1. Working with Stacks

## Reverse a String

Create a program that:

* **Reads** an **input string**
* **Reverses** it backwards (letter by latter, from the last to the first) **using a** Stack<T>
* **Prints** the result back at the console

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| I Love C# | #C evoL I |
| Stacks and Queues | seueuQ dna skcatS |

### Hints

* Use a Stack<string> and the methods **Push()**, **Pop()**.
* Push all chars from the input string, then pop and print them one by one.

## Stack Sum

Create a program that:

* **Reads** an **input of integer numbers** and **adds** them to a **stack**.
* **Reads and executes commands** until **"end"** is received.
* Process the following commands:
  + **Add <n1> <n2>**: pushes two numbers into the stack
  + **Remove <n>**: removes the n elements from the stack or does nothing if the stack holds less than **n** elements.
* **Prints** the **sum** of the remaining elements of the **stack**.

### Input

* On the **first** **line,** you will receive **an array of integers** (space-separated).
* On the **next** **lines**, until the "**end**" command is given, you will receive **commands** – a **single** **command** and **one** or **two** numbers after **the command, depending** on what **command** you are given.
  + If the **command** is "**add**", you will **always** be given **exactly two** numbers after the command, which you need to **add** to the **stack**.
  + If the **command** is "**remove**", you will **always** be given **exactly** **one** number after the command, which represents the **count** of the numbers you need to **remove** from the **stack.** If there are **not enough elements,** skip the command.
* Commands are **case-insensitive**, which means that “**Add**”, “**add**” and “**aDD**” are the same command.
* A **single space** is used as a **separator** between commands and numbers.

### Output

* When the **command** "**end**" is received, you need to **print the sum** of the **remaining** elements in the **stack**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1 2 3 4  adD 5 6  REmove 3  eNd | Sum: 6 | The stack initially holds [1, 2, 3, 4].  After the “Add 5 6” command, the stack holds [1, 2, 3, 4, 5, 6].  After the “Remove 3” command, the stack holds [1, 2, 3].  The sum of the elements [1, 2, 3] is 6. |
| 3 5 8 4 1 9  add 19 32  remove 10  add 89 22  remove 4  remove 3  end | Sum: 16 | The stack initially holds [3, 5, 8, 4, 1, 9].  The stack now holds [3, 5, 8, 4, 1, 9, 19, 32].  The command “Remove 10” is ignored (not enough elements).  The stack now holds [3, 5, 8, 4, 1, 9, 19, 32, 89, 22].  The stack now holds [3, 5, 8, 4, 1, 9].  The stack now holds [3, 5, 8].  The sum of the elements [3, 5, 8] is 16. |

### Hints

* Use a Stack<int>
* Use the methods **Push()**, **Pop()**
* Commands **may** be given in **mixed case**.

## Simple Calculator

**Create a simple calculator** that can **evaluate simple expressions** with only **addition** and **subtraction**. There will not be any parentheses. Numbers and operations are **space-separated**.

Solve the problem **using a Stack**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2 + 5 + 10 - 2 - 1 | 14 |
| 2 - 2 + 5 | 5 |

### Hints

* **Split** the input expression by space to **extract its tokens** (numbers and operations).
* **Reverse** the input tokens, then **push** them in a **Stack<string>**.
* Example:
  + Input expression: 2 + 5 + 10 - 2 - 1
  + Stack: 1 - 2 - 10 + 5 + 2
* **Pop** the last **number** (in the above example 2). It is the current result.
* **Pop** an **operation** and **number** (e. g. **+** **5**). Execute the operation. In our example: result = 2 + 5 = 7.
* **Repeat** the previous step until the stack gets empty.

## Matching Brackets

We are given an arithmetic expression with brackets. Scan through the string and extract each sub-expression.

