CSCI-UA.0201 – 007 Computer Systems Organization
FALL 2022

PROJECT 1

Due date: 10.02.2022, 11.59pm
Late submissions due date: 10.05.2022, 11.59pm

This is an individual work. No team work is allowed.
Similarity check will be applied to submitted codes.

BIZARRE CIPHER STACK MACHINE

In your first project, you are going to write a cipher stack machine (a dynamic stack which is implemented with dynamic memory allocation) which inserts bizarre numbers, e.g. *prime*, *perfect square*, *abundant*, *deficient*, and *perfect*, after applying some operations on them. Your program should also recover the inserted numbers. Please follow the steps given below in order to prevent any point reduction from your grade.

First of all, let's try to understand our bizarre number types. In this program, we will have five types of numbers, namely:

- Prime: a natural number that cannot be formed by multiplying other two whole numbers.
- Perfect square: result of squaring an integer.
- Abundant: a positive number whose proper divisors' sum is greater than the number itself.
    - E.g. 12. Divisors are 1,2,3,4,6. Total sum: 16. 16 > 12
    - Abundance (difference in between the sum and the number): 16-12: 4
- Deficient: a positive number whose proper divisors' sum is less than the number itself.
    - E.g. 8 (1,2,4. Total sum: 7 < 8)
    - It is clear that all prime numbers are also deficient numbers.

- Perfect: a positive number whose proper divisors's sum is equal to the number itself.
    - E.g. 28 = 1+2+4+7+14

You can find the push rules here. If the user enters a….

- prime number: push its square to the stack.
- perfect square number: take its square root until no more whole number root is found and increment the number of operations (nrOfOpr) by one each time. Push the last square root to the stack. (E.g. for 625, push 5 and increment the nrOfOpr of the node to 2)
- perfect number: directly push to the stack.
- deficient number: do not push it to the main stack but push it to a second – so called helper-stack.
- abundant number: push abundance amount to the helper stack and number-abundance to the main stack (e.g. for 12, sum is 16, abundance is 4. Push 8 to main, 4 to helper stack)

Assume that the user entered 625, 3, 8, 28 and 12 in order. Then our stacks will look like below:

Main Stack

num:8 nrOfOpr: 0 type: AB
num:28 nrOfOpr: 0 type: PF
num:-1 nrOfOpr: 0 type: “”
num:9 nrOfOpr: 0 type: PR
num:5 nrOfOpr: 2 type: SQ

Bottom

Helper Stack

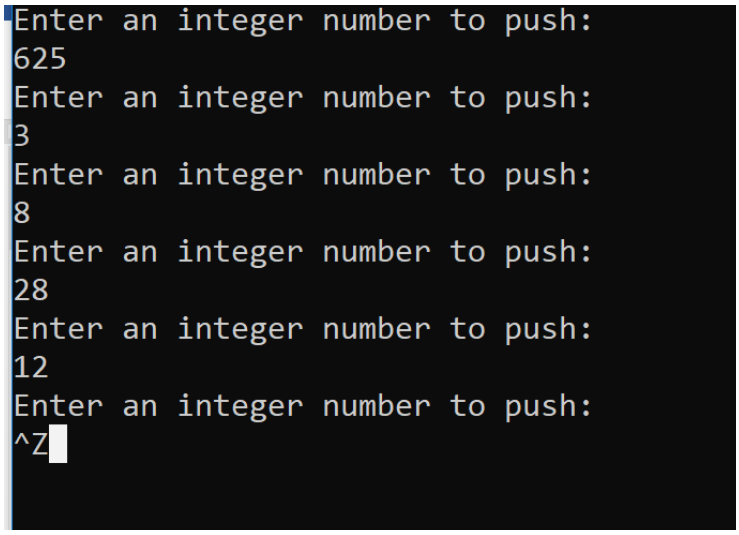
num:4 nrOfOpr: 0 type: AB
num:-1 nrOfOpr: 0 type: “”
num:8 nrOfOpr: 0 type: DF
num:-1 nrOfOpr: 0 type: “”
num:-1 nrOfOpr: 0 type: “”

Bottom

You will create one header file and two implementation files.
In main.c file:

- Create your two stacks (main and helper).
- The user will continue to enter an integer number until EOF (Ctrl+Z) is entered.
    - For each number, check its type, create a corresponding BizarreNumber and push the computed new number to the related stack. When EOF is reached, clear the screen.
- Now print the main stack and the helper stack in a simple print mode.
- Recover the original numbers using recoverCipher function. If you started your operations from the top of the stacks as usual, it means that your final stack will be in reversed order. Reverse this stack using reverseStack function and print the returned stack in detail.

