting the previous homework to use pointers to pointers. We will also add some additional functionality.

2 The Big Idea

As in Homework 8, in this assignment, you will be using a database that holds final game scores for NCAA basketball games. This database contains dates, team names, and final scores for each game.

The structures follow:

```
typedef struct date_struct
  int month;
  int day;
  int year;
 date_t;

typedef struct game_info
  date_t date;
  char *home_team;
  int  home_score;
  char *away_team;
  int  away_score;
 game_info_t;

typedef struct double_list
  struct game_info *info;
  struct double_list *prev_ptr;
  struct double_list *next_ptr;
 double_list_t;
```

3 The Assignment

You are to write a series of functions that extract game information from a database using structures. The database is a NCAA final score game file that lists the date of the game, the two teams, and the score for each team (the highest scoring team wins).

In this homework, you are going to write even functions to:

- Create a new game node (a node contains data of a single game);
- Delete a single, separated game node;
- Insert the new game node into an internal database, a doubly-linked list, (in order);
- Delete an entire database (doubly-linked list);
- Find a game played within the database of games;
- Remove (isolate) a game from the database of games;
- Get a schedule of played games for a particular team (as a new doubly-linked list);
- Delete a list—just the list, not its contents.

Examine hw10.h carefully. It contains declarations for each type you will use. It also contains prototypes for the functions that you will write.

It is worth noting that each node of the list only has a *pointer* the game data structure. This permits multiple lists to point to (and access) the same data without duplicating it in memory. It also allows us to access the data without always first going through a doubly-linked list node.

For the purposes of this assignment, all doubly-linked lists will be NULL-terminated. I.e., the tail element of the list will have a NULL `next_ptr`, and the head of the list will have a NULL `prev_ptr`.

3.1 Functions You Will Write

You will write the following functions:

```
void create_game_node(date_t, char *, int, char *, int, double_list_t **);
```

This function should allocate and populate a new `double_list_t` node. Remember to set both the `next_ptr` and `prev_ptr` to NULL.

The parameters are ordered as follows: date, home team name, home team score, visitor team name, visitor score.

Do not forget to allocate the necessary memory for the internal pointers!

The function should modify the pointer pointed to by the last argument to point to the newly allocated list node.

You should assert that the first three date fields are valid (day should range from 1 to 31, month from 1 to 12, and year should be larger than, say, 0). You should assert that both string arguments are not NULL. You should assert that the scores are positive (>= 0). You should assert that the final argument is not NULL and that what it points to *is* NULL.

The ID field for this assignment is a little open-ended. We recommend that you use some form of calculation involving the actual date such that the ID is a single number that uniquely identifies the date. This allows you to compare a single field instead of three when determining the insertion order. The constants `MONTHS_IN_YEAR` and `DAYS_IN_MONTH` are provided in an effort to help you.

The ID field will not be examined by any portion of the test module and is entirely optional.

```
void delete_game_node(double_list_t **);
```

This function should deallocate a single doubly-linked list node and all of its associated data.

You should assert that the argument is non-NULL and that what it points to is not NULL. You should also assert that the node is isolated (i.e., not part of a list).

On completion, set the pointer pointed to by the first argument to NULL.

```
void insert_game(double_list_t **, double_list_t *);
```

This function should insert the singleton list node (second argument) into the database (doubly-linked list) of games pointed to by the value of the first argument.

Please note: if the list exists, the first argument is not guaranteed be a pointer to a pointer to the head!

Insert the node such that the list continues to be sorted by date. In the event of a tie, sort alphabetically by home team name.

The function should set the pointer pointed to by the first argument to point to the head, if it does not already.

Assert that both arguments are not NULL. It is not an error for the first argument to point to a pointer that is NULL.

It is recommended that you draw a number of pictures so that you can visualize how nodes will be inserted at various locations within the doubly-linked list. Be sure to handle all corner cases.

You may wish to again create your own `prepend()` and `append()` functions.

```
int delete_database(double_list_t **);
```

This function deallocates an entire doubly-linked list. The argument points to a pointer to somewhere inside of the list (not necessarily the head).

