You have a captain's log due before 2024-04-21 (in 1 day)! Log it now! (/captain_logs/5596018/edit)

0x19. C - Stacks, Queues - LIFO, FIFO

C | Group project | Algorithm | Data structure

⚙ Weight: 2

👥 Project to be done in teams of 2 people (your team: Mohamed Madian)

📅 Project over - took place from Oct 17, 2023 6:00 AM to Oct 20, 2023 6:00 AM

☑ An auto review will be launched at the deadline

In a nutshell…

- **Contribution:** 100.0%
- **Auto QA review:** 0.0/43 mandatory & 0.0/87 optional
- **Altogether: 0.0%**
  - Mandatory: 0.0%
  - Optional: 0.0%
  - Contribution: 100.0%
  - Calculation: 100.0% * (0.0% + (0.0% * 0.0%)) == **0.0%**

Resources

Read or watch:

- Google (/rltoken/tn1X658KGumYYq_szFJI5w)

- How do I use extern to share variables between source files in C? (/rltoken/0KVWTdE8xXy__jUfBfakCw)
- Stacks and Queues in C (/rltoken/udmomL4F4mF630D2Z-ltqg)
- Stack operations (/rltoken/fj_-SJXW-pWxgAnstsARoQ)
- Queue operations (/rltoken/6Y_GVoIH_rV45xd7w0a9FA)

Learning Objectives

At the end of this project, you are expected to be able to explain to anyone (/rltoken/_kxhiyVFey_mAGnzuHKL1w), **without the help of Google**:

General

- What do LIFO and FIFO mean
- What is a stack, and when to use it
- What is a queue, and when to use it
- What are the common implementations of stacks and queues
- What are the most common use cases of stacks and queues
- What is the proper way to use global variables

Copyright - Plagiarism

- You are tasked to come up with solutions for the tasks below yourself to meet with the above learning objectives.
- You will not be able to meet the objectives of this or any following project by copying and pasting someone else's work.
- You are not allowed to publish any content of this project.
- Any form of plagiarism is strictly forbidden and will result in removal from the program.

Requirements

General

- Allowed editors: `vi`, `vim`, `emacs`
- All your files will be compiled on Ubuntu 20.04 LTS using gcc, using the options -Wall -Werror -Wextra -pedantic -std=c89
- All your files should end with a new line
- A `README.md` file, at the root of the folder of the project is mandatory
- Your code should use the `Betty` style. It will be checked using betty-style.pl (https://github.com/alx-tools/Betty/blob/master/betty-style.pl) and betty-doc.pl (https://github.com/alx-tools/Betty/blob/master/betty-doc.pl)
- You allowed to use a maximum of one global variable
- No more than 5 functions per file
- You are allowed to use the C standard library
- The prototypes of all your functions should be included in your header file called `monty.h`
- Don't forget to push your header file
- All your header files should be include guarded
- You are expected to do the tasks in the order shown in the project

There should be one project repository per group. If you clone/fork/whatever a project repository with the same name before the second deadline, you risk a 0% score.

More Info

Data structures

Please use the following data structures for this project. Don't forget to include them in your header file.

```
/**
 * struct stack_s - doubly linked list representation of a stack (or queue)
 * @n: integer
 * @prev: points to the previous element of the stack (or queue)
 * @next: points to the next element of the stack (or queue)
 *
 * Description: doubly linked list node structure
 * for stack, queues, LIFO, FIFO
 */
typedef struct stack_s
{
        int n;
        struct stack_s *prev;
        struct stack_s *next;
} stack_t;
```

```
/**
 * struct instruction_s - opcode and its function
 * @opcode: the opcode
 * @f: function to handle the opcode
 *
 * Description: opcode and its function
 * for stack, queues, LIFO, FIFO
 */
typedef struct instruction_s
{
        char *opcode;
        void (*f)(stack_t **stack, unsigned int line_number);
} instruction_t;
```

Compilation & Output

- Your code will be compiled this way:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c89 *.c -o monty
```

- Any output must be printed on `stdout`
- Any error message must be printed on `stderr`

- Here is a link to a GitHub repository (/rltoken/NUGvCZqs609VjEHeutklCw) that could help you making sure your errors are printed on `stderr`

Tests

We strongly encourage you to work all together on a set of tests

The Monty language

Monty 0.98 is a scripting language that is first compiled into Monty byte codes (Just like Python). It relies on a unique stack, with specific instructions to manipulate it. The goal of this project is to create an interpreter for Monty ByteCodes files.

Monty byte code files

Files containing Monty byte codes usually have the `.m` extension. Most of the industry uses this standard but it is not required by the specification of the language. There is not more than one instruction per line. There can be any number of spaces before or after the opcode and its argument:

```
julien@ubuntu:~/monty$ cat -e bytecodes/000.m
push 0$
push 1$
push 2$
  push 3$
                pall     $
push 4$
    push 5    $
      push   6        $
pall$
julien@ubuntu:~/monty$
```

Monty byte code files can contain blank lines (empty or made of spaces only, and any additional text after the opcode or its required argument is not taken into account:

```
julien@ubuntu:~/monty$ cat -e bytecodes/001.m
push 0 Push 0 onto the stack$
push 1 Push 1 onto the stack$
$
push 2$
  push 3$
                pall     $
$
$
                        $
push 4$
$
    push 5    $
      push   6        $
$
pall This is the end of our program. Monty is awesome!$
julien@ubuntu:~/monty$
```

- Usage: `monty file`
    - where `file` is the path to the file containing Monty byte code
- If the user does not give any file or more than one argument to your program, print the error message `USAGE: monty file`, followed by a new line, and exit with the status `EXIT_FAILURE`
- If, for any reason, it's not possible to open the file, print the error message `Error: Can't open file <file>`, followed by a new line, and exit with the status `EXIT_FAILURE`
    - where `<file>` is the name of the file
- If the file contains an invalid instruction, print the error message `L<line_number>: unknown instruction <opcode>`, followed by a new line, and exit with the status `EXIT_FAILURE`
    - where is the line number where the instruction appears.
    - Line numbers always start at 1
- The monty program runs the bytecodes line by line and stop if either:
    - it executed properly every line of the file
    - it finds an error in the file
    - an error occured
- If you can't malloc anymore, print the error message `Error: malloc failed`, followed by a new line, and exit with status `EXIT_FAILURE`.
- You have to use `malloc` and `free` and are not allowed to use any other function from `man malloc` (realloc, calloc, …)

Quiz questions

Great! You've completed the quiz successfully! Keep going! (Show quiz)

Tasks

0. push, pall
`mandatory`

Score: 0.0% (*Checks completed: 0.0%*)

Implement the `push` and `pall` opcodes.

The push opcode

The opcode `push` pushes an element to the stack.

- Usage: `push <int>`
    - where `<int>` is an integer
- if `<int>` is not an integer or if there is no argument given to `push`, print the error message `L<line_number>: usage: push integer`, followed by a new line, and exit with the status `EXIT_FAILURE`

- ○ where is the line number in the file
- (/). You won't have to deal with overflows. Use the `atoi` function

The pall opcode

The opcode `pall` prints all the values on the stack, starting from the top of the stack.

- Usage `pall`
- Format: see example
- If the stack is empty, don't print anything

```
julien@ubuntu:~/monty$ cat -e bytecodes/00.m
push 1$
push 2$
push 3$
pall$
julien@ubuntu:~/monty$ ./monty bytecodes/00.m
3
2
1
julien@ubuntu:~/monty$
```

Repo:

- GitHub repository: `monty`

[ ] Done? | Check your code | Ask for a new correction | >_ Get a sandbox | QA Review

1. pint   `mandatory`

Score: 0.0% (*Checks completed: 0.0%*)

Implement the `pint` opcode.

The pint opcode

The opcode `pint` prints the value at the top of the stack, followed by a new line.

- Usage: `pint`
- If the stack is empty, print the error message `L<line_number>: can't pint, stack empty`, followed by a new line, and exit with the status `EXIT_FAILURE`

Q

```
julien@ubuntu:~/monty$ cat bytecodes/06.m
(/)
push 1
pint
push 2
pint
push 3
pint
julien@ubuntu:~/monty$ ./monty bytecodes/06.m
1
2
3
julien@ubuntu:~/monty$
```

Repo:

- GitHub repository: `monty`

Done? | Check your code | Ask for a new correction | >_ Get a sandbox | QA Review

2. pop

Score: 0.0% (*Checks completed: 0.0%*)

Implement the `pop` opcode.

The pop opcode

The opcode `pop` removes the top element of the stack.

- Usage: `pop`
- If the stack is empty, print the error message `L<line_number>: can't pop an empty stack`, followed by a new line, and exit with the status `EXIT_FAILURE`

```
julien@ubuntu:~/monty$ cat bytecodes/07.m
(/)
push 1
push 2
push 3
pall
pop
pall
pop
pall
pop
pall
julien@ubuntu:~/monty$ ./monty bytecodes/07.m
3
2
1
2
1
1
julien@ubuntu:~/monty$
```

Repo:

- GitHub repository: `monty`

☐ Done?   Check your code   Ask for a new correction   >_ Get a sandbox   QA Review

3. swap
`mandatory`

Score: 0.0% (*Checks completed: 0.0%*)

Implement the `swap` opcode.

The swap opcode

The opcode `swap` swaps the top two elements of the stack.

- Usage: `swap`
- If the stack contains less than two elements, print the error message `L<line_number>: can't swap, stack too short` , followed by a new line, and exit with the status `EXIT_FAILURE`

```
(/)julien@ubuntu:~/monty$ cat bytecodes/09.m
push 1
push 2
push 3
pall
swap
pall
julien@ubuntu:~/monty$ ./monty bytecodes/09.m
3
2
1
2
3
1
julien@ubuntu:~/monty$
```

Repo:

- GitHub repository: `monty`

☐ Done?   Check your code   Ask for a new correction   >_ Get a sandbox   QA Review

4. add   mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Implement the `add` opcode.

The add opcode

The opcode `add` adds the top two elements of the stack.

- Usage: `add`
- If the stack contains less than two elements, print the error message `L<line_number>: can't add,` `stack too short`, followed by a new line, and exit with the status `EXIT_FAILURE`
- The result is stored in the second top element of the stack, and the top element is removed, so that at the end:
    - The top element of the stack contains the result
    - The stack is one element shorter

```
julien@ubuntu:~/monty$ cat bytecodes/12.m
(/)
push 1
push 2
push 3
pall
add
pall

julien@ubuntu:~/monty$ ./monty bytecodes/12.m
3
2
1
5
1
julien@ubuntu:~/monty$
```

Repo:

- GitHub repository: `monty`

☐ Done?   Check your code   Ask for a new correction   >_ Get a sandbox   QA Review

5. nop   mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Implement the `nop` opcode.

The nop opcode

The opcode `nop` doesn't do anything.

- Usage: `nop`

Repo:

- GitHub repository: `monty`

☐ Done?   Check your code   Ask for a new correction   >_ Get a sandbox   QA Review

🔍

Done with the mandatory tasks? Unlock 14 advanced tasks now!