ECE 361 Fall 2018
Homework #3

This assignment is done individually and is worth 100 points.

THIS ASSIGNMENT SHOULD BE COMPLETED BY 10:00 PM ON SAT, 27-OCT. WE ARE USING GITHUB CLASSROOM FOR THIS ASSIGNMENT SO MAKE YOUR FINAL COMMIT TO YOUR GITHUB PRIVATE REPOSITORY FOR THE ASSIGNMENT BEFORE THE DEADLINE. SOURCE CODE FOR YOUR PROGRAMMING SOLUTIONS SHOULD HAVE A `.c` EXTENSION. HEADER FILES SHOULD HAVE A `.h` EXTENSION. YOUR TRANSCRIPTS (LOGS) SHOULD BE SUBMITTED AS TEXT FILES (`.txt`) BY EITHER REDIRECTING THE OUTPUT FROM YOUR SHELL TO A FILE OR BY USING THE LINUX TEE COMMAND TO SEND OUTPUT TO BOTH THE CONSOLE AND A FILE (EX: $ echo "hello world" |tee helloworld.txt). THE CALL TREES CAN BE HAND DRAWN (MAKE SURE THE FIGURE IS READABLE) AND SCANNED AS A .PDF OR CREATED WITH YOUR FAVORITE DRAWING TOOL AND SAVED AS A .PDF. NAME ALL OF THE FILES IN THE REPOSITORY WITH DESCRIPTIVE NAMES. BE SURE YOUR CODE IS ORGANIZED WELL, USES MEANINGFUL VARIABLE NAMES, AND INCLUDES COMMENTS THAT AID IN UNDERSTANDING YOUR CODE.

Question 1 (40 pts): Recursion

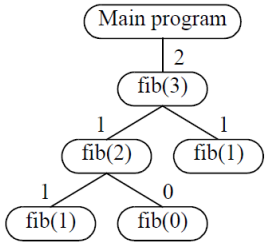
a. (5 pts) The Fibonacci sequence is:

0 1 1 2 3 5 8 13 21 . . .

Each Fibonacci number is the sum of the preceding two Fibonacci numbers. The sequence starts with the first two Fibonacci numbers and is defined recursively as:

$$\text{fib}(n) = \begin{cases} 0 & \text{if} n = 0, \\ 1 & \text{if} n = 1, \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if} n < 1. \end{cases}$$

Warford uses a call tree to describe the operation of an algorithm. Here is the call tree for `fib(3)`:



Draw the call tree in this style for `fib(4)`. How many times is `fib()` called, including the call from `main()`?

b. (5 pts) The mystery numbers are defined recursively as:

$$\text{myst}(n) = \begin{cases} 2 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ 2 \times \text{myst}(n-1) + \text{myst}(n-2) & \text{if } n > 1. \end{cases}$$

Draw the call tree for `myst(4)` in the style for `fib(3)` above. What is the value of `myst(4)`?

c. (15 pts) Write a C function:

```
int maximum (int list[], int n);
```

that recursively finds the largest integer between `list[0]` and `list[n]`. Assume at least one element is in the list. Test it with a main program (you write this program) that takes as input an integer count followed by the values. If you are not comfortable using `scanf()` you may "hardwire" a list and its size into your program. Output the original values followed by the maximum. Do not use a loop in `maximum()`. Output the value in the main program, not in the function.

Sample Input:

```
5   50 30 90 20 80
```

Sample Output:

```
Original list: 50   30   90   20   80
Largest value: 90
```

d. (15 pts) Compile and execute your program from part c. using the `gcc` command line. A single program (in a single .c file) containing the `maximum()` function and `main()` is OK. Include the source code and a transcript (log) of your output.

Question 2 (10 pts): Hash Tables

Answer the following questions about hash tables.

a. Circle all of the correct answers, -1 pt for each incorrect answer) Given the following input `(4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199)` and the hash function `x mod 10`, which of the following statements are true?
   - 9679, 1989, 4199 hash to the same value
   - 1471, 6171 hash to the same value
   - 4322, 1334, 6173 hash to the same value
   - All elements hash to the same value
   - Each element hashes to a different value

b. (5 pts) Briefly describe the difference between conflict resolution by Chaining and conflict resolution by Open Addressing. List a few advantages and disadvantages of each.

Question 3 (50 pts): Stacks and Queues

In this problem we are going to implement C modules that implement what is (hopefully) reusable functionality. Each of the modules should make use of abstraction and information hiding where appropriate and include a header file (`.h`) and a source code file (`.c`). As discussed in class, the header file should define the API for your module. You may model these modules from the Java versions I reviewed in class…only write them in C. Make use of the `static` class to control access to variables and functions that you do not want to make visible to clients of your modules.

These modules should be standalone, that is, they should not contain a `main()`. We will write a `main()` (or two `main()`'s if you want to test the modules separately) to test your modules…but not until part c.

Note: I kept the definition of these modules simple so their functionality will be somewhat limited because you can only have a single stack or a single queue (the array containing the stack or queue is declared inside the module). If you'd like to do better check out the Extra Credit at the end of the assignment.

a. (15 pts) Code a C module and a header file that encapsulates the functionality of a Stack of `doubles` using an array of 10 elements. Your stack module should implement these functions as a minimum, but you may expand the API to include additional functionality if you'd like.
   - `int push(double d)` – inserts the `d` onto the stack. Returns 1 if the insertion was successful, 0 otherwise.
   - `double pop(void)` – returns the `double` on the top of the stack and deletes it from the stack. Returns `NaN` (https://www.quora.com/What-is-NaN-in-C-programming) if the stack is empty.
   - `double peek(void)` – returns the `double` on the top of the stack but does not delete it from the stack. Returns `NaN` if the stack is empty.
   - `int isEmpty(void)` – returns 1 if the stack is empty, 0 otherwise.
   - `int isFull(void)` – returns 1 if the stack is full, 0 otherwise.
   - `int listStackContents(void)` – Displays the contents of the stack on `stdout` (the console). The entries on the stack should be listed one per line. Does not delete any entries from the stack. Returns the number of entries that were displayed.

b. (15 pts) Code a C module and a header file that encapsulates the functionality of a Queue of `char`. The Queue should be implemented as a circular array of 10 elements. Your Queue module should implement these functions as a minimum, but you may expand the API to include additional functionality if you'd like.
   - `int enqueue(char d)` – inserts the `char d` at the end of the queue. Returns 1 if the insertion was successful (i.e. the Queue was not full), 0 otherwise.

- `char dequeue(void)` – returns the `char` on the front of the queue and deletes it from the queue.
- `int isEmpty(void)` – returns 1 if the queue is empty, 0 otherwise.
- `int isFull(void)` – returns 1 if the queue is full, 0 otherwise.
- `int listQueueContents(void)` – Displays the contents of the queue on `stdout`. The entries on the stack should be listed one per line. Does not delete any entries from the queue. Returns the number of entries that were displayed.

c. (15 pts) Write a C module that includes `main()` to test your API's. You may create either a single `main()` that tests the API's of both your stack and queue or individual modules, each with their own `main()`, to test the stack and queue API's separately. Compile and execute your test program(s). Include the source code for your Stack and Queue modules and for your test program(s). Include a transcript (log) showing that your stack and queue modules work as expected.

d. (5 pts) Write a `Makefile` to control the compilation and linking of your application

EXTRA CREDIT (up to 5 points)

Create a 2nd version of either your Queue or Stack module and API that can be used to manage more than one queue or stack. You may add and make changes to the API, but make sure you document everything clearly.

<finis>