Print the result back at the terminal.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 + (2 - (2 + 3) \* 4 / (3 + 1)) \* 5 | (2 + 3)  (3 + 1)  (2 - (2 + 3) \* 4 / (3 + 1)) |
| (2 + 3) - (2 + 3) | (2 + 3)  (2 + 3) |

### Hints

* Scan through the expression from its start to its end, searching for brackets.
  + If you find an **opening** bracket, **push its index** (position in the input expression) into the stack.
  + If you find a **closing** bracket **pop the topmost** element from the stack. This is the **index** of the **opening bracket**.
  + Use the **current** and the popped index to extract the sub-expression.

1. Working with Queues

## Print Even Numbers

Create a program that:

* **Reads** an array of **integers** and **adds** them to a **queue**.
* **Prints** the **even** numbers **separated** by "**,** ".

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4 5 6 | 2, 4, 6 |
| 11 13 18 95 2 112 81 46 | 18, 2, 112, 46 |

### Hints

* Parse the input and enqueue all the numbers in a Queue<int>.
* **Dequeue** the elements one by one and print all **even** values.

## Supermarket

You are given a **sequence of input strings**, each staying on a separate line. Each input string holds either a customer **name**, or the command “**Paid**” or the command “**End**”. Your task is to read and process the input:

* When you receive a **customer name**, add it to the queue.
* When you receive the "**Paid**" command, **print** the customer names from the queue (each at separate line), then empty the queue.
* When you receive the "**End**" command,print the count of the remaining customers from the queue in the format: "**{count} people remaining."** and stop processing the commands (see the examples below).

### Examples

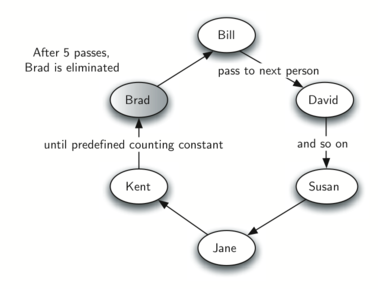
|  |  |
| --- | --- |
| **Input** | **Output** |
| Liam  Noah  James  **Paid**  Oliver  Lucas  Logan  Tiana  **End** | Liam  Noah  James  4 people remaining. |
| Amelia  Thomas  Elias  **End** | 3 people remaining. |

### Hints

Use a queue and follow the description. Just read and implement the commands.

## Hot Potato

Hot potato is a game in which **children form a circle and start passing a hot potato**. The counting starts with the first kid. **Every nth toss the child left with the potato leaves the game**. When a kid leaves the game, it passes the potato along to its next neighbor. This continues **until there is only one kid left**.

