CMPUT 175 - Lab 5: Stacks & Exceptions

Goal: Become familiar with a new data structure: Stack. Also practice raising and handling exceptions.

Exercise 1:

1. Download and save stack.py from eClass. This file contains implementation #2 of the Stack class covered in the lectures.

2. Modify the *pop()* and *peek()* methods so that they raise an Exception with a relevant message as a custom argument if these methods are invoked on an empty stack. The exception should **NOT** be handled in the Stack class.

3. The ADT for the Stack has been updated so that it has an additional behaviour:

   clear()

   - Removes all items currently in the stack; does nothing if the stack is currently empty.
   - It needs no parameters, and returns nothing.

   Update your Stack implementation by adding a new *clear()* method that follows the updated ADT specification.

Exercise 2:

You are tasked with creating a web browser simulator. The simulator will work the same as in Lab 4, but this time you must implement it using **two stacks**: one to enable the **back button** functionality, and one to enable the **forward button** functionality.

1. Download and save a copy of lab5_browser.py from eClass. (Be sure that you save it in the same directory as stack.py.) This file contains a main() function which controls the flow of operation of a web browser simulation – you must complete the try statement's else clause in this main() function. (*Hint: look at the code you were given for Lab 4.*) In the following steps, you will complete the functions that this main() function calls.

2. Complete *getAction()*. This function prompts the user to enter either a '=' (to enter a new website address), '<'(back button), '>' (forward button), or 'q' to quit the browser simulation. If the user enters something other than these 4 characters, an Exception should be raised with the argument 'Invalid entry.' This Exception is **NOT** handled in this function, but in main(). This function has no inputs. If no exception is raised, this function returns the valid character entered by the user (str).

3. Complete *goToNewSite()*. This function is called when the user enters '=' during getAction(). This function prompts the user to enter a new website address, and returns that address as a string. It also updates the two stacks, as appropriate. (*Hint:* you should be using the new Stack method, clear().) Note that you do not need to explicitly return the two stacks because the Stack (as we implemented it) is a mutable object – so **bck** and **fwd** are actually just aliases for the stacks called **back** and **forward** in your main function. The inputs for this function are the current website (str), a reference to the Stack holding the webpage addresses to go back to, and a reference to the Stack holding the webpage addresses to go forward to.

4. Complete *goBack()*. This function is called when the user enters '<' during getAction(). Handle any exceptions that are raised by the Stack class (i.e. when there are no webpages stored in the back history) by displaying an error message and returning the current site (str). Otherwise, the previous webpage is retrieved (and returned as a string), and the two stacks are updated as appropriate. The inputs for this function are the current website (str), a reference to the Stack holding the webpage addresses to go back to, and a reference to the Stack holding the webpage addresses to go forward to.

5. Complete *goForward()*. This function is called when the user enters '>' during getAction(). Handle any exceptions that are raised by the Stack class (i.e. when there are no webpages stored in the forward history) by displaying an error message and returning the current site (str). Otherwise, the next website is retrieved (and returned as a string), and the two stacks are updated as appropriate. The inputs for this function are the current website (str), a reference to the Stack holding the webpage addresses to go back to, and a reference to the Stack holding the webpage addresses to go forward to.

Sample run:

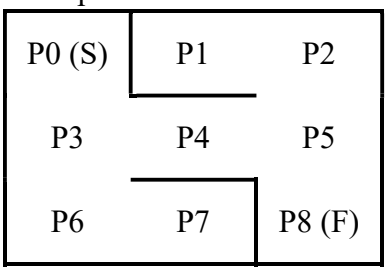
```
Currently viewing www.cs.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: 123
Invalid entry.

Currently viewing www.cs.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: >
Cannot go forward.

Currently viewing www.cs.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: <
Cannot go back.

Currently viewing www.cs.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: =
URL: www.google.ca

Currently viewing www.google.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: <

Currently viewing www.cs.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: >
```

```
Currently viewing www.google.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: =
URL: docs.python.org

Currently viewing docs.python.org
Enter = to enter a URL, < to go back, > to go forward, q to quit: <

Currently viewing www.google.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: <

Currently viewing www.cs.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: =
URL: www.beartracks.ualberta.ca

Currently viewing www.beartracks.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: >
Cannot go forward.

Currently viewing www.beartracks.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: <

Currently viewing www.cs.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: >

Currently viewing www.beartracks.ualberta.ca
Enter = to enter a URL, < to go back, > to go forward, q to quit: q
Browser closing...goodbye.
```

