Assignment 2: Stacks and Queues
Due at 11:59 pm on Sunday, 2/11/2024
Demo due by 11:59 pm on Friday 2/23/2024

This assignment is intended to have you start working with stacks and queues and to start building ADTs on top of other data structures. There are a couple parts to the assignment, each described below.

Part 0. Download the skeleton code and unzip

You may download the skeleton code for this assignment using the wget command:
wget https://classes.engr.oregonstate.edu/eecs/winter2024/cs261-020/assignments/assignment2.zip

To unzip the file, use the following command:
`unzip assignment2.zip`

It's important that you don't modify the function prototypes specified in `queue.h` and `stack.h`. To help grade your assignment, we will use a set of tests that assume these functions exist and have the same prototypes that are defined in those files. If you change the prototypes, it will cause the tests to break, and your grade for the assignment will likely suffer.

Feel free to add any additional functions you might need to `queue.c` and `stack.c`. In addition, you may modify the dynamic array implementation provided in `dynarray.h` and `dynarray.c` or the linked list implementation provided in `list.h` and `list.c` as needed to help implement the queue and stack.

Part 1. Implement a stack

In this assignment, you'll implement two new ADTs on top of the data structures you implemented in the previous assignment. The first ADT you'll implement for this assignment is a stack.

For this assignment, the interface for the stack is already defined for you in the file `stack.h`. Your first task in this assignment is to implement definitions for the functions that comprise this interface in `stack.c`.

The stack functions you'll need to implement are outlined briefly below. All of these functions use a type called `struct stack`, which is defined in `stack.c` and represents the stack itself. For more details, including information on function parameters and expected return values, see the documentation provided in `stack.c`.

- `stack_create()` – This function should allocate, initialize, and return a pointer to a new stack structure.

- `stack_free()` – This function should free the memory held within a stack structure created by `stack_create()`. Note that this function only needs to free the memory held by the stack itself. It does not need to free the individual elements stored in the stack. This is the responsibility of the calling function.

- `stack_isempty()` – This function should return 1 if the stack is empty and 0 otherwise.

- `stack_push()` – This function should insert a new element on top of the stack. **This operation must have O(1) average runtime complexity.**

- `stack_top()` – This function should return the value stored at the top of the stack without removing it. **This operation must have O(1) average runtime complexity.**

- `stack_pop()` – This function should pop a value from the stack and return the popped value. **This operation must have O(1) average runtime complexity**.

Importantly, the stack you build **MUST** use a linked list as its underlying data storage. You are provided with a linked list implementation in `list.h` and `list.c` that you may use for this purpose. Feel free to modify this linked list implementation as needed to implement your stack, **with the constraint that your stack may only interact with the linked list implementation via its interface functions**. In particular, you may not directly access or modify the fields of the linked list structure (`struct list`) from your stack. In other words, you may not change the fact that `list.h` only contains a forward declaration of `struct list`, and you may not redefine the list structure in `stack.h` or `stack.c`.

Also, note that, as with the data structures you implemented in assignment 1, values in the stack will be stored as void pointers.

Part 2. Implement a queue
The second ADT you'll implement for this assignment is a queue.

For this assignment, the interface for the queue is already defined for you in the file `queue.h`. Your second task in this assignment is to implement definitions for the functions that comprise this interface in `queue.c`.

The queue functions you'll need to implement are outlined briefly below. All of these functions use a type called `struct queue`, which is defined in `queue.c` and represents the queue itself. For more details, including information on function parameters and expected return values, see the documentation provided in `queue.c`.

- `queue_create()` – This function should allocate, initialize, and return a pointer to a new queue structure.

- `queue_free()` – This function should free the memory held within a queue structure created by `queue_create()`. Note that this function only needs to free the memory held by the queue itself. It does not need to free the individual elements stored in the queue. This is the responsibility of the calling function.

- `queue_isempty()` – This function should return 1 if the queue is empty and 0 otherwise.

- `queue_enqueue()` – This function should insert a new element at the back of the queue. **This operation must have O(1) average runtime complexity**.

- `queue_front()` – This function should return the value stored at the front of the queue without removing it. **This operation must have O(1) average runtime complexity**.

- `queue_dequeue()` – This function should dequeue a value from the queue and return the dequeued value. **This operation must have O(1) average runtime complexity.**

Importantly, the queue you build **MUST** use a dynamic array as its underlying data storage. You are provided with a dynamic array implementation in `dynarray.h` and `dynarray.c` that you may use for this purpose. Feel free to modify this dynamic array implementation as needed to implement your queue, **with the constraint that your queue may only interact with the dynamic array implementation via its interface functions**. In particular, you may not directly access or modify the fields of the dynamic array structure (`struct dynarray`) from your queue. In other words, you may not change the fact that `dynarray.h` only contains a forward declaration of `struct dynarray`, and you may not redefine the dynamic array structure in `queue.h` or `queue.c`.

Also, note that, as with the data structures you implemented in assignment 1, values in the queue will be stored as void pointers.

