ECE 361 Fall 2020
Homework #2

THIS ASSIGNMENT SHOULD BE SUBMITTED TO D2L BY 10:00 PM ON SUN, 25-OCT-2020. THE ASSIGNMENT WILL BE GRADED AND IS WORTH 100 POINTS. IT IS EASIEST TO GRADE, AND I BELIEVE EASIEST FOR YOU SUBMIT ANY SOURCE CODE FILES AS TEXT FILES INSTEAD OF TRYING TO CUT/PASTE YOUR CODE INTO A `.doc` OR `.docx` FILE. SHORT ANSWER, TRUE/FALSE AND MULTIPLE CHOICE QUESTIONS SHOULD BE SUBMITTED IN A SINGLE TEXT OR .PDF FILE. CLEARLY NAME THE FILES SO THAT WE KNOW WHICH PROBLEM THE CODE REFERS TO AND SUBMIT THE PACKAGE AS A SINGLE .ZIP OR .RAR FILE (EX: `rkravitz_ece361f20_hw2.zip`) TO YOUR D2L HOMEWORK #2 DROPBOX .

NO LATE ASSIGNMENTS WILL BE ACCEPTED AFTER 11:59 PM ON SUN, 25-OCT BECAUSE I AM PLANNING TO REVIEW THE SOLUTIONS IN CLASS ON MON 26-OCT.

Problem 1 (30 pts) Programming a recursive algorithm

a. (10 pts) Write a C function:

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

that recursively finds the smallest integer between `list[0]` and `list[n]` and returns it as an `int`. Integers can be both positive and negative. 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
Smallest value: 20
```

b. (10) Write a test program that verifies that your function works correctly. Your `main()` should use `scanf()` and take as input an integer count followed by the values on a single line. Output the original values followed by the smallest integer. Terminate the program when the user enters a zero or negative count.

c. (10 pts) Compile and execute your program using the `gcc` command line. For this problem it is OK to submit a single `.c` file containing the `minimum()` function and `main()`. Include the source code and a transcript (log) showing several test cases with positive and negative numbers.

Problem 2 (40 pts) Application programming with stacks

Note: this problem 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 `StackWithLinkedList.c` to separate the Stack ADT and `main()` into two files. Create a header (`.h`) file for your newly minted Stack ADT module. The original code for the stack ADT is included in the starter_code folder for this assignment.

b. (20 pts) Write a program that evaluates RPN expressions using the refactored stack implementation 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.

Problem 3 (30 pts): Programming array-based ADT's

a. (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 the following functions. You may add additional functions to meet the needs of your API.

   - `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 remove any entries from the queue. Returns the number of entries that were displayed.

b. (15 pts) Write a C module that includes `main()` to test your Queue API. Compile and execute your test program using `gcc`. Include the source code for your Queue module and your test program. Include a transcript (log) showing that your queue ADT works as expected. The starter_code folder includes a simple test program that you can use or build on.