ECE 361 Fall 2019
Homework #2

This assignment is done individually and is worth 100 points.

THIS ASSIGNMENT SHOULD BE COMPLETED BY 10:00 PM ON SUN, 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. C SOURCE CODE FOR YOUR PROGRAMMING SOLUTIONS SHOULD BE TEXT FILES (NOT .DOCX, ETC.) THAT HAVE A `.c` EXTENSION. HEADER FILES SHOULD BE TEXT FILES THAT 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 BASH 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 (35 pts): Recursion

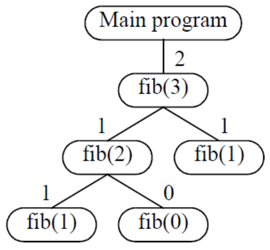
a. (5 pts) The Fibonacci sequence is:

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:

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

Call trees such as the one shown below are often used to describe the operation of a recursive 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:

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

What is the value of `myst(4)`? Show your work.

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. Your test program should try several cases besides the sample given below. You may “hardwire” the integer counts and value lists into your program instead of using `scanf()`if you prefer. Output the original values followed by the maximum number in the list. Do not use a loop in `maximum()`.

Sample Input:

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

Sample Output:

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

d. (10 pts) Compile and execute this program 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 (25 pts): Abstract Data Types

During the class meeting when we discussed linked lists (Lecture 3_2) Roy indicated that he did not feel that Karumanchi's implementation of the linked list was a good example of an Abstract Data Type (ADT). Chief amongst his complaints was that Karumanchi's algorithm created the head pointer in `main()` rather than implementing a `createList()` function in the ADT. As a result, the solution exposed details of the ADT that could have been hidden.

Fortunately, Karumanchi redeemed himself (in Roy's opinion) with his code examples for Stacks (Chapter 4) and Queues (Chapter 5) by implementing create functionality and hiding the implementation details. Now you have a chance to do the same for a linked list ADT.

a. (15 pts) Rewrite the code in `SinglyLinkedList.c` (included in the starter repository) to bring the declaration/creation of the head pointer of the list into the ADT. Consider adapting the approach that Karumanchi used in his Stack and Queue examples or you may do as Roy suggested in class and include an array of head pointers in the ADT, returning an index into that array whenever a new linked list (not a node…a new linked list) is created. Your ADT should return an indication of whether the list was created successfully (ex: a NULL pointer or an illegal index like -1 if the create list functionality did not succeed).

b. (10 pts) Rewrite `main()` in `SinglyLinkedList.c` to use your new ADT. Compile and execute the program. Include your source code and a transcript (log) showing that your new ADT works.

Question 3 (40 pts): Application Programming using Stacks and Queues

Note: this project is based on a programming project problem from "C Programming: A Modern Approach" by K.N. King but it is not identical. You may collaborate with others on the design of this solution but the code you write and the work that you submit must be your own.

Early HP calculators used a system of writing mathematical expressions known as Reverse Polish Notation (RPN). In RPN, operators (+, -, *, /, …) are placed after their operands instead of between their operands. For example, the expression 1 + 2 would be written as 1 2 + and 1 + 2 * 3 would be written as 1 2 3 * +. RPN (https://en.wikipedia.org/wiki/Reverse_Polish_notation) expressions can be easily evaluated using a stack. The algorithm involves reading the operators and operands in an expression from left to right, performing the following operations:

- When an operand is encountered, push it onto the stack
- When and operator is encountered, pop its operands from the stack, perform the operation on those operands, and then push the result back onto the stack.

a. (5 pts) Refactor Karamanchi's Dynamic Array-based stack example to separate out the Stack ADT and `main()`. Create a header (`.h`) file for the newly minted Stack ADT module. The original code for the stack ADT is included in the starter repository for this assignment.

b. (20 pts) Write a program that evaluates RPN expressions using the refactored implementation of Karumanchi's Dynamic Array-based stack module that you created in part a. The operands will be single-digit integers (`0`, `1`, `2`, …, 9). The operators are `+`, `-`, `*`, `/`, and `=`. The `=` operator causes the top stack item to be displayed. After the stack item is displayed the stack should be cleared and the user should be prompted to enter another expression. The process continues until the user enters a character that is not a valid operator or operand. For example:

```
Enter an RPN expression: 1 2 3 * + =
Value of expression: 7
Enter an RPN expression: 5 8 * 4 9 - / =
Value of expression: -8
Enter an RPN expression: q
```

Use `scanf(" %c", &ch)` to read the operators and operands.

c. (15 pts) Compile and execute your application using the `gcc` command line. Include the source code and a transcript (log) including the terminal output from several test cases. Your test cases should include the test cases above and several of your own.

<finis>