Technische Universität München

Informatics I for MSE

MSE IN8011
Prof. Bjoern Menze, Giles Tetteh
Philipp Kaeß

WS 19/20
Homework 8

Due Date: 19.01.20, 23:59

Release Date: 10.01.20

Homework 8: Waiting List

Topics: Doubly Linked Lists, Pointers, Structs, Inserting, Project Creation

Introduction

As you might have already experienced, the registration for the informatic tutorials can be connected with difficulties. If one time slot is very popular, the number of students trying to register will be much larger than the maximum number of allowed students per tutorial. Even though this solution may not be perfect and sometimes lead to student uprisings, TUM-online makes use of the first-come-first-serve principle, which then eventually results (in the case of an already completely filled group) in a waiting list of students in the order of registration time.

MSE first semesters protesting against long waiting lists for the informatics tutorials (2019)

The idea of this waiting list is the following: If one regularly registered student changes the group or unregisters, the first person on the waiting list will get his/her place. The waiting list follows the principle FIFO (The student that **First** goes **In**to the list, is the **First** one to get **Out**), and can therefore be considered as a queue of students. To ensure that everything is fair and works properly, you have been hired by TUM to realize the waiting list in C (only the waiting list, not the complete registration process).

As you just heard of linked lists in last week's tutorial, you decide to use a doubly linked list to model the queue (explanation see Tool Box). You are given a header file, which acts as an interface between your file and the system (main-file) which will later call your functions. It contains the struct definitions and the declaration of the functions that are needed and will later be called by the system. They will be explained in detail below.

Specifications

The provided header file contains three `struct` definitions:

We use `struct Student` to store and process students. Therefore, the following `struct` members are defined:

- `char name[50]` , the full name of the student
- `int matr_nr` , the matriculation number of the student
- `int semester` , the current semester of the student.

We use `struct Node` to represent the nodes in our stack. Therefore, the following `struct` members are defined:

- `struct Student *s` , a pointer to the student of this node
- `struct Node *next` , a pointer to the next node in the stack
- `struct Node *previous` , a pointer to the previous node on the stack.

We use `struct WaitingList` to represent the waiting list. Therefore, the following `struct` members are defined:

- `struct Node *head` , a pointer to the head node of the stack
- `struct Node *tail` , a pointer to the tail node of the stack.

The functions you need to code are explained in the following scenario:

When the registration process starts and the need of a waiting list is detected, the system first calls the function `new_empty_waiting_list()`, which has no input parameters and returns a pointer to a newly allocated and initialized element of type `struct WaitingList`. This list will then later be filled with students. To signalize that the list is empty, both `head` and `tail` have to point to NULL in the initialized state.

To add a student to the waiting list, the function will call `add_student(struct WaitingList *list, char name[50], int matr_nr, int semester)`, where the input parameters represent the list the student should be inserted into, and the stats of the student. The function should then create a new student, create a new node referencing that student, and insert that node at the bottom end (tail) of the list. We highly encourage you to create your own additional functions for creating the student and creating the node, which you can then simply call in your insert function.

In the case of an unregistration for the tutorial group, the system fills the free place with the first student in the waiting list, and therefore calls the function `pop_student(struct WaitingList *list)`, where the input parameter represents the list the student should be popped from. This function should then return a pointer to the `struct Student` that is referenced in the first node (head) and remove this node from the list. As only a pointer to the student, and not the student itself is stored in this node, the function should then free the memory of the node (not the student !). The student will further be processed by the system. **If the waiting list is already empty and no student can be popped, the function should return a NULL-pointer.**

In addition to these basic functions, a few more functionalities are required:

The function `get_list_length(struct WaitingList *list)` should return the current number of students in the given waiting list.

The function `is_waiting(struct WaitingList *list, int matr_nr)` should return `1`, if a student with the given matriculation number is currently waiting in the waiting list, and `0` if not.