SAMPLE RUN:



```
The main stack is:
TOP --> 8 --> 28 --> NULL --> 9 --> 5 --> BOTTOM

The helper stack is:
TOP --> 4 --> NULL --> 8 --> NULL --> NULL --> BOTTOM

Recovered stack is:
  **TOP**
 12, 0, AB
 28, 0, PF
  8, 0, DF
  3, 0, PR
625, 2, SQ
**BOTTOM**

Process returned 0 (0x0)   execution time : 19.220 s
Press any key to continue.
```

In your header file, declare your functions (as given below) together with your **BizarreNumber_t** and **stackNode** structs:

```
typedef struct {
    char type[3];
    int nrOfOpr;
    int num;
} BizarreNumber_t;

struct stackNode {
   BizarreNumber_t data;
   struct stackNode *nextPtr;
};
```

BizarreNumber_t struct should keep the type and the value of each number. *nrOfOpr* is initially 0 and used only for perfect square numbers while taking their square root. Type of each number is given as “SQ” (perfect square number), “PR” (prime number), “DF” (deficient number), “PF” (perfect number) or “AB” (abundant number).

Required Functions:

// stack related

- **void** push(StackNodePtr *topPtr, BizarreNumber_t info);
- BizarreNumber_t pop(StackNodePtr *topPtr);
- **int** isEmpty(StackNodePtr topPtr);
- **void** printStack(StackNodePtr currentPtr);
- **void** printStackDetailed(StackNodePtr currentPtr);
- StackNodePtr reverseStack(StackNodePtr currentPtr);

// maths related

- **int** isAbundantNumber(**int** num); //returns abundance (if 0 perfect, if > 0 abundant, if < 0 deficient)
- **int** isPrime(**int** num);
- **int** isPerfectSquare(**int** num);

//recovery related

- StackNodePtr recoverCipher(StackNodePtr mainStack, StackNodePtr helperStack);

Functions should be implemented in your implementation file (<u>not in main.c</u>). Here is the explanation of these functions:

- **void** push(StackNodePtr *topPtr, BizarreNumber_t info)
  - push a bizarre number into your stack
- BizarreNumber_t pop(StackNodePtr *topPtr);
  - pop the top item from the stack and return it
- **int** isEmpty(StackNodePtr topPtr);
  - return 1 if the stack is empty; return 0 otherwise.
- **void** printStack(StackNodePtr currentPtr);
  - print only values in one line: TOP → number of node → number of node →…….. →BOTTOM
- **void** printStackDetailed(StackNodePtr currentPtr);
  - print line by line in detail:
  - **TOP**
  - num, nrOfOpr, type of node
  - num, nrOfOpr, type of node
  - …..
  - **BOTTOM**
- StackNodePtr reverseStack(StackNodePtr currentPtr)
  - reverse the given stack pointed by currentPtr and return the pointer pointing the reversed stack, e.g. 5→3→7 becomes 7→3→5
- **int** isAbundantNumber(**int** num)
  - checks the number and returns the abundance so that if the abundance is larger than 0, the number is abundant, if equals to 0, the number is perfect, if smaller than 0, the number is deficient.

- **int** isPrime(**int** num)
    - returns 1 if the number is prime, 0 otherwise.

- **int** isPerfectSquare(**int** num)
    - returns 1 if the number is a perfect square, 0 otherwise. The only way to know if a number is a square is to check that the exponents of its prime decomposition are even. However, you can imagine more simpler, faster and practical ways.

- StackNodePtr recoverCipher(StackNodePtr mainStack, StackNodePtr helperStack)
    - checks the type of each node and recovers the number using the main stack and the helper stack, and returns the newly created recovered stack.

Any usage of built-in C functions instead of implementing above mathematical functions is strictly forbidden!!! (This does not mean that you are not allowed to use sqrt function).

All of the above functions should be fully implemented.