<u>Part 3. Application: Implement a basic call center using a stack and queue:</u>
Finally, in `callcenter.c`, you should create a program that simulates a call center. The program should use a queue to keep track of the calls that are waiting to be answered, and a stack to keep track of the calls that have been answered. **No error handling needed for this part!** Here are some things you should bear in mind as you're implementing your application:

- Your call center implementation must use **BOTH** your stack and your queue.
- At the beginning, the stack and queue should be empty.
- Each call needs to be stored using a struct that consists of an integer ID (starts from 1), a caller's name, and a call reason. (You may assume both the name and the call reason are one word/string.)
- In each iteration, the user will choose one from the following options:

    - Receive a new call – take user inputs of caller’s name and call reason to construct a call struct item. Then add it to the queue.
    - Answer a call – remove the first element from the queue and add it to the stack of answered calls. If the queue is empty (i.e., no call to answer), a special message should be displayed.
    - Display the current state of the stack – print out the number of calls answered, and the details of the last call answered (ID, caller’s name and call reason).
    - Display the current state of the queue – print out the number of calls to be answered, and the details of the first call to be answered (ID, caller’s name and call reason).
    - Quit.
- Remember, the stack and queue data structures you implement will store data as void pointers, so you'll need to figure out how to store call structs from the input as void pointers in these data structures.
- Your program should be properly decomposed into tasks and subtasks using functions, including main(). To help you with this, use the following:
    - Make each function do one thing and one thing only.
    - No more than 15 lines inside the curly braces of any function, including main(). Whitespace, variable declarations, print statements, single curly braces, vertical spacing, comments, and function headers do not count.
    - Functions over 15 lines need justification in comments.
    - Do not put multiple statements into one line.
- Make sure your application doesn't have any memory leaks!

Example output: The user inputs are highlighted.

```
1. Receive a new call
2. Answer a call
3. Current state of the stack – answered calls
4. Current state of the queue – calls to be answered
5. Quit
Choose an option: 1

Enter caller’s name: Roger
Enter call reason: Register_for_data_structure_class_at_OSU

The call has been successfully added to the queue!

1. Receive a new call
2. Answer a call
3. Current state of the stack – answered calls
4. Current state of the queue – calls to be answered
5. Quit
Choose an option: 2
```

```
The following call has been answered and added to the stack!

Call ID: 1
Caller’s name: Roger
Call reason: Register_for_data_structure_class_at_OSU

1. Receive a new call
2. Answer a call
3. Current state of the stack – answered calls
4. Current state of the queue – calls to be answered
5. Quit
Choose an option: 2

No more calls need to be answered at the moment!

1. Receive a new call
2. Answer a call
3. Current state of the stack – answered calls
4. Current state of the queue – calls to be answered
5. Quit
Choose an option: 3

Number of calls answered: 1
Details of the last call answered:

Call ID: 1
Caller’s name: Roger
Call reason: Register_for_data_structure_class_at_OSU

1. Receive a new call
2. Answer a call
3. Current state of the stack – answered calls
4. Current state of the queue – calls to be answered
5. Quit
Choose an option: 4

Number of calls to be answered: 0

1. Receive a new call
2. Answer a call
3. Current state of the stack – answered calls
4. Current state of the queue – calls to be answered
5. Quit
Choose an option: 5

Bye!
```

Extra credit: use two stacks to implement a queue

For up to 10 points of extra credit, you can implement a data structure that uses two instances of your stack data structure to implement a queue. In other words, you should implement a queue that uses two stacks to form the underlying container in which data is stored (instead of, for example, a dynamic array or a linked list). For example, when the user calls `enqueue()` on your `queue-from-stacks` data structure, it will add the newly-enqueued element into one of the two stacks, as appropriate, and when the user calls `dequeue()`, your `queue-from-stacks` will remove the appropriate element from one of the two stacks. To the user of your queue-from-stacks, it will behave just like a normal queue. (This is yet another "job interview" kind of problem!)

***Hint: to implement a queue using two stacks, it might help to think of one stack as an "inbox" and one stack as an "outbox".**

The interface of your queue-from-stacks is defined in `queue_from_stacks.h`, and you must complete each of the functions implementing the queue-from-stacks in `queue_from_stacks.c`. Each of the functions in `queue_from_stacks.c` has a function header comment that describes more precisely how it should behave.

To be able to earn this extra credit, your stack implementation above must be working correctly, and importantly, you may only use the functions from your stack implementation prototyped in `stack.h` to interface with your two stacks. You may not access the underlying stack data directly. Also, make sure your queue-from-stacks implementation does not have any memory leaks! Note that there are no runtime complexity requirements for the queue-from-stacks operations. In other words, you can still earn the full extra credit even if your enqueue and/or dequeue operation is O(n).

To test your queue-from-stacks implementation, a testing application `test_queue_from_stacks.c` is provided. This application will be compiled automatically using the provided `makefile`. You can run it like so: `./test_queue_from_stacks`.

Testing your work

In addition to the skeleton code provided here, you are also provided with some application code in `test_stack.c` and `test_queue.c` to help verify that your stack and queue implementations, respectively, are behaving the way you want them to. In particular, the testing code calls the functions from `stack.c` and `queue.c`, passing them appropriate arguments, and then prints the results. You can use the provided `Makefile` to compile all of the code in the project together, and then you can run the testing code as follows:

```
make
./test_stack
./test_queue
```

Example output of these two testing programs using correct implementations of the stack and queue is provided in the `example_output`/ directory.

In order to verify that your memory freeing functions work correctly, it will be helpful to run the testing applications through `valgrind`.

Submission

In order to submit your homework assignment, you must create a **zip file** that contains `assignment2/` folder with **your implementation, and all provided files (.c, .h, Makefile)**. This zip file will be submitted to TEACH . In order to create the zip file, go to the directory where you can access the `assignment1/`, and use the following command:

```
	zip assignment2.zip assignment2 -r
```

Remember to sign up with a TA to demo your assignment. The deadline of demoing this assignment without penalties is 2/25/2024.