The director of the TUM also commands you to implement the function `add_vip(struct WaitingList *list, char name[50], int matr_nr, int semester)`. With the help of this function, he intends to insert his favorite students (Children of professors or main sponsors and former tutors in Informatics 1 for MSE) not at the end, but at the very top of the waiting list (head). Even though you express moral concerns, you have no choice but to implement also this function.

The function `clear_list(struct WaitingList *list)` should remove all students from the list and delete them. (Remember to free the memory of the nodes and the students, **but not the struct WaitingList itself**!) After the run of this function, `head` and `tail` of `list` should point to NULL again.

Tasks

We will not provide a template C-File this time, as we want you to learn to start a project from scratch. You will have to create a file called ***waiting_list.c*** with all above defined functions in it. You may of course also define own, additional functions in that file. Please note:

- Do not print anything
- You are provided with these files:
  - ***main_list.c*** is the main file we provide you to test your functions. Do not change anything in this file. This file will test your functions as described below (Example Output)
  - ***waiting_list.h*** is the header to the template file ***waiting_list.c***. It has to be included in ***waiting_list.c*** as well as in ***main_list.c***. It contains the above mentioned `struct` definitions and function declarations.
- To run your functions, compile the two C-Files together (`gcc -Wall -Werror main_list.c waiting_list.c`).
- When you are finished, put ***waiting_list.c*** directly in a zip-File named "HW8_Mustermann_Max" (with your name of course) and upload this file to Moodle. Do not upload ***waiting_list.h*** or ***main_list.c***, and do not put any additional folders into the zip-File.

Example Output

The main will first create a new empty list with your function and check if the initialization is correct. It will then repeatedly call your functions to add and pop students to/from that list, and compare the return values of the pop-calls to the expected outputs. Therefore, this main will not check the structure / correctness of the list itself, but only its functionality.

To test the other functions, a new list is created, and various (VIP and "normal") students are added and popped, and the functions `is_waiting()`, `get_list_length()` and `clear_list()` are called multiple times. Once again, the output is compared to the expected one.

If an output does not match the solution, an error will be printed. Some tests are worth two or more cases, this will be printed as well. To make it easier for you to understand where and why an error appears, the single steps of the main (adding, popping, calling functions etc.) are printed.

Please be aware of the fact that your functions highly depend on each other. If the initialization of the list is not correct, the rest of your functions won't work as well, even though they may be correct. Similar, the call of your pop-function may return an error even though the function itself may be correct, because something went wrong at the adding process. Simply solve the errors in the order of appearance, and everything should work fine. If the program crashes during runtime, or gets stuck in an infinite loop, it is likely that the structure of the list is not correct. (e.g. `next` pointer of `tail` / `previous` pointer of `head` do not point to NULL, interconnections between nodes are broken etc.)

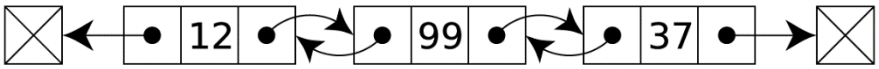
After you compiled and executed the main together with your functions, your terminal should look like this:

```
Testing new_empty_waiting_list (3 cases)...

Testing add_student and pop_student (9 cases)...
        Popping, checking output...
        Adding student to list...
        Popping, checking output...
        Popping, checking output...
        Adding student to list...
        Adding student to list...
        Popping, checking output...
        Adding student to list...
        Adding student to list...
        Popping, checking output...
        Popping, checking output...
        Adding student to list...
        Popping, checking output...
        Popping, checking output...
        Popping, checking output...

Testing add_vip (6 cases), get_list_length (4 cases), is_waiting (5 cases) and clear_list (3 cases)...
        Adding normal student to list...
        Adding vip-student to list...
        Adding normal student to list...
        Popping, checking output (2 cases)...
        Adding vip-student to list...
        Adding normal student to list...
        Calling get_list_length()...
        Calling is_waiting() (2 cases)...
        Calling is_waiting()...
        Popping, checking output (2 cases)...
        Calling clear_list()...
        Calling get_list_length() (2 cases)...
        Calling is_waiting()...
        Adding vip-student to list...
        Calling get_list_length()...
        Calling is_waiting()...
        Popping, checking output (2 cases)...

Passed 30/30 cases
```

