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

How Prutor Manages Memory

LAB-PRAC-10_MAT-FUN

How Prutor Manages Memory [20 marks]

Problem Statement
Prutor gets tons of requests for memory allocation from you and your friends. These can be due to your programs declaring variables, static arrays or dynamic arrays. Prutor also has to free all this memory when either your programs end or else when you release dynamically allocated memory using the free() function.

Prutor handles such requests on a first-come-first-serve basis. This is very much like a queue (or line) for food in the mess - if you arrive first in the mess, you will be the first in the queue and when food is served, you will be the first one to receive food. In the first line of the input, you will be given a strictly positive integer MAX denoting the total amount of memory available to Prutor in bytes.

In the next several lines, you will see four different kinds of instructions which are outlined below. We assure you that we will not give you more than 100 instructions. However, there may be zero or more instructions. Each instruction will be given on a separate line.

1. Memory Allocation request: If the instruction line contains the character 'M' (without quotes) followed by a space followed by a strictly positive integer n, it is a request to allocate n bytes of memory. However, do not execute this instruction just yet - just store it somewhere.
2. Memory Release request: If the instruction line contains the character 'F' (without quotes) followed by a space followed by a strictly positive integer n, it is a request to release n bytes of allocated memory. However, do not execute this instruction just yet - just store it somewhere.
3. Request execution request: If the instruction line contains the character 'X' (without quotes), this is a request to execute the earliest execution stored with you that has not been executed yet. If there are no non-executed instructions stored with you, print "NO MORE INSTRUCTIONS" (without quotes), else execute the earliest instruction.
4. Termination: If the instruction line contains the character '#' (without quotes), this means that there are no more instructions to execute and you should end your program now.

Keep in mind the following rules while processing instructions.

1. If it is a memory allocation request and there are less bytes remaining than what is requested, print the error message "NOT ENOUGH MEMORY". Otherwise, reduce the number of available bytes by the amount of memory requested, and print the message "Y BYTES LEFT" (without quotes) where Y is the number of bytes left after performing the allocation operation.
2. If it is a memory release request and the request tries to release more memory than what has so far been allocated in total, print the error message "SEGFAULT". Otherwise, increase the number of available bytes by the amount of memory released, and print the message "Y BYTES LEFT" (without quotes) where Y is the number of bytes left after performing the free operation.

Caution

1. Do not execute allocation/release instructions the moment you receive them. Execute them only when you get the execute instruction.
2. Print all error messages as well as status messages on separate lines.
3. 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.
4. 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.
5. The malloc/calloc/realloc/free process is much more careful than what we described above. However, to keep the question simple, we have not bothered you with all those details. With malloc/calloc/realloc/free, we cannot release arbitrary amounts of memory at arbitrary locations. We can only use free to release the entire chunk of memory allocated by a single malloc/calloc/realloc operation. Whereas you do not have to worry about these details in the question, do remember this when actually using malloc/calloc/realloc/free in your programs.

HINTS:

1. Use arrays with 100 elements to store instructions that have not been executed yet (there wont be more than 100 instructions). You can use a variable to store the location of the earliest instruction that has not been executed yet.
2. In order to differentiate between allocation and release instructions, you can maintain two arrays, one that stores the type of instruction 'M' or 'F' and the other that stores the amount of bytes in that instruction. Another neat way is to use a single array but store allocation instructions as positive integers and release instructions as negative integers.
3. Write functions to perform various queue operations to ensure your code looks neat and clean.

---

EXAMPLE:
INPUT

```
5
X
M 3
M 2
X
M 1
F 4
X
X
X
X
X
F 2
F 1
X
#
```

OUTPUT:

```
NO MORE INSTRUCTIONS
```

```
2 BYTES LEFT
0 BYTES LEFT
NOT ENOUGH MEMORY
4 BYTES LEFT
NO MORE INSTRUCTIONS
NO MORE INSTRUCTIONS
SEGFAULT
```

Explanation

1. There are a total of 5 bytes available. Instruction list is empty []
2. Command 1: X - no instructions to execute
3. Command 2: M 3 - instruction list is [(M,3)]
4. Command 3: M 2 - instruction list is [(M,3) (M,2)]
5. Command 4: X - execute instruction (M,3) - 2 bytes left - instruction list is [(M,2)]
6. Command 5: M 1 - instruction list is [(M,2) (M,1)]
7. Command 6: F 4 - instruction list is [(M,2) (M,1) (F,4)]
8. Command 7: X - execute instruction (M,2) - 0 bytes left - instruction list is [(M,1) (F,4)]
9. Command 8: X - execute instruction (M,1) - not enough bytes - instruction list is [(F,4)]
10. Command 9: X - execute instruction (F,4) - 4 bytes left - - instruction list is empty []
11. Command 10: X - no instructions to execute
12. Command 11: X - no instructions to execute
13. Command 12: F 2 - instruction list is [(F,2)]
14. Command 13: F 1 - instruction list is [(F,2) (F,1)]
15. Command 14: X - execute instruction (F,2) - but right now only 1 byte is allocated - segfault - instruction list is [(F,1)]
16. Command 15: #

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/6206)