IKI10100I: Data Structures & Algorithms 2015-16
Faculty of Computer Science, Universitas Indonesia
Tutorial 7 (Week 8): Stack
Deadline: Monday, 28th March 2016 18:00 WIB

Implementing Stack

The code template (`tutorial7.py`) contains a node-of-linked-list implementation from previous tutorial (i.e. `ListNode` class)andacodeskeletonforimplementingstackdatastructure,whichis defined in `MyStack` class definition. The class has two empty methods, `push(d)` and `pop()`, that represent stack's push and pop operations.

Your task is to implement `MyStack` class as a stack data structure that uses singly linked list as backingcollectionfortheitems.The `push(d)` and `pop()` should behave as one would expect from push and pop operations in any ordinary implementation of stack data structure.

Implementing a simple HTML document validator

An HTML document consists of series of tags that enclose an element that could be a string value or another tags. The tags enclose an element by having an opening tag before the element and a closing tag after the element. The format of opening and closing tags are `<name>` and `</name>` where `name` is the name of the tag. Notice that the closing tag has an extra backslash symbol before the name of the tag.

An example of HTML document is as follows:

```
<html>
      <head>
            <title>Simple HTML Document</title>
      </head>
      <body>
            <h1>Hello, Depok!</h1>
            <p>How are you?</p>
      </body>
</html>
```

It can be seen that each tags have its opening and closing tags enclosing another elements. <html> tags enclose <head> and <body> tags, where each of them enclose another tags. Since every opening tags have its own closing tags, the HTML document is considered `valid`. An HTML document is considered as invalid if there is at least one or more tags that do not have its pairing tag.

You are going to implement a simple HTML document validator that can recognise whether the given HTML document is valid or not. The validator only covers tags that have opening and closing tags. It does not support self-closing tag, e.g. `<input />`, `<hr />`, etc. The algorithm used to validate an HTML document is similar to the algorithm used for matching parentheses.

The pseudocode for matching parentheses is outlined as follows:

```
input : series of parentheses // e.g. [‘{‘, ‘(‘, ‘)’, ‘}’]
output : true if all parentheses matches, false otherwise

let S be an empty stack
for each parentheses p do
    if p is an opening parentheses
        push p into S
    else if p is a closing parentheses
        if S is empty
            return false // We have nothing to match
        if any element ‘popped’ from S does not match with p
            return false // Wrong type of parentheses (e.g. { == ])

if S is empty
    return true // All matches
else
    return false // Some parentheses were never matched
```

The same algorithm can be applied to check for HTML tags instead of parentheses. The Python code implementation is left for your exercise.

In this tutorial, you are given a custom HTML parser class **`MyHTMLParser`** that uses Python’s **`HTMLParser`** class for parsing the HTML document. It works by reading the document that being fed into the parser using **`feed()`** method call. While reading the document, the parser will read the document sequentially and invoke the appropriate methods internally each time it read an opening or closing tag. When the parser encountered an opening tag, the parser will call **`handle_starttag(tag, attrs)`** to handle the opening tag. Similarly, when the parser encountered a closing tag, the parser will call **`handle_endtag(tag)`**.

For example, suppose that we have the following **`HTMLParser`** implementation reading the HTML document example provided in the beginning of this document:

```
class MyHTMLParser(HTMLParser):
    def handle_starttag(tag, attrs):
        print(“Encountered opening tag: “ + tag)

    def handle_endtag(tag):
        print(“Encountered closing tag: “ + tag)

x = “example HTML document”
parser = MyHTMLParser()
parser.feed(x)
```

The first 6 lines generated by the sample program above is as follows:

```
Encountered opening tag: html
Encountered opening tag: head
Encountered opening tag: title
Encountered closing tag: title
Encountered closing tag: head
Encountered opening tag: body
```

As for the **`MyHTMLParser`** used in this tutorial, the handler functions and **`feed()`** method are left empty for your exercise. Note that the constructor initialises a stack as instance variable that uses previously defined **`MyStack`** class and a **`_valid`** Boolean variable that will store the verdict whether the read HTML document is valid or not. The **`_valid`** variable is initially set to **`True`** as we assume in the beginning that the HTML document read by the parser will be valid. The value may change when processing the closing tags and final condition of the stack after parsing every tags in the document.

You can read more about **`HTMLParser`** class and observe simple examples in Python Standard Libray documentation in the following link: http://docs.python.org/3.3/library/html.parser.html

The specifications for the methods you need to implement are as follows:

handle_starttag(tag, attrs)

This method is called by the parser each time it read an opening HTML tag. The method should simply store each tags it read into the stack. (See line 3 – 4 in the pseudocode)

handle_endtag(tag)

This method is called by the parser each time it read a closing HTML tag. The method should check the stack and check for any mismatch of the closing tag with the most recent tag in the stack. If there is any mismatches or invalid stack condition, set the **`_valid`** instance variable to **`False`** immediately and make no further processing for any remaining closing tags. It means that our current HTML document is really an invalid one. (See line 6 – 9 in the pseudocode)

feed(data)

This method calls parent's implementation to parse the given HTML data, which in turn will call **`handle_starttag(tag, attrs)`** and **`handle_endtag(tag)`** internally when encountered any corresponding HTML tags during parsing. After the method finished calling the parent's implementation, it should check the final condition of the stack to see if every tag matches and set the **`_valid`** variable to either **`True`** or **`False`**. The method should set **`_valid`** to **`True`** if the document is a valid HTML document, **`False`** otherwise. (See line 11 – 14 in the pseudocode)

Submission

Compress your .py files into a ZIP file named according to the following file naming format: **`YourNPM_Tutorial_7.zip`**. Upload the ZIP file to the provided submission slot in SCeLE. The submission deadline is **`28/3/2016 18:00`**.