Lab 02: Queue ADT and Stack ADT

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. New comers 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 for 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 only fully describe implementation of Queue ADT here.

Queue ADT

As described above, we know that a queue is simply a list with restricted insert and delete operations. 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 uses `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(int maxLength){
    return createList(maxLength);
}
```

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

Queue Operations

It is mentioned above that queues have two operations:*insert at the back* and *delete at front* of the list. For simplicity, we call the first operation as *enqueue* and the latter as *dequeue*.

ADT	insert op	delete op
Queue	enqueue	dequeue
List	Insert at Tail	Delete at Head

The two queue operations 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 Operations

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 op	delete op
Stack	push	pop
List	Insert at Head	Delete at Head

Tasks

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

Part 1: Queue ADT

1. Put your **fully working** `list.h` in the same directory of this file *i.e* `list.h` and `list.c` are prerequisites for this implementation of queue and stack ADTs.

2. Complete the implementation of `QUEUE` ADT in `queue.c` by defining each function in `queue.h`. Remember, functions in `list.h` can be reused for this; to minimize effort, without sacrificing correctness.
3. Test your implementation by compiling and running `main1.c` and using the shell program `program1.cs` as input. Expected output is stored in `expected1.out`. (This is also done by running the command `make`.)
4. Keep testing until you find no more errors or bugs in your code. Of course, fix errors or bugs that you encounter.

Part 2: Stack ADT

On this part, you will write `stack.h` (and a corresponding `stack.c`) from scratch to create `STACK` ADT in the same way `QUEUE` ADT was created.

1. Make sure you have a **fully working** `list.h` and `list.c`.
2. Create a file named `stack.h` in the same directory of this file.
3. Begin by putting the *header guard*. 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. Define `STACK` using `typedef`.
5. 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, these functions can be implemented by reusing `LIST` ADT's operations.
6. (Optional) Create a simple program for testing your stack implementation; you may create an interpreter for a shell program that modifies a stack (similar to `main1.c`) and input file for this interpreter (*i.e* the shell program; similar to `program1.cs`). Use your test program to debug your implementation. See file name format below.

Be sure to document your code.

Submission

Submit a `.zip` file named `SurnamePrelab02.zip` containing the following files in our Google Classroom:

1. `list.h`
2. `list.c`
3. `queue.h`
4. `queue.c`
5. `main1.c`
6. `program1.cs`
7. `Makefile`
8. `stack.h`
9. `stack.c`

If you created a test program for `STACK` ADT, include the following:

1. `main2.c` - the interpreter program for stack
2. `program2.cs` - the shell program for stack
3. Modify the `Makefile` to have a command `make stack` which will compile, link, and run the test for stack.

Finish the tasks above (Part 1 and Part 2) within 30 minutes.

Questions?

If you have any questions, approach your lab instructor.