Exercise 02: Queue ADT and Stack ADT (PreLab)

Previously, we have implemented the List ADT using singly-linked lists. Now, this implementation will be used to create two new ADTs - queue ADT and stack ADT. Queues and stacks are similar to lists; all of these have a linear structure for organizing its data items. But how do they differ?

Queue ADT

A queue is a linear ADT that follows the *First In, First Out* (a.k.a. *FIFO*) structure. The FIFO rule simply restricts the insertion of new data at one end (usually, front or head) of the list and the deletion at the other end (usually, back or tail) of the list. An example of a queue in action would be an *Open Tambayan*, wherein an organization serves free food for everyone who fall in line in front of their booth. The individual in front of the line will be served first, followed by the next one in line, and so on. Newcomers should come to the back of the line and wait for their turn to be served. *First come, first served basis*. (Don't worry, in Queue ADT is strict; *bawal ang singit*.)

Stack ADT

Similar to a queue, stack is linear; however, it has a *Last In, First Out* (a.k.a. *LIFO*) structure. Equivalently, it has *First In, Last Out* structure. In stacks, both insertion and deletion are restricted on one end of the list. For example, you bought a cone of ice cream, the vendor would start putting scoops of ice cream, one over the other, until it's enough. When you eat it, you can only take on the top part. If you requested a refill, the vendor will add another scoop of ice cream on top of your cone. This is how a stack works: insertion and deletion happen only on one end, at what is called the *top of stack*.

Implementation of Queue ADT and Stack ADT

The implementation of Queue ADT and Stack ADT would be very similar. For the simplicity of discussion, we will first describe the implementation of Queue ADT (see the provided files: `queue.h` and `queue.c`).

Queue ADT

As described above, we know that a queue is simply a list with restricted insert and delete functions. Thus, we can simply create and define the `QUEUE` ADT as another `LIST` ADT. This is done in `queue.h` using the following line of code:

```
typedef LIST QUEUE;
```

Technically, this method just creates an alias for `LIST`; a different name, but the same C structure. This saves us a lot of effort to rewrite another structure for `QUEUE` ADT. Also, all functions which use `LIST` ADT can also be used for `QUEUE` ADT.

For example, a function to create a `QUEUE` is needed. To implement this, `createList()` is just reused, as shown below:

```
QUEUE* createQueue(){
    return createList();
}
```

This is also shown in `queue.c`, implementation file.

Queue Functions

It is mentioned above that queues have two functions: *append at the tail* and *delete at the head* of the list. For simplicity, we call the first operation as *enqueue* and the latter as *dequeue*.

ADT	insert	delete
Queue	enqueue	dequeue
List	appendTail	deleteHead

The two queue functions can be implemented in the same way `createQueue(...)` is implemented (even if there are no explicitly written functions for inserting at tail and deleting at head in `list.h`).

Stack ADT

`STACK` ADT can be defined the same way as `QUEUE` ADT was defined above, using `typedef`. `STACK` is also `LIST`, functions for `LIST` can be reused to implement constructor and other operations for `STACK`.

Stack functions

Stacks insert and delete on one end of the list only. Here, we set the top of stack as the front of the list. Thus, the two operations for stack are insert at head and delete at head, which will be called as *push* and *pop* operations, respectively.

ADT	insert	delete
Stack	push	pop
List	insertHead	deleteHead

Tasks for the Pre-Lab of Exer02

To fully implement the Queue ADT and Stack ADT, do the following:

Part 1: Queue ADT

1. Put your **fully working** `list.h` and `list.c` (from Exer01 In-Lab exercise) in the same directory of this file. `list.h` and `list.c` from Exer01 In-Lab exercise are the prerequisites for the implementation of queue and stack ADTs.
2. `queue.h` and `queue.c` are already provided, no need to modify them. In `queue.c`, `LIST` functions `createLIST`, `appendTail` and `deleteHead` are called from `list.h` and `list.c` to implement `createQueue`, `enqueue`, and `delete` functions, respectively. Reusing the `LIST ADT` functions to implement `QUEUE ADT` functions can minimize effort, without sacrificing correctness.
3. Test your implementation by running `make queue`. It will compile and run `main_queue.c` and use the shell program `program_queue.cs` as input. The expected output is stored in `expected_queue.txt`.
4. Keep testing until you find no more errors or bugs in your code.

