Mechanics of Programming CSCI243 Abstract Data Types Homework 6

(Updated 04/11/2017)

Due Date

08/01/2016 at 11:59pm

Introduction

For this homework you will build a simple Abstract Data Type (ADT) implementation of a queue module. Your queue must be able to hold any number of 64-bit items simultaneously, and must not be restricted to one specific 64-bit type (i.e., being able only to enqueue long integers is not acceptable).

Abstract Data Types

An ADT is a data type whose internal representation is not known to the application program using the type. The only information about the type that is available to the program comes from the ADT's *specification*, which can be thought of as the set of "rules" for using the type.

Commonly, the specification comes in the form of a header file that defines a type representation along with a set of function prototypes. From the application program's point of view, the indicated functions are the only way to manipulate an instance of the data type.

In an object-oriented language, a class declaration is an easy way to create an ADT. The class typically contains one or more private data members and one or more public member functions. The data members are *encapsulated*; that is, inaccessible from outside of the class; the only way to manipulate them is by invoking the predefined member functions.

Building an ADT in a non-OO language is a bit more challenging than defining a class, especially when the developer wants to ensure that the information held by an instance of the type is as opaque (non-visible) to the application as possible.

A specification for a queue type named `QueueADT` is provided for you. The type itself is opaque; to the application program, it appears to be a pointer to an anonymous `struct`, which prevents the application from attempting to access any data members within the structure. You will develop the implementation of the ADT.

Assignment

Queue Specification

The interface to the queue ADT is provided by the following functions and their behavior:

- `QueueADT que_create( int (*cmp)(const void*a,const void*b) );`
  Create a queue using the specified comparison function, which may be `NULL`.
- `void que_destroy( QueueADT queue );`
  Destroy the specified queue, deallocating any dynamic storage.
- `void que_clear( QueueADT queue );`
  Empty out the queue so that it becomes empty.
- `void que_insert( QueueADT queue, void * data );`
  Insert a data values into the queue.
- `void * que_remove( QueueADT queue );`
  Remove the first item in the queue.
- `bool que_empty( QueueADT queue );`
  Return whether or not the queue was empty.

See the documentation in the `queueADT.h` header file for more details on the operation of these functions, including behavior under error conditions.

The queue itself is a *self-ordering* queue. Optionally associated with each queue is an ordering function which will be used by the insertion method to determine the correct position of the value being inserted within the queue. If there is no ordering function, the queue is maintained in standard FIFO (first-in, first-out) order, with insertions being done at the end of the queue and removals from the front.

The data item being inserted into the queue is supplied to the `que_insert()` method as a generic pointer (i.e., a `void*`). Similarly, the value returned from the `que_remove()` method is a generic pointer. Thus, the queue can contain any data item that can be cast into a `void*`.

The ordering function is provided as an argument through the parameter to the `que_create()` method; the parameter type is a function pointer. It will be the address of a function having the following prototype:

```
int name( const void *, const void * );
```

When an insertion into a queue is attempted, the insertion method must check to see if there is an ordering function associated with the queue. If the argument passed through the `cmp` parameter of the queue creation method was `NULL`), the queue is a traditional FIFO queue, and the insertion is made at the end of the queue.

If there is an ordering function, the insertion performed is a sorted insertion. The new data item is positioned in the queue according to its relationship to the items already there. The queue implementation never directly compares two data items; it relies on the ordering function to describe the relationship between two data items (e.g., the data item being inserted and one of the data items already in the queue).

The ordering function is used to compare two data items in order to determine the relationship between the items in some fashion. While the queue implementation does not know the actual type of the queue contents, the ordering function must know, and therefore the function is able to properly compare the values of its parameters.

The return value from the ordering function indicates the relationship between the two parameters. If the first parameter is less than the second, the return value will be negative; if the two parameters are equal, the return value will be zero; otherwise, the return value will be positive. (This is very much like the return value from the C library `strcmp()` string comparison function.)

From the point of view of the insertion method, the queue is sorted in ascending order. That is, the element in position `i` is "less than or equal to" element at `i+1`, for all `i` less than the length of the queue. The insertion method uses the ordering function whenever it needs to compare two data items to determine their relationship. A new data item will always be inserted immediately before the first item already in the queue which is "greater than" the new item as determined by the ordering function.

Supplied Files

To retrieve the assignment materials, execute the following command from your CS account:

```
        get csci243 hw6
```

This command will copy the following files to your working directory:

- Specification for the `QueueADT` type;
- Several sample test programs (`queueTest*.c`); and
- Example output files (`stdout.*`) produced by the test programs.
- A flags files (`header.mak`) used to build the programs.

You should use the following flags to compile:

```
        CFLAGS = -std=c99 -ggdb -Wall -Wextra -Werror
```

Notice that the `-Werror` flag turns any warning into an error, and try will not accept code that does not compile and link without errors. Also note that `-pedantic` is not used because it would cause warnings about the intentional use of empty struct bodies.

Constraints

You may implement the `QueueADT` type in any way that seems appropriate, as long as your implementation uses the provided header file and matches the specification given in the `queueADT.h` header file.

Your implementation must accept any 64-bit data item given to it as data to be added to a queue, and it must be able to insert any number of data items into the queue.

If compiling and linking your program produces any warnings, these will be considered errors, and `try` will not accept your code.

Submission

If using a version control system that does not insert information into the source files, you should also submit a `revisions.txt` file to provide the necessary version and revision information.

You may also submit a `readme.txt` file to provide additional information you may wish to convey about your implementation.

DO NOT submit the `queueADT.h` header file, or any of the test programs; these will be provided by `try` itself, and attempts to submit these files will cause `try` to reject your submission.

Submit your completed `queueADT.c` file, along with any other supporting source files you created, using the following command:

```
        try grd-243 hw6-1 queueADT.c revisions.txt <otherfiles>
```

where `<otherfiles>` are the names of any other C source (`.c`) or header (`.h`) files you have created as part of your solution.

The `try` program will compile, link, and test your implementation; its test programs will not be the same as the ones that were provided.

You can verify the list of files you have submitted with this command:

```
        try -q grd-243 hw6-1
```

The `-q` option will display a list of files that were submitted, and ask “Would you like to see the contents of these files?”; answer “y”, to see the contents of the files submitted in the shell’s terminal window.

Please refer to try-summary.html for more information and guidance on using try.