This function should deallocate all internal data for each node as well as every node in the list.

On completion, the pointer pointed to by the first argument should be set to NULL. The function should return the number of nodes deallocated.

We encourage you to traverse the list and use `delete_game_node()`.

Assert that the argument is non-NULL. It is not an error if the pointer it points to is NULL.

`int find_game(double_list_t **, date_t, char *);`

This function should search through the doubly-linked list pointed to by the value of the first argument (again, not necessarily the head) and identify the game that matches the date (second argument) and the first part of the team name (third argument). First check the home team name, then the visitor's.

If a match is found, this function should return the node "placement" number—the number of nodes traversed from the head of the list to the first matching node. The head node is considered position #1. The value pointed to by the first argument should be updated to hold the address of the found node.

Otherwise, the function should return `GAME_NOT_FOUND`.

Assert that the first and third arguments are non-NULL. It is not an error for the first argument to point to a NULL pointer.

`void remove_game(double_list_t *);`

This function removes the node pointed to by the first argument from the doubly-linked list in which it currently resides. Do not `free()` the node. Only remove it.

If the argument is NULL, do nothing.

Be sure to set the removed node's next and previous pointers to NULL.

`int get_schedule_list(double_list_t *, char *, double_list_t **);`

This function should search through the provided database (pointed to by the first argument—not necessarily the head) for all games played by the specified team (second argument).

This function should build a new list of `double_list_t`s *without duplicating the actual data payload of each node*. Put anther way, the `info` pointer for each node should point to already allocated data. The value at the address pointed to by the third argument should be set to the head of this list.

The team name must match exactly.

The function returns the number of games the specified team played (number of nodes in the new doubly-linked list). If the specified team did not play, the function should return 0 and set the pointer pointed to by the third argument to NULL.

Assert that all three arguments are not NULL. Assert that the pointer pointed to by the third argument is NULL.

`int delete_list(double_list_t **);`

This function performs the opposite of `get_schedule_list()`. Since `get_schedule_list()` generates a doubly-linked list that contains payloads that point to data in the database, `delete_list()` should delete and deallocate the entire list of nodes *without* deallocating the actual payloads.

This function differs from `delete_database()` in that the latter deallocates the nodes AND associated payloads while `delete_list()` only deallocates the node.

This function should return the number of nodes deallocated as well as set the pointer pointed to by the argument to NULL.

Assert that the argument is not NULL and what it points to is not NULL.

3.2 Input Files

We will take a break from input files on this homework.

3.3 Header Files

We provide a header file, hw10.h, for you. It contains prototypes for each of the functions that you will write as well as #definitions for the constants. You should not alter this file. We will replace it with the original when grading.

Carefully notice that the node of the double-linked list has only a single pointer to the data structure as a payload and not the actual contents as in homework 8. The purpose is so that multiple lists could point to (and access) the data structure without always accessing the original database (double-linked list).

3.4 Error Codes

The following error codes should be used by your functions:

- `GAME_NOT_FOUND` the given team (and date) does not exist in the database (doubly-linked list).

These Standard Rules Apply

- You may add any `#include`s you need to the top of your hw10.c file;
- You may not create any global variables other than those that are provided for you. Creation of additional global variables will impact your style grade;
- Do not look at anyone else's source code. Do not work with any other students.

Submission

To submit your program for grading, type:

```
$ make submit
```

In your hw10 directory. You can do this as often as you wish. We encourage you to submit your code as often as possible. Only your final submission will be graded.

4 Grading

The operation of your functions will be graded out of 100 points. The point breakdown will be determined by the test program.

The test program will be run many times when grading. It is your job to do the same. The lowest score will be your final grade.

This homework will also have a style grade based on 20 points, with 1 point deducted for each code standard violation found.

Your code must compile successfully using `-Wall -Werror -std=c99` to receive any credit. Code that does not compile will be assigned an automatic score of 0.

If your program crashes (e.g., segmentation fault) at any point during the testing process, your score will be reduced by 25 points.