Part 2: Stack ADT

1. You are going to use the same **fully working** `list.h` and `list.c` from Exer01 In-Lab exercise.
2. In this part, you will write `stack.h` (and a corresponding `stack.c`) from scratch to implement the `STACK ADT` in the same way `QUEUE ADT` was created. You may start by copying `queue.h` and `queue.c` to `stack.h` and `stack.c`, respectively
3. Modify the *header guard* in `stack.h`. A header guard is a C macro that is used to prevent header files from being included by the preprocessor multiple times. A header guard generally looks likes the following:

```
#ifndef some_unique_token
#define some_unique_token

// ... contents of the header file

#endif
```

Header guards prevent the re-inclusion of a header file by defining some unique token once it is included in a program. A simple method of creating this token is by using the filename - the filename, including the file extension, is converted to uppercase and decorated with underscores.

For `stack.h`, use `_STACK_H_` as the unique token.

4. Include `stack.h` in `stack.c`
5. Define `STACK` using `typedef`.
6. Create and implement the following functions for `STACK` ADT.
   - `STACK* createStack(int);`

   - `void push(STACK*, NODE*);` - stack's insert operation
   - `int pop(STACK*);` - stack's delete operation

   Remember, push and pop functions can simply call and reuse the `LIST` functions `insertHead` and `deleteHead`, respectively.
7. Test your code by running `make stack` (similar to `make queue` that you run to test your queue implementation. To be able do this, first, you need to create `main_stack.c` (similar to `main_queue.c`),an interpreter for a shell program that modifies a stack and you can still use `program.cs` (the input file for queue) as the input file for this interpreter. Correspondingly, modify the `Makefile` to have a command `make stack` which will compile, link, and run the test for stack. Again, it is similar to the command `make queue` that you used for queue.
8. Be sure to document your code.

Notes on Files in this Directory

Each file in this directory are described below:

- `Exer02(PreLab).pdf` is this file; the first file you must open and read, since this contains the description and guide for everything.
- `queue.h` is a C-header file which contains important definitions for our `QUEUE` ADT
- `queue.c` is a C file which will contain all implementations of all functions declared in `queue.h`. We will make this a standard practice - we will give you a header (e.g. `file.h`) file and you will implement all functions in a corresponding implementation file (e.g. `file.c`).
- `main_queue.c` is a simple interpreter for a simple shell program that interacts with the `QUEUE` ADT. Using a shell program is easier for testing our ADTs.
- `program.cs` is the shell program for both queue and stack. The common commands + for insertion, − for deletion, and other commands are described in the succeeding section.
- `expected_queue.txt` is the expected output result when `program.cs` is run using `make` or `make queue`.
- `expected_stack.txt` is the expected output result when `program.cs` is run using `make stack`.
- `Makefile` is a configuration file for the `make` utility to ease our *code, compile, link, execute* cycle. If you want to learn about the `make` utility, go here. Usage of `make` for this `Makefile` is described in the succeeding section.

Shell Commands

A simple shell program can be created to interact with the `LIST` ADT. The available commands are described below:

- ‘+ `vwill insert the valuevas the new node using eitherenqueueorpush, depending on the ADT that is being tested;v`‘ must be a valid integer.
- `-` will delete the value of the `head` of the list.
- `E` will report if the ADT is empty or not.
- `p` will report the contents of the list.
- `Q` will terminate the program.

Invalid commands will be reported in the `stdout`

`make` commands

The following `make` commands are available:

- `make` default action is `make queue`
- `make clean` will delete all object files created by the `make` command
- `make queue` will compile and build the program for queue; then the shell program `program.cs` for both queue and stack is executed.
- `make build_queue` will create the executable file `queue`
- `make compile_queue` will compile `main_queue.c`, `queue.c` and `list.c`.

You can copy and modify the commands for queue to *make commands* for stack. + `make stack` should compile and build the program for stack; then the shell program `program.cs` for both queue and stack is executed.

Submission

Submit to our Lab Google Classroom a `.zip` file named Ex02PreLab.zip (e.g.U1-1LDelaCruzEx02PreLab.zip) containing the following:

1. `list.h`
2. `list.c`
3. `queue.h`
4. `queue.c`
5. `main_queue.c`
6. `program.cs`
7. `Makefile`
8. `stack.h`
9. `stack.c`
10. `main_stack.c`

NOTE: If you have working `list.h` and `list.c` from your Exer01 Pre-Lab exercise, you may finish the both tasks above (Part 1 and Part 2) within 30 minutes.

Questions?

If you have any questions, contact your lab instructor.