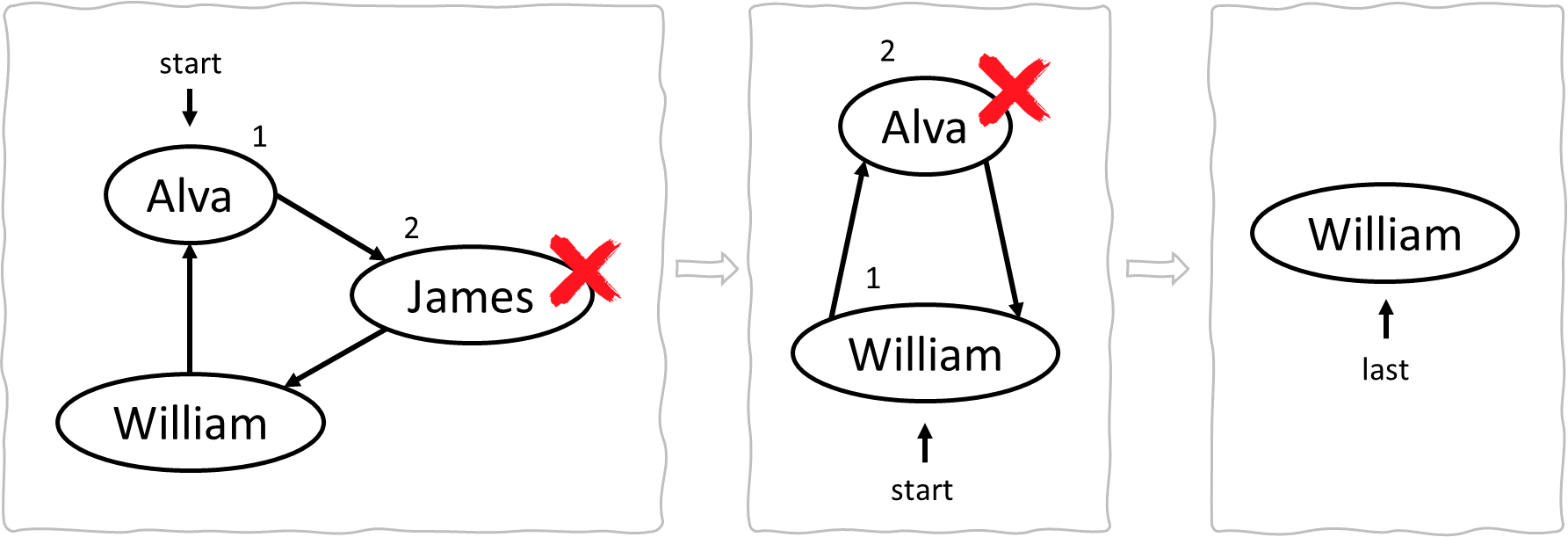


Create a program that simulates the game of Hot Potato. **Print every kid that is removed from the circle**. In the end, **print the kid that is left last**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Alva James William  2 | Removed James  Removed Alva  Last is William |
| Lucas Jacob Noah Logan Ethan  10 | Removed Ethan  Removed Jacob  Removed Noah  Removed Lucas  Last is Logan |
| Carter Dylan Jack Luke Gabriel  1 | Removed Carter  Removed Dylan  Removed Jack  Removed Luke  Last is Gabriel |

Illustration for the first example (Alva + James + William, n=2):



### Hints

* Enqueue all kids in a **Queue<string>**.
* For each round do the following:
  + (n-1) times deque an element and enqueue it again.
  + Remove an element and print it (this is the nth element).
* Repeat the above until the queue remains holding only 1 element.

## Traffic Jam

Create a program that simulates the **queue** that forms during a **traffic** **jam**. During a traffic jam, only **N** cars can **pass** the crossroads when the **light** **goes** **green**. Then the program reads the **vehicles** that **arrive** one by one and **adds** them to the **queue**. When the light **goes** **green** **N** number of cars **pass** the crossroads and **for** **each,** a **message** "**{car} passed!**" is displayed. When the "**end**" command is given, **terminate** the program and **display** a **message** with the **total** **number** of cars that **passed** the crossroads.

### Input

* On the **first** **line,** you will receive **N** – the number of cars that can pass during a green light.
* On the **next** **lines,** until the "**end**" command is given, you will receive **commands** – a **single** **string**, either a **car** or "**green**".

### Output

* Every time the "**green**" command is given, **print** **out** a message for **every** **car** that **passes** the crossroads in the format "**{car} passed!**".
* When the "**end**" command is given, **print** **out** a message in the format "**{number of cars} cars passed the crossroads.**".

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  Hummer H2  Audi  Lada  Tesla  Renault  Trabant  Mercedes  MAN Truck  green  green  Tesla  Renault  Trabant  end | Hummer H2 passed!  Audi passed!  Lada passed!  Tesla passed!  Renault passed!  Trabant passed!  Mercedes passed!  MAN Truck passed!  8 cars passed the crossroads. |
| 3  Enzo's car  Jade's car  Mercedes CLS  Audi  green  BMW X5  green  end | Enzo's car passed!  Jade's car passed!  Mercedes CLS passed!  Audi passed!  BMW X5 passed!  5 cars passed the crossroads. |