Optional Exercise:

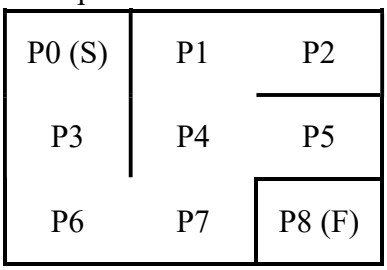
You are tasked with determining whether a maze is solvable or not by using a stack and the algorithm discussed in the lecture. However, in order to implement this algorithm, you must have a way to describe the maze so that the computer knows which moves can be made from the current position (i.e. where to move without running into a wall). One way to do that is to define a dictionary of next moves. The keys of this dictionary will represent each position in the maze. The value corresponding to each key will be a list of valid next positions that can be moved to from the key position. If there are no valid next moves, the list of values will contain an empty string. The dictionary must also contain an 'S' key whose value corresponds to the beginning position of the maze. The key position which corresponds to the end of the maze should have a value of ['F'].

Example 1: Solvable Maze




```
nextMoves = {'S': ['P0'], 'P0': ['P3'], 'P1': [''], 'P2':
['P1'], 'P3': ['P4', 'P6'], 'P4': ['P5'], 'P5': ['P2', 'P8'],
'P6': ['P7'], 'P7': [''], 'P8': ['F']}
```

Example 2: Unsolvable Maze




```
nextMoves = {'S': ['P0'], 'P0': ['P3'], 'P1': ['P2'], 'P2':
[''], 'P3': ['P6'], 'P4': ['P1', 'P5'], 'P5': [''], 'P6':
['P7'], 'P7': ['P4'], 'P8': ['F']}
```

1. Create a new file: lab5_maze.py. Import the updated Stack class from Exercise 1.

2. In lab5_maze.py, create a function to prompt the user for the name of the .txt file that stores information about a maze. You should be able to reuse (copy) your *getInputFile()* from lab 2. Place a comment in your header comment at the top of lab5_maze.py to cite where you originally wrote this function.

3. Create a function called *defineNextMoves(filename)*, which requires the name of a .txt file as input, and returns a dictionary of next moves. The dictionary should look similar to the ones shown in the examples above. The dictionary is filled using the contents of the .txt input file. Each line in the .txt file will have a key-value pair, where the key and value are separated by a colon (:). Note that multiple lines may have the same key, in which case you will have to append the new value to the existing key's list of values. Download and save maze1.txt (corresponds to example 1 above) and maze2.txt (corresponds to example 2 above) from eClass, and use these files to test your function.

4. Create a function called *solveMaze(nextMovesDict)*, which requires the dictionary created by *defineNextMoves* as input, and returns nothing. This function must use a stack to determine whether a maze can be solved or not. It should display a message saying whether the maze is solvable or not. If it can be solved, also display all of the positions visited while searching for the solution. Test your function using maze1.txt and maze2.txt and compare your output to the sample output below.

5. Draw out your own maze and write the corresponding maze3.txt file. Do you get the expected output when you use your program to try and solve it?

Sample Run (solvable maze):

```
Enter the input filename: maze1
Invalid filename extension.  Please re-enter the input filename: maze1.txt
Congratulations! You reached the end of the maze!
You visited: P0, P3, P6, P7, P4, P5, P8
```

Sample Run (unsolvable maze):

```
Enter the input filename: maze2.txt
This maze cannot be solved.
```