Practice Arena

Practice problems aimed to improve your coding skills.

- PRACTICE-02_SCAN-PRINT
- PRACTICE-03_TYPES
- LAB-PRAC-02_SCAN-PRINT
- LAB-PRAC-01
- PRACTICE-04_COND
- BONUS-PRAC-02
- LAB-PRAC-03_TYPES
- PRACTICE-05_COND-LOOPS
- LAB-PRAC-04_COND
- LAB-PRAC-05_CONDLOOPS
- PRACTICE-07_LOOPS-ARR
- LAB-PRAC-06_LOOPS
- LAB-PRAC-07_LOOPS-ARR
- LABEXAM-PRAC-01_MIDSEM
- PRACTICE-09_PTR-MAT
- LAB-PRAC-08_ARR-STR
- PRACTICE-10_MAT-FUN
- LAB-PRAC-09_PTR-MAT
- LAB-PRAC-10_MAT-FUN
  - Stack
  - The Prutor Editor
  - Finding your identity
  - Queue
  - The Prutor Editor Part II
  - Only Ones
  - Graphs
  - How Mr C actually does Math
  - The Hidden Positives and Negatives
  - How Prutor Manages Memory
  - Message in the Matrix
  - The Hidden Key
- PRACTICE-11_FUN-PTR
- LAB-PRAC-11_FUN-PTR
- LAB-PRAC-12_FUN-STRUC
- LABEXAM-PRAC-02_ENDSEM
- LAB-PRAC-13_STRUC-NUM
- LAB-PRAC-14_SORT-MISC

Stack

LAB-PRAC-10_MAT-FUN

Stack [20 marks]

Problem Statement
The stack is a data structure that is incredibly useful in computer science. Mr C uses a stack to manage your programs. Your Windows and Linux operating systems also use stacks to perform various tasks. However, the basic premise of a stack is extremely simple and all of you encounter stacks all the time.

In the hall mess, when several plates are placed on top of each other, it is called a *stack* of plates. Note that when you want to take a plate from this stack. it is not very easy to take a plate from the bottom of the stack. Instead, what you do is take the plate at the top of the stack. Similarly, when the mess workers want to add more plates to the existing stack, they dont add new plates to the bottom of the stack since it is very difficult to do that. Instead, they simply add new plates to the top of the existing stack. Stacks are often called LIFO (last-in-first-out) structures since the last element (e.g. plate, or) that was added to the stack is the first to be remove.

The computer science stacks work exactly the same way. The stacks we will implement in this question will have three operations. The stack will have a limit size of MAX

1. Push x: if there are already MAX elements on the stack, print the error message "FULL" (without quotes), otherwise put the element x on the top of the stack.
2. Pop: If the stack is empty, print the error message "EMPTY" (without quotes), otherwise remove the element at the top of the stack and print that element.
3. Check: If the stack is empty, print "EMPTY" (without quotes), else if it is full, print "FULL" (without quotes), else print "NOT EMPTY" (without quotes)

In the first line of the input, you will be give the value of MAX as a strictly positive integer. In the next several lines, you will see instructions on what to do

1. E x: push the integer x onto the stack or print the error message. There will be a single space between the character 'E' and the integer x.
2. D: pop the element at the top of the stack and print it or print the error message
3. C: perform the check operation and print the appropriate message
4. X: no more operations to perform

Thus, the list of operations will be terminated by an X. All elements pushed onto the stack will be integers.

Caution

1. Print all error messages as well as popped elements on separate lines.
2. If you are using getchar() to process input character by character, be careful to read in the newlines at the end of each line as well.
3. We will not penalize you for extra newlines at the end of your output. However, do not have extra newlines in the middle of your output or else have trailing spaces in any line of the output.

HINTS:

1. Use an array with MAX elements to manipulate the stack. Use an integer, say idx, to store the index of the top of the stack. Initially idx can be -1 to indicate that the stack is empty. If we then push 2 then push 3, the array will look like {2, 3, ...} and idx = 1. If we now pop an element, idx will become 0 to indicate that the top of the stack is the element 2 now since 3 got popped.
2. Write functions to perform various stack operations to ensure your code looks neat and clean.

EXAMPLE:
INPUT
2
C
E 5
C
E 4
C
E 3
D
D
D
X

OUTPUT:
EMPTY
NOT EMPTY
FULL
FULL
4
5
EMPTY

Explanation

1. The stack only has a capacity of 2 elements.
2. Command 1: C - Stack is empty at the moment so print EMPTY
3. Command 2: E 5 - Push 5 into the stack. Now stack looks like [5]
4. Command 3: C - Stack is neither full nor empty so print NOT EMPTY
5. Command 4: E 4 - - Push 4 into the stack. Now stack looks like [5 4]
6. Command 5: C - Stack is full so print FULL
7. Command 6: E 3 - Stack is already full - cannot push another element so print FULL - stack still looks like [5 4]
8. Command 7: D - Pop top element 4 from stack and print 4. Now stack looks like [5]
9. Command 8: D - Pop top element 5 from stack and print 5. Now stack is empty
10. Command 9: D - Stack is already empty - cannot pop another element so print EMPTY - stack is still empty
11. Command 10: X - No more processing to do.

Grading Scheme:
Total marks: **[20 Points]**

There will be partial grading in this question. There are several lines in your output. Printing each line correctly, in the correct order, carries equal weightage. Each visible test case is worth 2 points and each hidden test case is worth 4 points. There are 2 visible and 4 hidden test cases.

Please remember, however, that when you press Submit/Evaluate, you will get a green bar only if all parts of your answer are correct. Thus, if your answer is only partly correct, Prutor will say that you have not passed that test case completely, but when we do autograding afterwards, you will get partial marks.

Start Solving! (/editor/practice/6197)