Assignment 2 ─ Stack & Queue

TA: Yuri (yuchia@alum.ccu.edu.tw)

Deadline:11:59 p.m., October 24, 2024

1. HTML Tag Validator

This is an HTML Tag Validator that you must design using stacks. Your task is to check if the HTML code is properly closed. The tool needs to ensure that all HTML tags are correctly paired with opening and closing tags and none are missing. It also needs to identify self-closing tags like
 or <img> and handle them appropriately. By implementing this tool, you will understand the importance of stacks in handling Last-In-First-Out (LIFO) problems and learn how to use a stack to track and match HTML tags.

In this assignment, you will need to design a validator using a stack, and here are some hints:

1. **Stack Creation:**
   First, use an array to implement a stack. The stack will store the opening HTML tags so they can be matched with the closing tags later.

2. **Initialization of the Stack:**
   At the beginning of the validator work, initialize the stack to empty.

3. **Tag Parsing and Matching:**
   To read the HTML code, go through the characters one by one. If you come across an **opening tag** (e.g.,), add it to the stack. When you encounter a **closing tag** (e.g.,), remove the most recent opening tag from the stack and verify if it matches the closing tag. If you encounter a **self-closing tag** (e.g.,
 or <img/>), deal with it directly without any stack operations.

4. **Validation of the Input Structure:**
   After processing all the HTML code, check if the stack is empty. If it is not empty, it indicates that there are unclosed tags, which means the HTML structure is invalid. Conversely, if the stack is empty and all tags have been matched, then the HTML structure is valid.

Input:

An HTML code without line breaks, enclosed by "<" and ">". This string may contain multiple HTML tags, including opening, closing, and self-closing tags. Tags may have attributes and spaces, but these do not affect the matching of tag names.

Output:

- Stack Operation Log:
  Push to stack: <tag>, indicates an opening tag has been pushed onto the stack.
  Pop from stack: <tag>, indicates a closing tag has matched and the corresponding opening tag has been popped from the stack.
  Self-closing tag: <tag>, indicates a self-closing tag has been encountered.
- Error Messages:
  Before popping the error tag, please pop the Pop from stack: <tag> **at first**, and display the error message.
  Error: Unclose tag - <tag>, indicates the tag <tag> was not properly closed.
- Final Output:
  HTML code is valid: true/false, indicates whether the HTML structure is valid.

--Example 1--

Input

```
<div class="container"> <h1>Welcome</h1> <img src="logo.png"/>
<p>This is a paragraph.</p></div>
```

Output

```
Push to stack: div
Push to stack: h1
Pop from stack: h1
Self-closing tag: img
Push to stack: p
Pop from stack: p
Pop from stack: div
HTML code is valid: true
```

--Example 2--

Input

```
<div class="container"> <h1>Welcome</h1> <img src="logo.png"/>
<p>This is a paragraph.<p></div>
```

Output

```
Push to stack: div
Push to stack: h1
Pop from stack: h1
Self-closing tag: img
Push to stack: p
Push to stack: p
Pop from stack: p
Error: Unclose tag - <p>
HTML code is valid: false
```

2. Queueing Problem

A supermarket has multiple cashiers, each of whom takes a different amount of time to check out customers. Each customer arrives at the checkout queue in a specified order, with some customers given priority (denoted by a 'V' prefix). In this problem, the inputs are the number of cashiers, the time each cashier takes to check out, and a list of customer names.

Please simulate the process of a customer checkout queue using a queue in the data structure and output line by line, for each point in time, the status of the customer by the cashier number.

Input:

- The first line contains an integer n, which represents the number of cashiers ($1 \leq n \leq 10$).
- The second line contains n integers representing the amount of time T_i ($1 \leq T_i \leq 100$) taken by each cashier to check out, separated by spaces. Here we assume a cashier takes the same amount of time to check out each customer.
- The last input should be a list of customer names, where some may be prefixed with 'V' to indicate they are priority customers (e.g., VE, VG). The list should end with a '0' to signal the end of the customer queue.

Output:

At each point in time, the customer status is checked against the cashier number, and the following information is output, which indicates their status of Starting Checkout/ Checkout Completed, respectively:

- "**Cashier X is checking out Customer Y**", which indicates that cashier X started checking out customer Y (the customer moved from the queue to the cashier) or is checking out customer Y.
- "**Cashier X has finished checking out for Customer Y**", which Indicates that the checkout for customer Y has been completed. This status will be displayed when the customer's checkout is complete.
- Only when the customer's status is "Checkout in Progress" will the time the cashier spends checking out the customer be counted.
- All output messages should be in time order.
- At each time point, the customer status is checked against the cashier number, and **print out the 'checking out' message before the 'finished checking out' message.**
- "**Finish**", when all the cashiers have finished checking out for customers, and there are no customers waiting to check out.

-Sample Test-

Input

```
3
1 2 3
A B VC D E VF G H 0
```

Output

```
Cashier 1 is checking out Customer VC
Cashier 2 is checking out Customer VF
Cashier 3 is checking out Customer A
Cashier 1 has finished checking out for Customer VC
Cashier 1 is checking out Customer B
Cashier 2 has finished checking out for Customer VF
Cashier 2 is checking out Customer D
Cashier 1 has finished checking out for Customer B
Cashier 3 has finished checking out for Customer A
Cashier 1 is checking out Customer E
Cashier 3 is checking out Customer G
Cashier 1 has finished checking out for Customer E
Cashier 2 has finished checking out for Customer D
Cashier 1 is checking out Customer H
Cashier 1 has finished checking out for Customer H
Cashier 3 has finished checking out for Customer G
Finish
```

Time/ cashier	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	VC	Finish VC	B	Finish B	E	Finish E	H	Finish H
2	VF	-	Finish VF	D	-	Finish D		
3	A	-	-	Finish A	G	-	-	Finish G

The table represents the timeline for the example.

3. Readme, comments, and coding style

An indicator of good source code is readability. To keep source code maintainable and readable, you should add comments to your source code where reasonable. A consistent coding style also helps a lot when tracing the source code. For this assignment, please also compose a readme file in *.txt format and name it “README.txt”. This file should contain a brief explanation of how to use your program. Please remember to have your source code comments and readme file in English.

4. Submission

To submit your files electronically, log in to the DomJudge website through the following URL: http://domjudge.csie.io:54321

Press the submit button and choose the homework questions you want to submit. After submitting your code, DomJudge will give you a result to tell you whether your code is correct or not. Please note that your code will be evaluated by different sets of test cases. Please make sure your code can work correctly based on the description above. Additionally, you must compress your code and the README file into a zip file and upload it to **Ecourse2**. Otherwise, you will get zero points.

ATTENTION: Do NOT copy others’ work or you will get a zero.

5. Grade Policies

The TA(s) will mark and give points according to the following grading policies:
50 % Problem1(HTML Tag Validator)
45 % Problem2(Queueing Problem)
5% Readme, comments, and coding style.

The readme file should include your name, class ID, a brief description of the code, and other issues students think will be helpful for the TAs to understand their homework.