CS102L Data Structures & Algorithms
Lab 4

Department of Computer Science

Dhanani School of Science and Engineering

Habib University

Spring 2024

Contents

Introduction

1.1 Instructions

- This lab will contribute 1% towards your final grade.
- The deadline for submission of this lab is at the end of lab time.
- The lab must be submitted online via CANVAS. You are required to submit a zip file that contains all the *.py* files.
- The zip file should be named as *Lab_04_aa1234.zip* where *aa1234* will be replaced with your student id.
- **Files that don't follow the appropriate naming convention will not be graded.**
- **Your final grade will comprise of both your submission and your lab performance.**

1.2 Marking scheme

This lab will be marked out of 100.

- 50 Marks are for the completion of the lab.
- 50 Marks are for progress and attendance during the lab.

1.3 Lab Objectives

The objectives of this lab are:

- To understand what basic data structures are
- To understand how to program stacks and queues using lists in Python
- To use stacks and queues in different exercises

1.4 Late submission policy

There is no late submission policy for the lab.

1.5 Use of AI

Taking help from any AI-based tools such as ChatGPT is strictly prohibited and will be considered plagiarism.

1.6 Viva

Course staff may call any student for a Viva to provide an explanation for their submission.

Prerequisites

2.1 Assert Statements

Assert statements are crucial tools in programming used to verify assumptions and test conditions within code. For instance, in Python, an assert statement is typically as follows:

```
    assert condition, message
```

For example, in a function calculating the square root, in order to ensure that the input value is non-negative, one might add the following assert statement:

```
assert x >= 0, "x should be a non-negative number"
```

If the condition evaluates to False, an AssertionError is raised, highlighting the issue and its corresponding message.

2.2 Pytest

Pytest is a popular testing framework in Python known for its simplicity and power in facilitating efficient and scalable testing for applications and software. In order to use Pytest, one first needs to install it and create test functions prefixed with *test_* in a Python file. To install Pytest, you can write the following command on the terminal:

```
    pip install pytest
```

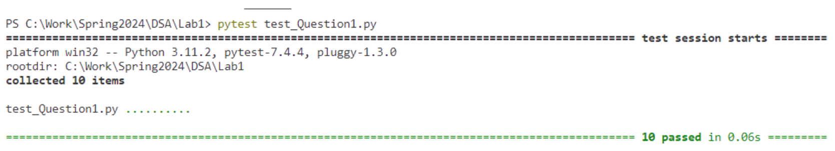
After installing Pytest, one can execute all the tests by typing the following command on the terminal:

```
    pytest
```

If you want to test any particular file, then you will have to specify the filename as follows:

```
    pytest <filename>.py
```

If all the tests pass successfully, then you should receive a similar output.

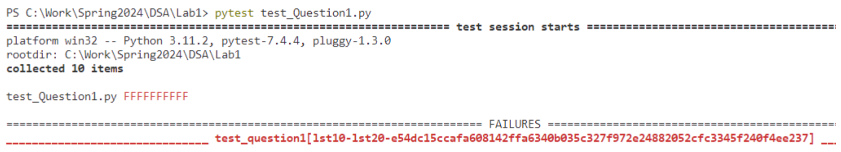


```
PS C:\Work\Spring2024\DSA\Lab1> pytest test_Question1.py
=============================================================================================== test session starts ========
platform win32 -- Python 3.11.2, pytest-7.4.4, pluggy-1.3.0
rootdir: C:\Work\Spring2024\DSA\Lab1
collected 10 items

test_Question1.py ..........

=============================================================================================== 10 passed in 0.06s =========
```

However, if all the test fail, then you should receive a similar output.



```
PS C:\Work\Spring2024\DSA\Lab1> pytest test_Question1.py
===================================================================== test session starts ========================================
platform win32 -- Python 3.11.2, pytest-7.4.4, pluggy-1.3.0
rootdir: C:\Work\Spring2024\DSA\Lab1
collected 10 items

test_Question1.py FFFFFFFFFF

========================================================================== FAILURES ================================================
_______________________________ test_question1[lst10-lst20-e54dc15ccafa608142ffa6340b035c327f972e24882052cfc3345f240f4ee237] __
```

Lab Exercises

3.1 Stack

3.1.1 Function Descriptions

Implement the following functions:

- `push(lst, item)`: Adds element item at the top of the stack, lst.
- `pop(lst)`: Removes and returns the element at top of the stack, lst.
- `top(lst)`: Returns (but doesn't remove) element at top of stack, lst.
- `is_empty(lst)`: Returns `True` if the stack, lst, is empty. Else returns `False`.

These functions should be implemented in the file `Question1.py`, accessible within the Lab 04 module on CANVAS.

3.1.2 Sample

```
>> lst = Initialize(5)  # Creating an empty stack
>> push(lst, 5) # Pushes 5 to the top of the stack
>> print(lst)
[5, None, None, None, None]
>> push(lst, 6) # Pushes 6 to the top of the stack
>> push(lst, 1) # Pushes 1 to the top of the stack
>> print(lst)
[5, 6, 1, None, None]
>> X = top(lst) # Returns the element at the top of the
   stack
>> print(X)
1
>> print(lst)
[5, 6, 1, None, None] # See how the stack remains unchanged
   after top
>> Y = pop(lst) # Removes and returns the element at the top
    of the stack
>> print(Y)
```

```
1
>> print(lst)
[5, 6, None, None, None]
>> Z = is_empty(lst) # Checkes if stack is empty
>> print(Z)
False
```

3.1.3 Testing

This exercise does not have any pytest file. The test cases given at the end of `Question1.py` should run as expected.