Tool Box
The explanations below **may** help you to solve this problem:

DOUBLY LINKED LIST

In the tutorials you already got to know the singly linked list. As you may have noticed, it can be quite time-expensive to iterate to the end of a singly linked list (e.g. to insert a node there). A solution to that can be to "mirror" the pointer connections and introduce a `previous` pointer in addition to the `next` pointer, and a `tail` pointer in addition to the `head` pointer. This enables us to iterate through the list in both directions. We call that approach a **doubly linked list**. Once again we realize the omnipresent trade-off between memory and runtime: The gain in speed when using a doubly linked list has to be paid by an increase of memory usage needed to realize the list.
An example of a doubly linked list to store integers is shown below:



Just as in the tutorial the arrows represent pointers and the crossed squares represent NULL-Pointers. In addition to the `head` pointer (in this case a pointer to the node containing 12) we introduce a `tail` pointer (i.t.c. to the node containing 37), which are both not shown in the above visualization.
We can insert and delete nodes in the doubly linked list similarly as we did it in the singly linked one, but we have to take care of both the `next` and the `previous` pointers now.

The value of the nodes in this exercise is no longer an integer, but a pointer to a struct. This is not really different though, you will only have to get used to the fact that the students are not stored inside the node, but only referenced there. This means that if you create a node, you won't create a student automatically, only enough space to store the address of a student. You will have to create a student manually, and store the reference to that student in the node. The same holds for freeing: Freeing a node will not automatically free the student, only the reference stored in the node.

CORNER CASES

The difficulty of inserting and popping is the fact that dependent on the current number of elements in the list, the procedure differs. There are different steps necessary to insert the very first element into an empty list, than to insert an element into an already filled one. Think of all these corner cases and catch them for special treatment. Otherwise you will encounter errors like e.g. dereferencings of NULL-Pointers, which will lead to runtime crashes. In general, if your program crashes during runtime it is likely that you dereferenced an invalid address or accessed unreserved memory.

Threshold
To pass this homework, your program has to fulfill the following conditions:

- Your file ***waiting_list.c*** must be compilable and runnable together with the main file without any errors and with the flags -Wall -Werror enabled
  (`gcc -Wall -Werror -o test main_list.c waiting_list.c`)

Your functions will be tested on a total of 30 cases very similar to the ones in the provided main file. (3 cases for `new_empty_waiting_list()`, 9 cases for adding and popping, 6 cases for `add_vip()`, 4 cases for `get_list_length()`, 5 cases for `is_waiting()` and 3 cases for `clear_list()`) To pass this assignment, you have to pass at least 80% (=24) of the test cases.

Information on Plagiarism and Copyright

As already announced in the lecture and the tutorials, you are supposed to do this homework alone. This homework is an official part of the exam at the end of this semester, and therefore fraud or plagiarism will be pursued and punished.

You are encouraged to discuss **specific** problems with other students (like e.g. “I don’t know how to print a string, can you help me?”), but you are forbidden to share ideas on the approach/structure of your functions, and of course to show other students your solution. This will result in a similar control-flow and structure of your submissions, which we will be able to detect even if you change variable names or loops etc.

We will compare your submissions with a code-optimized plagiarism scanner and if we find plagiarized code, you will fail this submission. If you are found guilty of plagiarism twice, you will lose the homework bonus, and in severe cases we will take further steps.

This homework is intellectual property of the TUM, namely the Chair of Image-Based Biomedical Computing. Publishing either this homework task itself or your solutions to it in any form (including on the MSE-Cloud) is strictly forbidden and can be pursued TUM-internally as well as legally.