CS 240: Programming in C
Fall 2020
Homework 7

Prof. Turkstra

Due Octobor 21, 2020 9:00 PM

1 Goals

The purpose of this homework is to provide insight into dynamic memory allocation, pointers to structures, and linked lists.

Note on `malloc()`

It is important to check the return value of `malloc()`. Make sure that you always `assert()` that the address returned by `malloc()` is not NULL.

Now that we are dealing with dynamic memory allocation, we have a special malloc debug library that you can use if you are having trouble. Simply replace:

```
#include <malloc.h>
```

with

```
#include <malloc_debug.h>
```

and recompile.

You should not include `stdlib.h`, but rather `malloc.h`

2 The Big Idea

In this homework, you will be using a singly linked list to store cards. The standard 52-card deck of playing cards includes 13 ranks in each of the four suits: Club, Diamond, Heart and Spade. Each suit includes an Ace, a King, Queen, Jack and ranks 2 through 10. The struct `card_node` described below will be used to record one card inside our deck.

```
typedef struct card_node {
   char *suit;
   char *rank;

   struct card_node *next_card;
} card_node_t;
```

Make sure you always remember to set `next_card` appropriately in your functions.

Examine "hw7.h" carefully. It contains a declaration for the `card_node_t` type and prototypes for all functions you will write.

3 The Assignment

3.1 Functions You Will Write

You will write the following functions:

`card_node_t *add_card_to_head(card_node_t*, char *, char *);`

The first argument is a pointer to a `card_node_t` and (if not NULL) will point to the head of the list. If the first argument is NULL, the function should assume that it is appending to a new (empty) list.

You need to create a `card_node_t` holding the suit value (the second input argument) and rank value (the third input argument), and append it to the header of the list and return a pointer to the head of the list.

This function returns a pointer to the head of the new list.

You should assert that the second and third arguments are not NULL.

`card_node_t *remove_card_from_head(card_node_t *);`

The input argument is a pointer to a `card_node_t` and (if not NULL) will point to the head of the list.

You need to remove the head of the given `card_node_t` and move the head of the list accordingly.

This function returns the new head of the list.

~~This function should return NULL if the list was empty or became empty.~~

Note, the memory of the removed node should be freed accordingly.

`card_node_t *add_card_to_tail(card_node_t *, char *, char *);`

The first argument is a pointer to a `card_node_t` and (if not NULL) will point to the head of the list. If the first argument is NULL, the function should assume that it is appending to a new (empty) list.

You need to create a `card_node_t` holding the suit value (the second input argument) and rank value (the third input argument), and append it to the tail of the list and return a pointer to the head of the list.

This function returns a pointer to the head of the new list.

You should assert that the second and third arguments are not NULL.

`card_node_t *remove_card_from_tail(card_node_t *);`

The input argument is a pointer to a `card_node_t` and (if not NULL) will point to the head of the list.

You need to remove the tail of the given `card_node_t`.

This function returns the new head of the list.

~~This function should return NULL if the list was empty or became empty.~~

Note, the memory of the removed node should be freed accordingly.

```
int count_cards(card_node_t *);
```

Traverse the list pointed to by the argument starting from the head and return the number of cards stored inside the list. ~~If the input argument is NULL, you should return 0.~~

```
card_node_t *search_by_index(card_node_t *, int);
```

Search the list pointed to by the first argument with the given index (the second argument) and return the corresponding `card_node_t`. ~~If the second argument is larger than the size of the list, you should return NULL instead.~~

~~You should assert that the second argument is larger than 0.~~

```
card_node_t *search_by_card(card_node_t *, char *, char *);
```

Search the list pointed to by the first argument for the given suit (the second input argument) and the rank (the third input argument) and return the `card_node_t` if found, otherwise, return NULL if not found.

You should traverse the list pointed to by the first argument starting from the head and return the first `card_node_t` matches the given data.

~~You should assert that the second and third arguments are not NULL.~~

```
card_node_t *modify_card_by_index(card_node_t *, int, char *, char *);
```

Search the list pointed to by the first argument with the given index (the second argument) and return the corresponding `card_node_t` with the new suit (the third input argument) and rank (the forth input argument). If the second argument is larger than the size of the list, you should return NULL instead.

~~You should assert that the second argument is larger than 0, and the third and forth arguments are not NULL.~~

```
card_node_t *move_to_tail(card_node_t *, int);
```

This function will take a pointer to a `card_node_t` as the first argument and an integer `N` as the second argument. It will move the first `N` `card_node_t`s of the list to the tail.

This function return a pointer to the head of the final list.

For example, the input lists `(1->2->3->4->5)` and `N = 3` would yield the output `(4->5->1->2->3)`.

~~You should assert that the second argument is greater and equal to 0 and less and equal to the size of the first input argument.~~

```
card_node_t *interleave_decks(card_node_t *, card_node_t *);
```

This function will take in two pointers to `card_node_t`. It should decide which list is larger or smaller and start to interleave based on the larger one. If both has the same size, then use the first input argument to start. Based on that, it should interleave the two lists into one list, and return the header of that one list.

For example, the input lists `(1->2->3->4->5->6->7)` and `(8->9->10)` would yield the output `(1->8->2->9->3->10->4->5->6->7)`.

~~This function *should not allocate any new memory.* It should reuse the existing `card_node_t`!~~

~~Assert that both arguments are not NULL.~~

```
void free_card_list(card_node_t *);
```

Traverse the list pointed to by the argument and free (deallocate) all of the `card_node_t`.

3.2 Input Files

There are no input files for this assignment.

3.3 Header Files

We provide a header file, hw7.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.

3.4 Error Codes

There are no error codes for this assignment.

These Standard Rules Apply

- You may add any `#include`s you need to the top of your hw7.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;
- You should check for any failures and return an appropriate value.
- Do not look at anyone elses source code. Do not work with any other students.

Submission

To submit your program for grading, type:

```
$ make submit
```

In your hw7 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 2 points 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 an automatic 0.

If your program exhibits any memory errors (e.g., corruption, double free) at any point during the testing process, your score will be an automatic 0.