3.2 Queue

3.2.1 Function Descriptions

Use a list to create a queue in Python where front of the queue points to the first element and rear of the queue points to the last element of the list. Implement the following functions:

- `enQueue(lst, item)`: Adds element item at the back of the queue, lst.
- `deQueue(lst)`: Removes and returns element at front of the queue, lst.
- `front(lst)`: Returns (but doesn't remove) element at front of queue, lst.
- `is_empty(lst)`: Returns `True` if the queue, , lst, is empty. Otherwise returns `False`.

These functions should be implemented in the file `Question2.py`, accessible within the Lab 04 module on CANVAS.

3.2.2 Sample

```
>> lst = Initialize(5)  # Creating an empty queue
>> enQueue(lst, 4) # Adds 4 to the queue
>> print(lst)
[4, None, None, None, None]
>> enQueue(lst, 5) # Adds 5 to the queue
>> enQueue(lst, 2) # Adds 2 to the queue
>> print(lst)
[4, 5, 2, None, None]
>> X = front(lst) # Returns the element at the front of the
   queue
>> print(X)
4
```

```
>> print(lst) # See how the queue remains unchanged after
   front
[4, 5, 2, None, None]
>> Y = deQueue(lst) # Removes and returns the element at the
    front of the stack i.e. 4
>> print(Y)
4
>> print(lst)
[5, 2, None, None, None]
>> Z = is_empty(lst) # Checkes if stack is empty
>> print(Z)
False
```

3.2.3 Testing

This exercise does not have any pytest file. The test cases given at the end of `Question2.py` should run as expected.

3.3 Balanced Braces

3.3.1 Problem Definition

One of the most important applications of stacks is to check if the parentheses are balanced in a given expression. You are designing a compiler for programming language and need to check that braces in any given file are balanced. Braces in a string are considered to be balanced if the following criteria are met:

- All braces must be closed. Braces come in pairs of the form (), {}, and []. The left brace opens the pair, and the right one closes it.
- In any set of nested braces, the braces between any pair must be closed.

For example, [{}] is a valid grouping of braces but [}] is not.

3.3.2 Function Descriptions

Write a function, `balanced_braces`, that takes a string `s`, and returns a Boolean based on whether the string contains balanced braces or not. Utilize your stack functions from `Question1.py`.

This function should be implemented in the file `Question3.py`, accessible within the Lab 04 module on CANVAS.

3.3.3 Sample

```
>> W = balanced_braces("()")
>> print(W)
True
>> X = balanced_braces("{()}[]()")
>> print(X)
True
>> Y = balanced_braces("((")
>> print(Y)
False
>> Z = balanced_braces("{[}]")
>> print(Z)
False
```

3.3.4 Testing

In order to test your function, type the following command on the terminal:

```
pytest test_Question3.py
```

3.4 Stutter

3.4.1 Function Descriptions

You are tasked with implementing a function called `stutter` that takes a queue, A and number of copies, n as input. The function should modify the queue by replacing every element in the queue with n copies of itself. Utilize your queue functions from `Question2.py`.

This function should be implemented in the file `Question4.py`, accessible within the Lab 04 module on CANVAS.

3.4.2 Sample

```
>> X = stutter([1, 2, 3], 2)
>> print(X)
[1, 1, 2, 2, 3, 3]
>> Y = stutter(['a', 'b', 'c'], 3)
>> print(Y)
['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c']
```

3.4.3 Note

You are allowed to make a new queue to store up the results.

3.4.4 Testing

In order to test your function, type the following command on the terminal:

```
pytest test_Question4.py
```

3.5 Infix to Postfix

3.5.1 Problem Definition

In Infix notations, operators are written in-between their operands. However, in Postfix expressions, the operator comes after the operands. Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and -, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on.

- Create an empty stack called op_stack for keeping operators. Create an empty list for output.
- Convert the input infix string to a list by using the string method split.
- Scan the token list from left to right.
- 1. If the token is an operand, append it to the end of the output list.
  2. If the token is a left parenthesis, push it on the op_stack.
  3. If the token is a right parenthesis, pop the op_stack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
  4. If the token is an operator, *, /, +, or -, push it on the op_stack. However, first remove any operators already on the op_stack that have higher or equal precedence and append them to the output list. When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

3.5.2 Function Descriptions

Write a function, `Infix_to_Postfix`, that takes a string expression which is in infix format, and returns the corresponding postfix expression. Utilize your stack functions from `Question1.py`.

This function should be implemented in the file `Question5.py`, accessible within the Lab 04 module on CANVAS.

3.5.3 Sample

```
>> X = Infix_to_Postfix("A * B + C")
>> print(X)
A B * C +
>> Y = Infix_to_Postfix("A * ( B + C )")
>> print(Y)
A B C + *
>> Z = Infix_to_Postfix("( A + B ) * C - ( D - E ) * ( F + G
    )")
>> print(Z)
A B + C * D E - F G + * -
```

3.5.4 Testing

In order to test your function, type the following command on the terminal:

```
pytest test_Question5.py
```